Challenge Description and Rules:
Secure UAV Communications

MITRE | SOLVING PROBLEMS FOR A SAFER WORLD™

Contents

1 Glossary

- *Command and Control (C2) SED* – The SED that gives delivery orders to UAV SEDs
- *Drop-zone (DZ) SED* – A ground-based SED kept by the end user at the delivery destination to verify proper delivery
- *FAA Transceiver* – A device that allows the Federal Aviation Administration (FAA) to communicate with the SED, bypassing the secure SCEWL protocol
- *FAA Channel* – The communication path between the SED CPU and the FAA Transceivers via the SCEWL Bus Controller
- *Radio Waves Emulator* – The backend program that connects SEDs
- *UAV SED* – A delivery drone SED
- *SCEWL* – The secure communications protocol used by the SCEWL Network
- *SCEWL Security Server (SSS)* – The server that manages SEDs through registration and deregistration
- *SCEWL-Enabled Device (SED)* – Devices with a SCEWL Bus installed
- *SCEWL Deployment* – A discrete setup of the SSS and all SEDs (active or not)
- *SCEWL Network* – The network of actively-deployed SEDs
- *SCEWL Bus* – the system of components and code on a device that allows secure connection to, and secure communication over, the SCEWL Network
- *SCEWL Bus Controller* – The chip that controls the radio transmission hardware and is responsible for adding security to SCEWL Transmissions
- *SCEWL Bus Driver* – The software that interfaces with and drives the SCEWL Bus Controller
- *SCEWL Transmission* – A message sent from an SED to one or all active SEDs

2 Challenge Overview

2.1 Motivational Scenario

You work at a company that is developing a fleet of delivery drones, or unmanned aerial vehicles (UAVs). For this system to be successful, UAVs need to be able to securely communicate between themselves and with devices on the ground. To reuse existing low-power, long-range radio technology pioneered by your company, a new protocol will be developed named the Secure Common Embedded Wireless Link, or SCEWL for short.

Using the SCEWL protocol, UAVs and ground-based devices – collectively called SCEWL-Enabled Devices (SEDs) – form the SCEWL Network (see Figure 2 on next page). Communications across the SCEWL network is facilitated by hardware installed on SEDs, named the SCEWL Bus. The SCEWL Bus, run by a chip named the SCEWL Bus Controller, connects the CPU of an SED to the radio antenna and implements the security of the protocol. The security of an SED's communications is bootstrapped by a server located at the home base – the SCEWL Security Server (SSS).[1]

Your challenge is to design and implement the SCEWL system.

Specifically, your team will design the SCEWL protocol for communication with SEDs. You must then implement this protocol in the SCEWL Bus Controller and the SSS and create the tools to build the system.

There are several threats that your company has identified that you must handle. SCEWL should prevent attackers from being able to:

Figure 1 - Example of what a SED might look like. Image credit: "Connect Robotics Delivery Drone" by "Wikimedia Commons" licensed under Creative Commons Attribution-Share Alike 4.0 International

- Read the content of SCEWL Transmissions
- Modify the content of SCEWL Transmissions
- Replay previously sent SCEWL Transmissions
- Add a spoofed SED to the SCEWL Network

The system that your team creates must meet the requirements specified in this document and defend against as many attacks as you (and your opponents) can think of. You must design and implement a functional communications system, including the tools to build a deployment. Once your system is completed, its functionality will be tested and then it will be subjected to attacks from opposing teams, while you get a chance to attack the designs of other teams.

To enable development before physical hardware is available, **your team will develop the design using emulated hardware**.

[1] If you didn't get that all in the first read: don't worry. It will be explained in detail throughout this document.

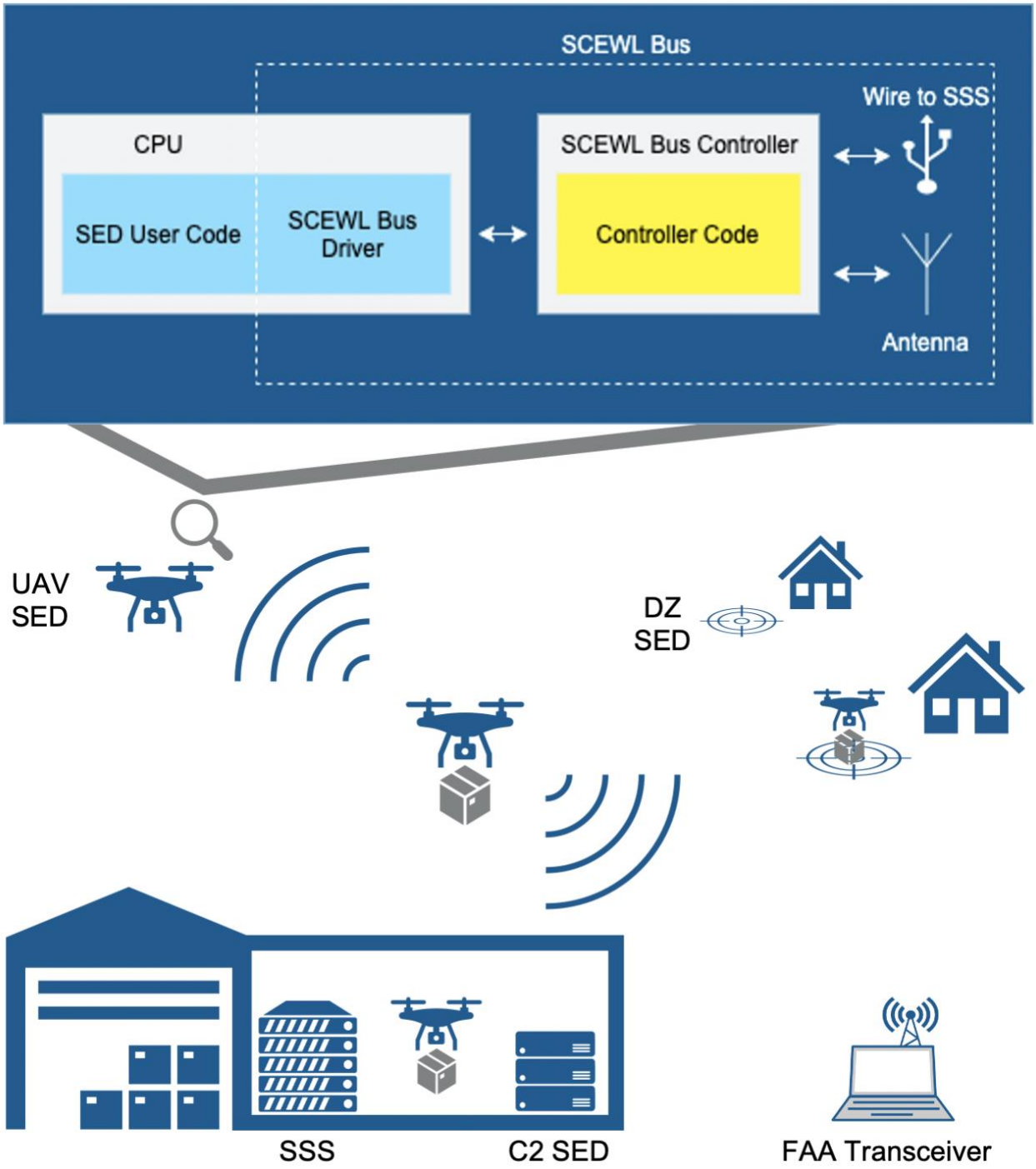


Figure 2: Overview of System (more details explained later)

2.2 Competition Phases

This is an attack-and-defend capture-the-flag with phases of the competition for both attack AND defense:

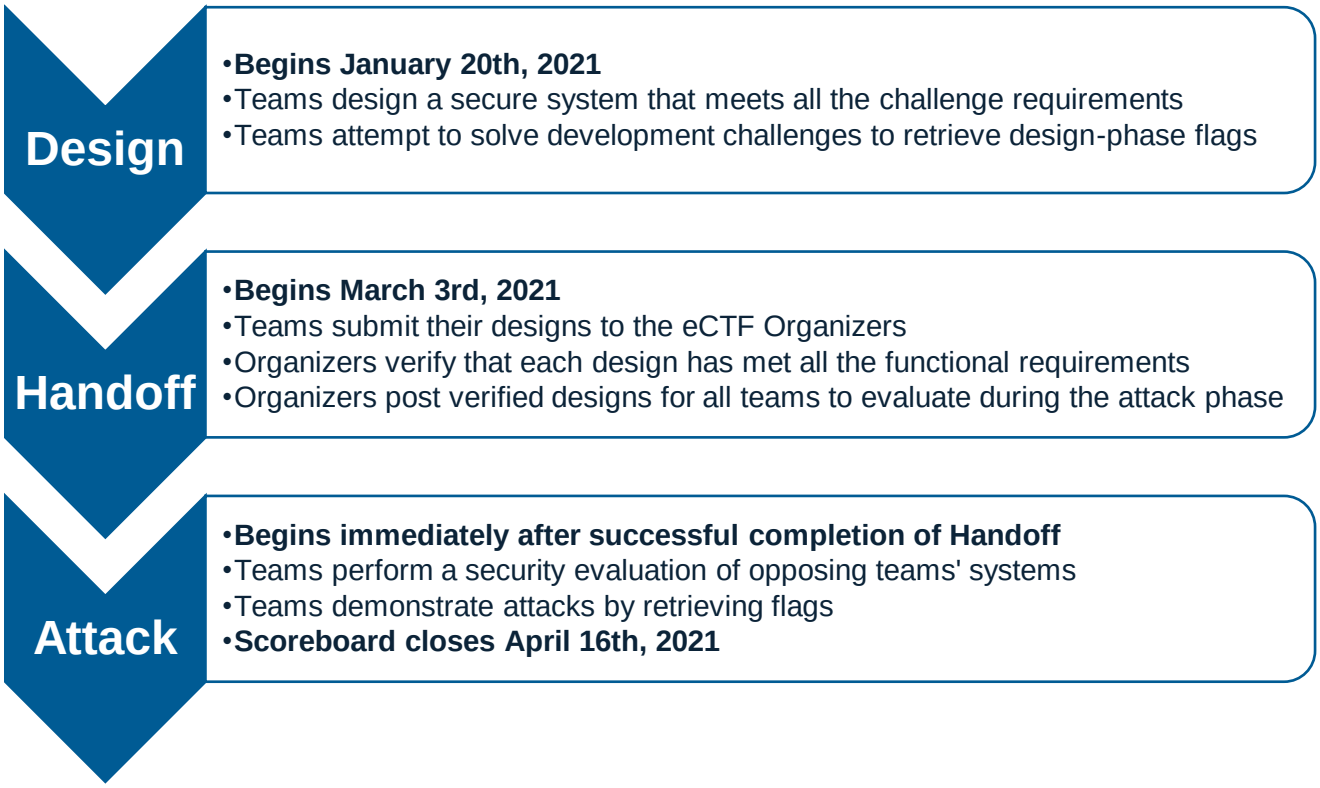


Figure 3: eCTF Phases

Positive Tip!

There are significant scoring advantages to entering the Attack phase as soon as possible. We highly recommend setting a schedule for your design and implementation and sticking to it as closely as possible. If your schedule starts to slip, your team will need to make hard decisions about what security features can be excluded to submit a working design on time (or at least as soon as possible). Final testing always takes longer than expected and sometimes reveals tricky problems – so plan to start your final testing earlier than you think is necessary.

2.2.1 Design Phase

In the Design Phase, each team must design and build a functional system. **Teams will be provided with a functional reference design that meets the functional requirements but makes no attempt at implementing security requirements**. Teams may use the reference design as a starting point or start from scratch, but the directory structure of the submitted design must be structured as detailed in 4.8 and meet all functional requirements.

During the Design Phase, teams may score points by capturing Milestone Flags, which show that teams are making progress towards having a complete design. Flags must be submitted by their deadlines for points to be awarded. Please see 7.1 for details.

NEW THIS YEAR: Teams may score points during the Design Phase in two new ways. First, teams may earn points for solving one or more of a series of reverse engineering challenges. See 7.1.2 for details. Additionally, teams may be rewarded with points for finding unintentional bugs in the reference design through a bug bounty program. See 7.1.3 for details.

2.2.2 Handoff Phase

After the completion of the Design Phase, each team must submit their design to the organizers. The Handoff phase is completed when your team submits a system and the eCTF organizers verify that your submission meets all functional requirements. Therefore, the date and time of transition between phases may vary between teams. Each team is allowed into the Attack phase by the eCTF organizers as soon as that team completes Handoff. For example: if Team-A and Team-B both submit systems on the handoff date, but only Team-A's system passes the tests, then Team-A will move to the Attack phase, while Team-B remains in the Handoff phase until they resubmit their system with the necessary fixes to pass the tests.

Submissions should include all source code, documentation, and supporting files necessary to build your system. All of this should be made available through the modification of the provided Dockerfiles, which will allow the organizers to easily build and launch your team's design.

Source code must be hosted on a public facing git repository (github, gitlab, etc). If your school provides a private git server, you are welcome to use it for development, however, you may be asked to move the source code to a public server if submission problems arise. When you are ready to submit, the organizers will provide an account that you must give access to your source code. The organizers will clone your repository and check out a commit tagged[2] "v1.0". If the organizers find minor problems that needs to be fixed without requiring a full resubmission, please increase the minor version number (e.g., from "v1.0" to "v1.1") on the resubmission. If the code was rejected due to missing functionality requirements, please increase the major version number (e.g., from "v1.1" to "v2.0").

NOTE: If using gitlab, the provided user account should be given at least "reporter" access, as any lower level will prevent cloning of your submissions.

After receiving a team's design, the eCTF organizers will provision a system and validate that the system meets the functional requirements described in this document. Any design that will not build or does not meet these requirements will not be able to progress to the attack phase of the competition. The organizers will contact each team with handoff status within three (3) **business** days after a team's submission, whether it is accepted as functional or not. As part of the testing, all designs must be able to run an attack-phase deployment continually for at least six (6) hours. If a design is not accepted, teams will be allowed to address any problems identified and submit

[2] You can create a tag named "v1.0" by running "`git tag v1.0`" once you are ready to submit, and then can push the tag by running "`git push origin v1.0`". See https://git-scm.com/book/en/v2/Git-Basics-Tagging

a revised version (with an increased version number). Teams may voluntarily pull their submission if they find functional or security flaws before the organizers have accepted the design. However, once a team's design has been accepted, they may not submit further designs (e.g., to patch security flaws that are identified after submission).

All source code (minus the .git folder) and documentation – as well as the commands used to build the deployment – will be provided to other teams during the attack phase to discourage security-by-obscurity, as well as to accelerate attack development. See 2.2.3.1 for a list of what will be provided to teams. Exemplary documentation will be worth extra points at the discretion of the eCTF organizers – see 7.4 for details.

2.2.2.1 Design Security Framework

NEW THIS YEAR: To aid with the development of a secure system, we are offering teams access to a proprietary MITRE framework. Use of it is optional but encouraged. Details will be released to teams after signing a license agreement.

2.2.3 Attack Phase

During the Attack Phase, each design that has been validated during the Handoff Phase will be available for attack. Teams will be given access to a server which hosts a live deployment of other teams' designs to attack.

2.2.3.1 Materials and Information Available to Attackers

For each design, the files listed below will be made available to all attacking teams.

- All source code (with the .git directory removed)
- Configurations used to build the attack-phase deployment
- The most recent documentation provided to the eCTF organizers
- An FAA Transceiver to receive messages sent across the FAA channel
- A basic man-in-the-middle interface that teams can adapt to intercept, modify, inject, or drop messages sent across the SCEWL Network
- The SCEWL Bus Controller binary extracted from a "downed UAV" that has since been removed from the deployment
- Interfaces to a drop zone controlled by the attacker with the ability to collect side-channel traces from its SCEWL Bus Controller

3 System Overview

3.1 SCEWL

For this challenge, your team must design and implement SCEWL. SCEWL is split into two physical[3] components: the SSS and the SED. The SSS is stationed in the home base of the UAVs where it manages security-related information and bootstraps SEDs into the SCEWL Network. The SSS and SED are connected by cable when communicating, so **attackers are not able to**

[3] Of course, this year, all components that would normally be physical are actually emulated

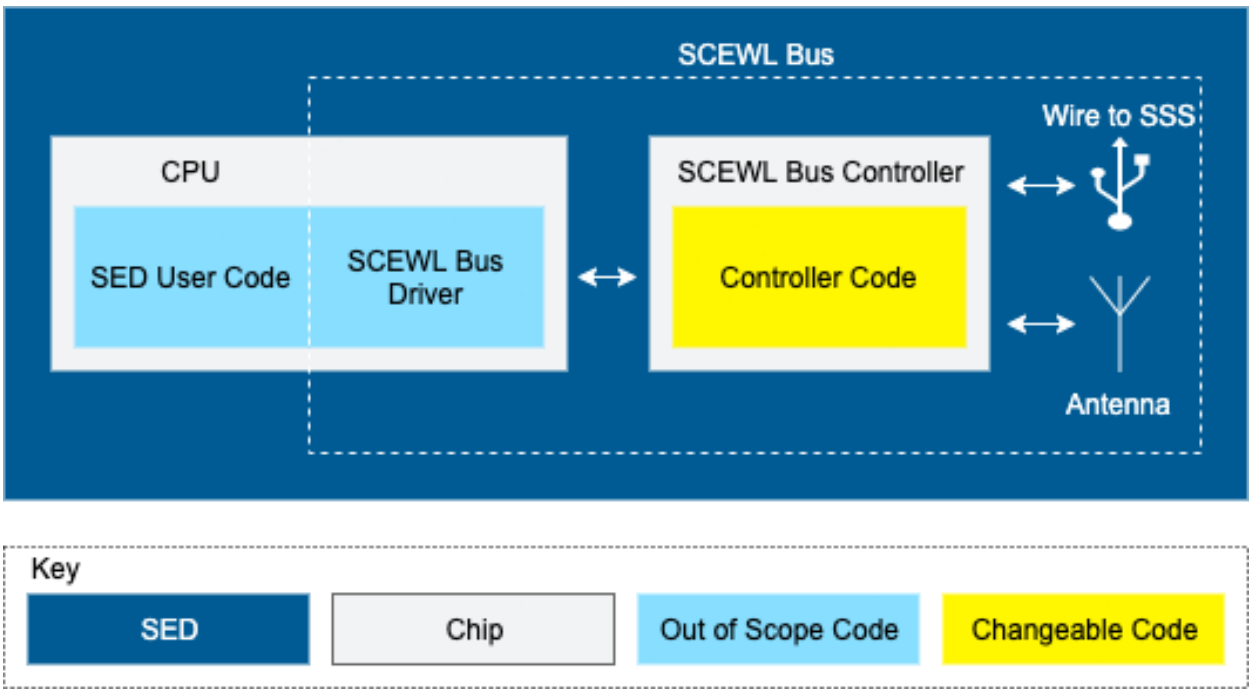


Figure 4: SED Block Diagram

access communications between the SSS and SEDs[4]. However, attackers are able to insert their own SED into the provisioning supply chain, so **the SSS should only register properly-provisioned SEDs.**

On the PCB (Printed Circuit Board) of each SED are two chips of interest to you (see Figure 4: SED Block Diagram). First is the CPU, which controls the general operation of the device. The code on it is written by other teams at the company and is out of your control. Next is the SCEWL Bus Controller, which is a small microcontroller that manages and secures incoming and outgoing radio transmissions with the SCEWL Network. Specifically, the SCEWL Bus Controller is a Stellaris LM3S6965 Microcontroller by Texas Instruments. The SED User Code interacts with the SCEWL Bus Controller through the SCEWL Bus Driver, a library compiled into the SED User Code. Your team is not allowed to modify the SED User Code or the SCEWL Bus Driver other than for your own testing purposes.

Your team must design and write the code that runs the SCEWL Bus Controller. The SCEWL Bus Controller must implement four types of communications (Figure 5: Possible Transmissions). First, the SCEWL Bus must communicate with the SSS to register and deregister itself. **Registration** must happen before any other transmission and bootstraps a SED, verifying the authenticity of the SED and bootstrapping it with whatever information is necessary to allow it

[4] For the DZ SED that will be distributed to customers' houses after registration (see Section 6), the SCEWL Bus Controller's interface to the SSS will be destroyed, preventing attackers from sending arbitrary data to the SCEWL Bus Controller's SSS interface

to join the SCEWL Network and communicate with other registered SEDs[5]. **Deregistration** removes the SED from the SCEWL Network once the SED has finished its mission and returned home. After an SED is deregistered, it will power down, however **it must be able to register again after powering back on for a new mission.**

Next, SCEWL Buses must be able to securely communicate with other registered SEDs through broadcasts and targeted transmissions. **Broadcasts** send messages securely from one SED to every other registered SED on the network. **Targeted transmissions** send messages securely from one SED to another.

[5] Already-deployed SEDs will not initially know that a new SED has been registered. However, your SCEWL protocol must provide a way for the new UAV to authenticate itself to deployed SEDs so that it may communicate with any deployed SED.

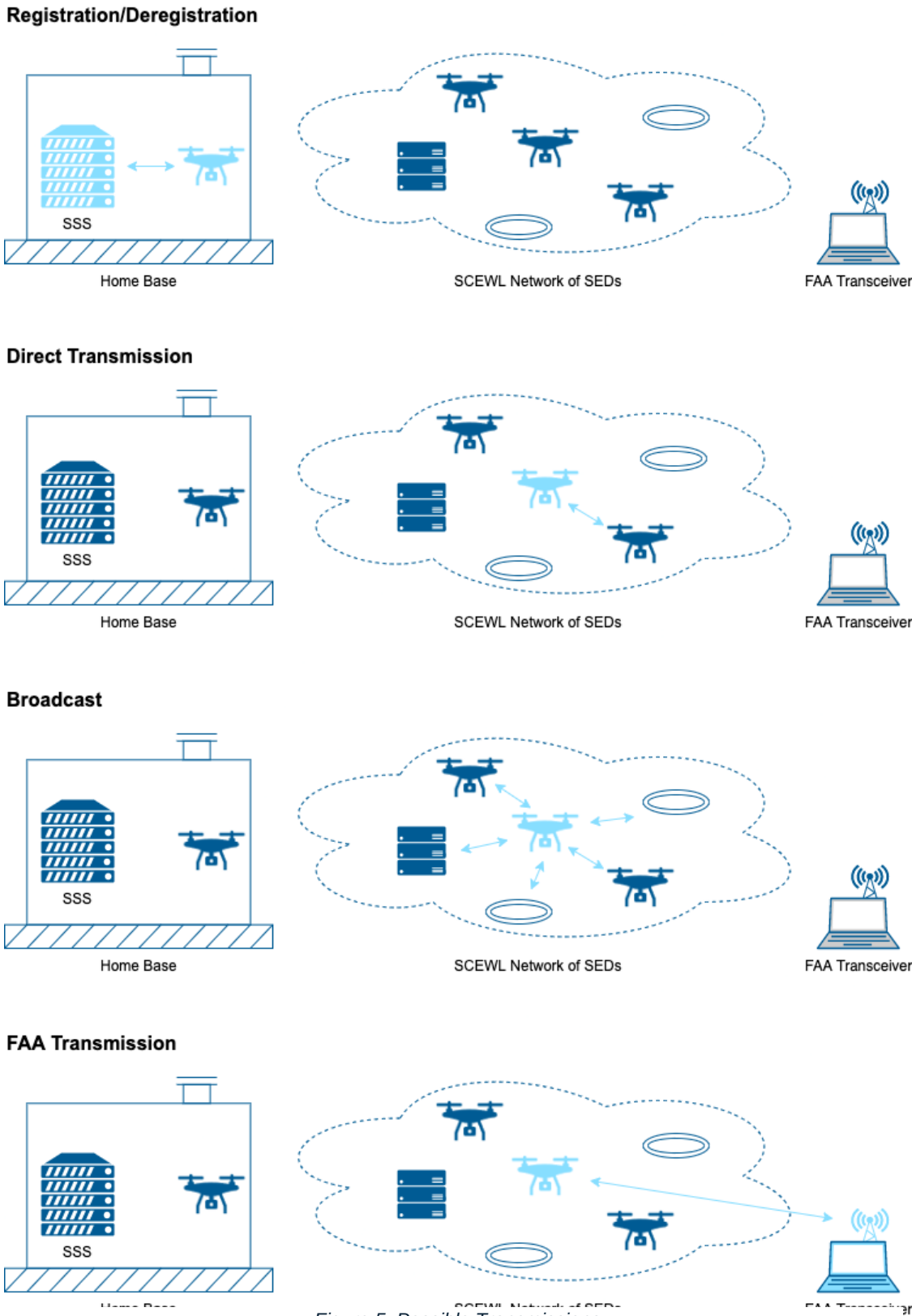


Figure 5: Possible Transmissions

The content, frequency, and type (targeted transmission or broadcast) of the SCEWL Transmissions is determined by the user code through calls to the appropriate SCEWL Bus Driver API. As these messages are sent over radio, the SCEWL Bus Controller on every SED on the SCEWL network will receive all transmissions sent from any other SED regardless of the intended recipient, (excluding communications with the SSS, as they are sent over wire). **It is the responsibility of the SCEWL Bus Controller to filter out messages not intended for its SED.**

As SCEWL is a network-level protocol, message delivery of direct transmissions and broadcasts are best-effort and do not guarantee delivery if there is underlying network loss. **You do not need to implement TCP-like recovery from network loss, but the timing requirements outlined in 4.7.1 need to be met even if there is network loss.**

Finally, due to new Federal Aviation Administration (FAA) regulations, there must be an interface for authorized FAA devices to communicate with SEDs via **FAA Transmissions**. These authorized FAA Transceivers will send and receive messages in the same format and over the same band as SCEWL Transmissions but are identified by a special SCEWL ID in the message header (see Message Format for details). FAA Transmissions may be received by the antenna or sent from the SED User Code. Authentication of the FAA messages is the responsibility of the user code and FAA Transceivers, so **the SCEWL Bus Controller should not authenticate or modify these messages and must pass them across as-is – both to and from the CPU.**

Before the system is deployed, it must first be built. **The entire build process is done while the system is not deployed, so your design does not need to handle new SEDs being created at runtime.** See 4.1 for details.

NOTE: For details on the setup of the emulation, please see Section 10.1

4 Functional Requirements

Your team is in charge of designing the SCEWL protocol, implementing the SSS and SCEWL Bus Driver, and creating the tools to build them into a full Deployment. The following sections will outline the functional requirements of the build process, the SSS, and the SCEWL Bus Driver.

4.1 Building

Before a SCEWL Deployment can be used, it must first be built according to a standardized process to aid with automation. At a high level, the SCEWL Deployment is first created and then SEDs are added and removed from it.

Adding and removing SEDs from the SCEWL Deployment is not the same as SEDs entering and leaving the SCEWL Network at runtime. SEDs can be added or removed once a deployment is built but before it is launched or after the entire deployment has been brought down. SEDs are able to enter and leave the network at runtime once the deployment is running, but a new SED cannot be added to or removed from the network (i.e. through calls to add_sed or remove_sed) until a deployment is completely spun down.

In practice, a deployment is implemented as a series of Docker containers[6] representing the radio waves emulator, the SSS, and CPU and SCEWL Bus Controller of each SED. These containers are created and modified through a series of Dockerfiles. For simplicity, the build process is automated through the Makefile in the top-level directory with a fixed API. **To customize the build process for your design, you should only change the Dockerfiles and not the top level Makefile.** As the entire build process is done inside a Docker container, **you should not need to modify or add packages to the host OS**. You may add any packages to the Docker container that you wish. If you believe you need an additional OS package, please contact the organizers.

The builds should be able to be invoked from the root directory of the repository with the root Makefile using the following commands. **See the top level README.md in the repository for a walkthrough of building a deployment**.

4.1.1 make create_radio

Create a Docker container containing from scratch containing the radio waves emulator using dockerfiles/0_create_radio.Dockerfile. This should only have to be run once ever (or if you accidentally delete the resulting “radio” container) and **its Dockerfile cannot be modified.**

NOTE: A container is prebuilt on Docker Hub tagged ectf/ectf-radio that is designed to work on the development servers we are offering, so this step is only necessary to run on other development platforms. It should be fetched automatically by Docker at runtime.

MAKE CALL:

[6] If you are unfamiliar with Docker, please follow this tutorial (https://docker-curriculum.com/) to gain a base understanding

```
$ make create_radio
```

RESULTING CALL TO DOCKER:

```
$ docker build . -f dockerfiles/0_create_radio.Dockerfile -t radio
```

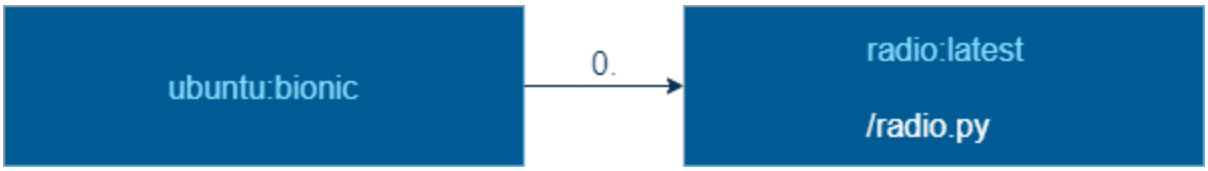
INPUTS:

- None

OUTPUTS:

- Container tagged radio:latest
  - `/radio.py`: Radio waves emulator

PROCESS:



4.1.2 make create_deployment

Create two Docker containers from scratch. First, dockerfiles/1a_create_sss.Dockerfile is called to build the initial SSS. Your team may modify this Dockerfile to build your SSS and create any deployment-wide secrets, which should be placed into /secrets in the container.

After, dockerfiles/1b_create_controller_base.Dockerfile is called to build a base container for the SCEWL Bus Controller. All packages needed for building or running the controller should be installed here. When specific SCEWL Bus Controllers are eventually built, they will inherit from this container to speed up the build process.

MAKE CALL:

```
$ make create_deployment DEPLOYMENT=<DEPL>
```

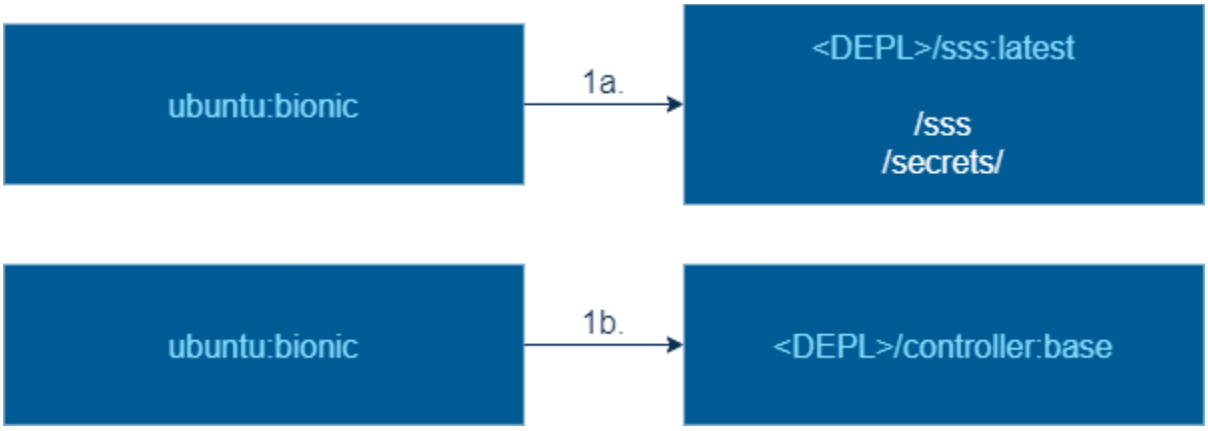
INPUTS:

- <DEPL>: The name of the deployment.

OUTPUTS:

- Container tagged <DEPL>/sss:latest
  - `/sss`: The SSS application as described in 4.3
  - `/secrets`: A directory containing any needed secrets
- Container tagged <DEPL>/controller:base
  - Any tools or files needed to build a SCEWL Bus Controller

PROCESS:



4.1.3 make add_sed

Create and modify Docker containers to add an SED to the deployment. First, the user code is built to create the CPU container using dockerfiles/2a_build_cpu.Dockerfile **which you may not modify**. Next, the already-built SSS container is modified to create any secrets for the new SED using dockerfiles/2b_create_sed_secrets.Dockerfile. Finally, the SCEWL Bus Controller is built from the base controller container with the newly generated secrets using dockerfiles/2c_build_controller.Dockerfile, its raw binary placed in its container at /controller, and its binary in ELF format at /controller.elf.

MAKE CALL:

```
$ make add_sed DEPLOYMENT=<DEPL> SED=<SED> SCEWL_ID=<SID> \
      NAME=<NAME> CUSTOM=<CUSTOM>
```

INPUTS:

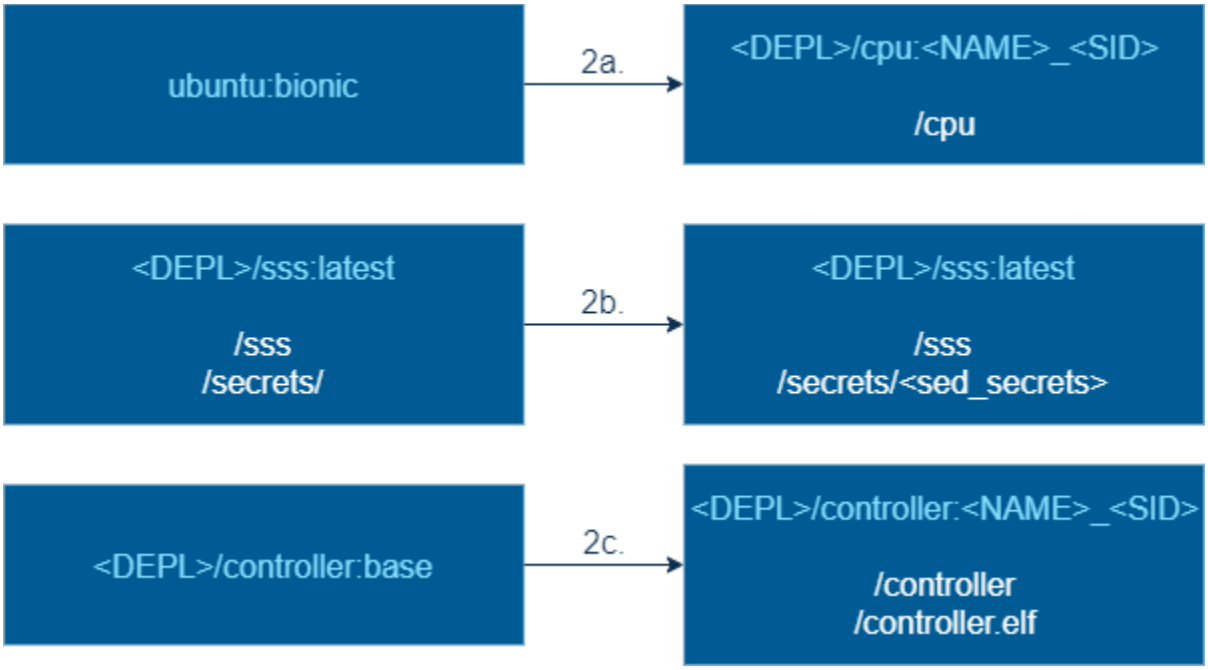
- `<DEPL>`: The name of the deployment
- `<SED>`: The name of the directory in seds/ to be built (e.g. test_echo)
- `<SID>`: The SCEWL ID of the SED
- `<NAME>`: The name of the sed; the containers will be tagged <NAME>_<SID>

- `<CUSTOM>`: A string to pass to the Makefile of the user code[7]

OUTPUTS:

- Container tagged <DEPL>/sss:latest
  - `/sss`: The SSS application as described in 4.3
  - `/secrets`: A directory containing any needed secrets
- Container tagged <DEPL>/cpu:<NAME>_<SID>
  - `/cpu`: The binary for the user code
- Container tagged <DEPL>/controller:<NAME>_<SID>
  - `/controller`: The raw binary for the SCEWL Bus Controller
  - `/controller.elf`: The binary in ELF format for the SCEWL Bus Controller

PROCESS:



4.1.4 make remove_sed

Modify and delete containers to remove an SED from the deployment. First, the CPU and controller images of the SED are deleted. Next, the SSS container is modified to remove the SED from the deployment. **REMINDER: attackers will have the extracted firmware of a "crashed UAV" that has been removed from the deployment using remove_sed; you must protect against them using those secrets to compromise the system or add their own SED to the network.**

[7] To correctly pass multiple arguments, it must be both double- and single-quoted (e.g. CUSTOM='"arg1=1 arg2=2"' or for clarity with spaces added: CUSTOM= ' " arg1=1 arg2=2 " ')

MAKE CALL:

```
$ make remove_sed DEPLOYMENT=<DEPL> SCEWL_ID=<SID> NAME=<NAME>
```

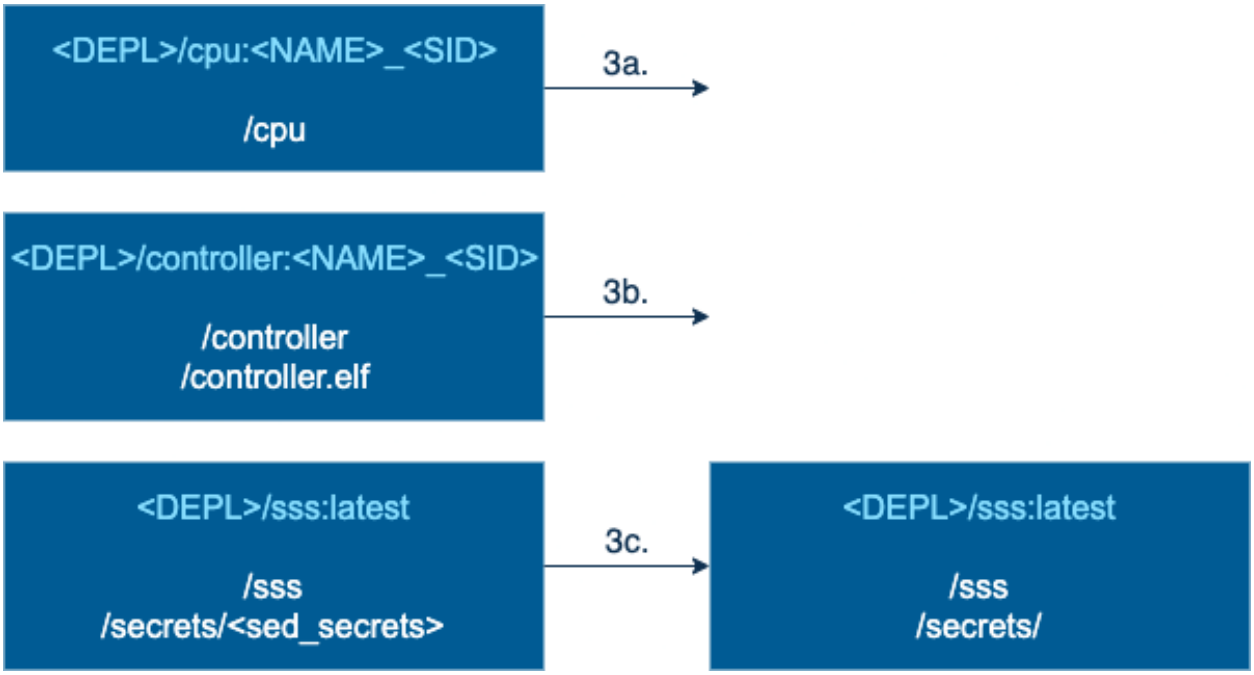
Inputs:

- `<DEPL>`: The path to the deployment directory
- `<SID>`: The SCEWL ID of the SED
- `<NAME>`: The name of the SED in the deployment

Outputs:

- Container tagged <DEPL>/sss:latest
  - `/sss`: The SSS application as described in 4.3
  - `/secrets`: The contents of the secrets directory **with the secrets from the removed SED deleted**[8]

Process:



[8] ***NOTE**: attackers will have the firmware of an SED that has been removed from the deployment using remove_sed and will also be able to attempt to add spoofed SEDs to the network, so proper key management during remove_sed is critical*

4.2 Radio Waves Emulator

The radio waves emulator acts as the backend to the network, forwarding transmissions and enabling man-in-the-middle attacks through the man-in-the-middle transceiver. Teams will be provided with a man-in-the-middle interface that allows recording, adding, dropping, and modifying messages, which enables debugging network traffic during development and conducting network attacks during the attack phase. **You may not change the Radio Waves Emulator as part of your design.**

4.3 SCEWL Security Server (SSS)

The SSS manages the registration and deregistration of SEDs. As a server, the SSS runs on a full operating system with SCEWL network card that allows it to communicate with SEDs via a hardwire connection (emulated of course) by listening on a socket (**NOTE**: the emulated setup requires the use of UNIX domain sockets, not TCP sockets). Since the SSS only communicates to SEDs via a physical connection, it can only do so when the SED is located at the homebase / warehouse. The SED should not rely on the SSS for communication to other SED when it is flying a mission[9].

The SSS may be implemented in a language of your team's choosing[10], however the reference implementation is done in Python.

The SSS may store or transmit to SEDs any information of your choosing. This may include encryption keys, entropy, or any other data necessary. The SSS is provided for allowing flexibility in your team's design; it is possible that the registration and deregistration protocols may be trivial. You may also store persistent variable secrets in a configuration file of your choice in `/secrets/` in the SSS Docker container. **Attacking teams will not have access to the content of communications between the SSS and SEDs. However, they will be able to launch their own spoofed SEDs onto the network, so the SSS must only register properly-provisioned SEDs.**

The SSS must conform to the following command line interface:

```
./sss <SOCKF>
```

INPUTS:

- `<SOCKF>`: The path to the socket to bind the SSS to

[9] In your dev environment during the design phase, you might notice that the SED is able to communicate to the SSS at all times. Teams should be careful to not rely on SSS communication for anything other than an initial registration and final deregistration – otherwise they will not pass testing during Handoff and not pass into the Attack phase.

[10] You may use any reasonable language to code the SSS, however the language should not be chosen with the intention of obfuscating the code. Please ask the organizers if you have any questions.

LAUNCHING:

The SSS will be launched in a Docker container with the Radio Waves Emulator using the following command. **As with building, the top-level Makefile must not be modified.**

```
make deploy DEPLOYMENT=<DEPLOYMENT> SSS_SOCK=<SSS_SOCK> FAA_SOCK=<FAA_SOCK>
MITM_SOCK=<MITM_SOCK> START_ID=<START_ID> END_ID=<END_ID>
SOCK_ROOT=<SOCK_ROOT>
```

INPUTS:

- `<DEPLOYMENT>`: The path to the deployment directory
- `<SSS_SOCK>`: The name of the socket for the SSS to open
- `<FAA_SOCK>`: The name of the socket the radio waves emulator will open for the FAA to connect to
- `<MITM_SOCK>`: The name of the socket the radio waves emulator will open for the MitM transceiver to connect to
- `<START_ID>`: The start of the range of possible SCEWL IDs
- `<END_ID>`: The end of the range of possible SCEWL IDs
- `<SOCK_ROOT>`: The path to the directory where sockets should be opened

4.4 SCEWL-Enabled Devices

An SED is comprised of two components: the CPU (running the SED User Code and SCEWL Bus Driver) and the SCEWL Bus Controller (see Figure 4: SED Block Diagram). **Your team only is responsible for designing and building the code that runs the SCEWL Bus Controller.**

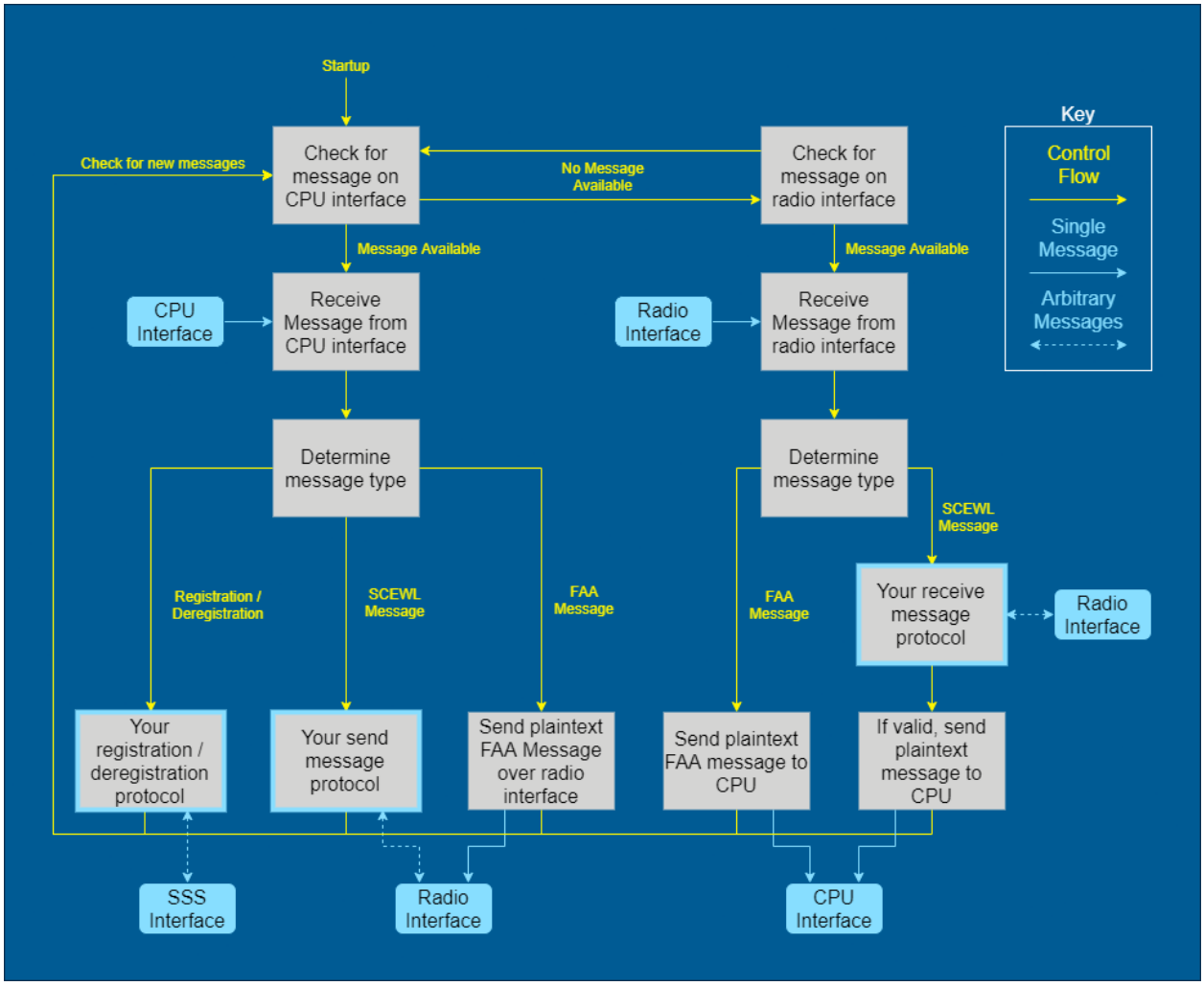
The SCEWL Bus Controller must run in an infinite loop, shuttling messages between the three interfaces: the CPU Interface connecting to the CPU, the SSS Interface connecting to the SSS, and the Radio Interface connecting to the SCEWL Network and FAA Transceiver (see Figure 6: SED State Flow Diagram). The protocols for communicating with other SEDs through direct transmission and broadcast and with the SSS are completely up to your discretion and may comprise multiple messages back and forth, however anything sent over the Radio Interface must conform to the message format described in 4.6 Messages to and from the FAA interface must be passed across with no verification or modification. If there is network loss or corruption, you may drop the offending packet(s), however **the SCEWL Bus Controller may never deadlock.**

Launching:

```
$ make launch_sed DEPLOYMENT=<DEPL> SCEWL_ID=<SID> NAME=<NAME>
SOCK_ROOT=<SOCK_ROOT>
```

Inputs:

- `<DEPL>`: The path to the deployment directory



- `<SID>`: The device ID (port number on the wireless emulator) of the SED
- `<NAME>`: The name of the SED
- `<SSS_SOCK>`: The name of the SSS socket
- `<SOCK_ROOT>`: The path to the directory where sockets should be opened

4.5 FAA Transceiver

The FAA Transceiver allows the FAA to communicate with SEDs per new UAV regulation. The transceiver may send and receive messages from any device. Once launched, the transceiver will display any message received and will provide a command prompt to send messages to SEDs. **NOTE: Although it's possible that an attacker may spoof an FAA transceiver, protection against that style of attack is up to the user code on the CPU which is outside**

Figure 6: SED State Flow Diagram

of your control. The SCEWL Bus Controller must pass all FAA messages along to the CPU without authentication[11].

API:

```
$./tools/faa.py <FAA_SOCK>
```

Inputs:

- `<FAA_SOCK>`: The path to the FAA socket created by the radio waves emulator

4.6 Message Format

While the format of the SCEWL messages is up to your team's discretion, they must be formatted into a frame expected by the emulation backend. Each frame has eight bytes of metadata followed by the frame body. The first two are the characters 'S' and 'C' followed by three 16-bit values in little endian order, which are respectively: the destination device ID, the source device ID, and the message length in bytes. **The frame header may not be encrypted** and is required to properly rout messages in the radio wave emulator. **NOTE: the body length is the length of the following body in bytes, not necessarily the length of the plaintext data given to scewl_send or scewl_brdcst.**

char 'S'	char 'C'	uint16_t dst_id	uint16_t src_id	uint16_t body_len	char body[body_len]

Figure 7: Message Format

The following device IDs are reserved and may not be assigned to any SED:

- 0 – signifies a broadcast message
- 1 – signifies a message to/from the SSS
- 2 – signifies a message over the FAA Transceiver

4.7 Bounds requirements

The system must be able to handle the following:

- Send at least 16kB of data in a message (length of 0x4000)
- Add at least 256 SEDs to a deployment
- Support at least 16 SEDs registered and deployed at the same time
- Run throughout the attack phase
    - If a deployment is set to run the course of the attack phase, fewer than 2^{32} messages will be sent

[11] If your spidey-sense is tingling and telling you that this is dangerous, you're right! The FAA messages are a vector for attack and the User Code would be the target. That code is outside your control, so it could be vulnerable. The only thing you can do is plan for resiliency – your system should be designed such that a compromised User Code won't result in a compromised Bus Controller.

- As we want to balance testing the long-term stability of a design with speedy testing turnaround, the organizers will test a live deployment for at least 6 hours before passing it into the attack phase. If the deployment manages to pass testing, it will not be penalized for encountering a corner case and crashing during the attack phase

4.7.1 Timing Requirements

The system must adhere to the following timing/performance requirements. If your system does not meet these requirements, it will not be accepted during Handoff. The following are assuming nominal network latency, no network loss, and enough cores for the processes. **NOTE:** timing on emulated systems across servers is tricky, but we will be testing on identical servers as your development servers so it should be similar to what you experience. If you are worried about timing, we're happy to work with you to figure out what's reasonable.

Operation	Max Time for Completion
scewl_init	2 seconds
scewl_register	5 seconds
scewl_deregister	5 seconds
scewl_recv	5 seconds[12]
scewl_send	5 seconds[13]
scewl_brdcst	10 seconds[14]

Figure 8: Timing Requirements

4.8 Directory Structure Overview

The project directory is structured in the following way. **Additional files and directories may be added, but the existing file layout should not be changed.** Files in ***bold italics*** cannot be modified in a submission. You may modify them to help your development and testing, but we will use unmodified versions for testing and the attack phase deployment.

- controller/ – Contains the SCEWL Bus Controller
  - lm3s/ – Contains support files for the LM3S chip architecture
  - CMSIS/ – ARM CMSIS library and interface
  - controller{.c, .h} – Reference implementation of the SCEWL Bus Controller
  - interface{.c, .h} – Implementation of APIs to the various interfaces
  - startup_gcc.c – Startup code for the SCEWL Bus Controller
  - controller.ld – Linker script for the SCEWL Bus Controller
  - Makefile – Makefile for the SCEWL Bus Controller
  - makedefs – Common definitions for the Makefile
- ***cpu/*** – Contains the CPU code
  - ***scewl_bus_driver/*** - Contains the SCEWL Bus Driver
  - ***seds/***
    - ***common/*** *-- Common utilities for SEDs*

[12] Assuming a 16kB message is available
[13] Sending a 16kB message from a newly registered SED
[14] Sending a 16kB message from a newly registered SED with 16 SEDs in range

        - ***echo_server/*** – Echo server for testing
        - ***echo_client/*** – Echo client for testing
- dockerfiles/ – Contains the Dockerfiles to build and launch deployments
    - ***0_create_radio.Dockerfile*** – Creates the Radio Waves Emulator container
    - 1a_create_sss.dockerfile – Creates the SSS container
    - 1b_create_controller_base – Creates the base SCEWL Bus Controller image
    - 2a_build_cpu.Dockerfile – Builds the CPU of an SED
    - 2b_create_sed_secrets.Dockerfile – Modifies the SSS container to create secrets
    - 2c_build_controller.Dockerfile – Builds the SCEWL Bus Controller of an SED
    - 3_remove_sed.Dockerfile – Modifies the SSS container to remove and SED
- ***radio/*** - Contains the radio waves emulator
    - ***tools/radio_waves.py*** – Radio Wave Emulator
- sss/ - contains the SSS
    - sss.py – Reference implementation of the SSS
- tools/ - contains support tools for the system
    - ***tools/deploy_echo.py –*** Launches a deployment with the echo server and client (see README)
    - ***tools/launch_sed.sh*** – Used by the root Makefile to launch the CPU and SCEWL Bus Controller of an SED
    - ***tools/faa.py*** – The FAA Transceiver to send and receive FAA messages
    - tools/mitm.py – An example man-in-the-middle interface to intercept, modify, inject, or drop messages into the deployment. You may modify this to add behavior of your choice to either debug your system or attack another system (you do not need to submit your modifications)
- ***Makefile*** – Top level Makefile

5 Security Requirements

As SCEWL will be installed in expensive drones carrying potentially expensive (or heavy) packages through the air far away from company property, the security of SCEWL is of the utmost importance.

5.1 Confidentiality

The confidentiality of SCEWL Transmissions is essential. Your company has put millions of dollars into developing the UAV system and does not want other companies reverse engineering and stealing intellectual property. Additionally, SCEWL Transmissions may contain confidential information of either the sender or the recipient, which could result in a PR nightmare if it leaked. As such, **SCEWL Transmissions between SEDs should not be readable by anyone other than the intended device(s)**. The user code should only be able to receive broadcast or direct messages sent to it.

5.2 Integrity

Data sent and received by devices must maintain its integrity. Integrity failures could result in UAVs crashing or being redirected and their packages being stolen. To avoid this, **data returned from a call to scewl_recv must be identical to the data that was sent by**

scewl_send or scewl_brdcst. Messages that are received by the SCEWL Bus Driver that have been maliciously modified with should be discarded as if no message had been received.

5.3 Authentication

Since SCEWL sends and receives messages over radio, it is crucial that devices cannot be spoofed and that only messages from properly provisioned devices are received. Otherwise, attackers would be able to compromise the network and inject their own messages. Therefore, **scewl_recv must only return data that was sent from a properly provisioned device**. Messages that are received by the SCEWL Bus Driver that are not from a properly provisioned device should be discarded as if no message had been received.

Additionally, as an attacker may attempt to enter a spoofed device into the system, **the SSS should only register devices that have been properly provisioned.**

5.4 Replay Protection

Attackers should not be able to send messages by replaying previous messages. Therefore, **scewl_recv should not return replayed messages.** If messages are reordered, you may throw out the out-of-order messages. A deployment should provide replay protection for up to 2^{32} messages.

5.5 Defense-in-Depth

While the team developing the SED User Code promises that they will build it securely, you know that is not likely. To insulate SCEWL from their incompetence, **the SCEWL Bus Controller should expect that the SED User Code may be compromised and should remain robust even in the face of arbitrary input from the CPU.**

6 Attack Phase Deployment

NOTE: Everything described by this section is implemented by the user code. Therefore, it is outside of the scope of what you need or are permitted to modify. This section is purely for your understanding of the context in which your secure communication system will operate during the attack phase.

During the attack phase, there will be three different types of SEDs. First, there are the UAV SEDs that fly from the home base to the customer's house to deliver packages. Second, there is the command and control (C2) SED that communicates with UAVs at the beginning of their missions to give them delivery orders. Finally, there are DZ SEDs that are located at each customer's house that communicate with UAVs to verify a package was received.

The following section describes the setup and operation of the attack phase deployment:

1. The SSS launches
2. The deployment spins up
    a. The C2 and DZs power on and register with the SSS
    b. The DZs are distributed to the customers' houses
3. The UAV launches
    a. The UAV powers on and registers with the SSS

    b. The UAV communicates with the C2 to receive orders and a package
4. The UAV flies to its target
    a. Once at the beginning of flight and on a regular basis thereafter, the UAV communicates its location and its “swarm ID” in a broadcast to all other UAVs and sends its location over the FAA channel
        i. If another UAV receives the broadcast and notices it is flying at the same altitude, it will respond with a deconflict message, telling the broadcasting UAV to fly 5m higher
        ii. On receiving the deconflict message, the UAV will fly 5m higher and send another broadcast of its location, repeating the process if another UAV is already flying at that altitude
5. The UAV delivers its package
    a. On arrival at its destination, the UAV sends an SYN message to the DZ, notifying the DZ of its arrival
    b. The DZ responds with an ACK
    c. The UAV responds with its package
6. The UAV returns to its home base
    a. Similar process to 4.
7. The UAV spins down
    a. Once arriving at the home base, the UAV deregisters from the SSS
    b. The UAV powers off until its next mission
8. Triggered at any point during a UAV’s mission
    a. Recovery mode
        i. Every message between SEDs contains a plaintext checksum in the message
        ii. If a UAV receives any message with a bad checksum
            1. The UAV enters recovery mode, assuming a hardware problem
            2. The UAV sends a recovery message over the FAA channel
            3. The UAV returns to its home base, deregisters from the SSS, and powers down until its next delivery
    b. Emergency drop
        i. If UAV receives the emergency drop message
            1. The UAV sends emergency drop notification to the FAA channel
            2. The UAV drops its package
            3. The UAV returns to its home base, deregisters from the SSS, and powers down

7 Scoring

Any flag submitted to the score board will be of the form `ectf{<flag_name>_<16 hex characters>}`. For example, the "Recovery Mode" flag would be of the form '`ectf{recoverymode_0123456789abcdef}`'.

7.1 Design-Phase Flags

7.1.1 Milestone Flags

To encourage teams to stay on schedule during the design phase, and to give the organizers insight into each team's progress, points will be awarded for reaching certain milestones. Each development flag has a deadline date at which point it can no longer be submitted for points. The development flags are:

Milestone	Description / How to obtain the flag	Deadline Date[15]
Read Rules	If you read all the rules, you'll know	1/27/21
Boot Reference	Provision and boot the scewl_echo and the scewl_echo_client SEDs (as explained in the REAME) to receive a flag	1/27/21
Design Document	Submit a design document containing descriptions of how each command will work on your system. Your design may change after submitting this first draft	2/3/21
Use Debugger	Use the debugger Makefile rule to step through a binary and retrieve a flag. Details in release package	2/10/21
Bug Bounty	Find and fix a bug in the reference design	3/3/21 (handoff)

Figure 9: Design Flags

7.1.2 Reverse Engineering Challenge

New this year is the reverse engineering challenge. Teams will be given three binaries from the CPUs of SEDs to reverse engineer. Each accepts input from the FAA channel and, if given the correct input, will print a flag to the FAA channel.

Your task is to determine the correct input for each binary, launch them, and send the correct input to receive the flag. The three binaries are of increasing difficulty and each also print a hint for the next binary.

Of particular note is the third binary, which is the CPU code of the attacker-controlled drop zone. A successful exploitation of this binary will gain arbitrary code execution on the CPU.

The binaries will be provided as part of the release package and can be launched by running the following from the root of the repo:

[15] The exact cutoff will be at 11:59am (eastern time zone) on the deadline date.

```
./tools/deploy_re.sh
```

7.1.3 Bug Bounty

If your team happens to find a bug in the reference design, you can earn points for it! The bug must be something that causes the reference design to fail a functional requirement[16]. Your team will receive 100 points for each bug found, and another 100 for also submitting the corresponding fix. If multiple teams find the same bug, points will distributed on a first come, first serve basis.

7.2 Offensive Flags

Each system is required to hold and protect "flags" that should only be revealed if the system is compromised. By submitting flags, a team is demonstrating that they have compromised the target system. A brief description is required for each attack that results in a flag submission. The following table lists the flags, as well as a description of each:

Flag Name	Capturing this flag proves that you can…	Requirement	To Submit this Flag…
UAV ID Recovery	Read a broadcasted SCEWL Transmission	Confidentiality	Obtain the UAV ID from the broadcasting UAV SED
Package Recovery	Read a targeted SCEWL Transmission	Confidentiality	Obtain a package not destined for the attacker-controlled DZ, which will be sent from the C2 to the UAV and from the UAV to the DZ
Recovery Mode	Modify the content of a SCEWL Transmission	Integrity	Corrupt the plaintext of a message so that the user code's checksum check fails and the UAV goes into recovery mode. Read the flag from the FAA channel
Drop Package	Have full control of the plaintext contents of a SCEWL Transmission	Integrity / Authentication	Have the user code of any deployed UAV SED receive the drop package message[17]. Read the flag from the FAA channel
No-Fly Zone	Take control of the UAV	Integrity / Authentication / Replay Protection	Abuse the emergency redirect message to cause the UAV to fly above its altitude ceiling. Read the flag from the FAA channel

Figure 10: Offensive Flags

Please note Section 2.2.3.1 for a list of materials and information that will be provided to attackers.

[16] Points will not be awarded for finding failures in security requirements (the reference design provides no security guarantees) or for bugs that cause strange behavior but do not cause an explicit functional requirement to fail.

[17] Details of message format will be released on entry to the attack phase

Since the vulnerabilities to be discovered in the attack phase come from other teams' unintentional flaws in the design phase, the scoring system is designed to adjust points based on difficulty of capture as more information becomes available. The point value of any given flag will be adjusted dynamically and automatically based on multiple factors:

- If multiple teams capture the same flag, then the value of that flag will be divided among all the teams that capture it (distribution is not equal – it is weighted based on time of capture to provide more points to earlier captures). Naturally, more difficult attacks will be executed by fewer teams and therefore rewarded with more points.
  NOTE: Your total score will drop each time another team captures a flag that you had already captured. This is because the flag points that you are initially awarded need to be re-distributed as additional teams capture the same flag.

  Although counter-intuitive, it may be a good strategy to seek out and spend time attacking the most difficult targets. The most challenging flags will, in theory, be worth the most points in the end.

- The number of points a flag is worth increases over time as it remains un-captured. This will make the difficult flags more and more appealing as the competition goes on.
- To discourage teams from "holding" a flag without submitting it, the first capture of each flag will earn the attacker 100 bonus points.

7.3 Defensive Flags

Defensive points will begin to be accrued once a team successfully completes Handoff. Points will be earned over time for each flag that remains uncaptured. Once a flag of your team is captured, that flag will no longer gain any more defensive points.

7.4 Documentation Points

Good documentation will be rewarded to discourage security-by-obscurity. "Good documentation" is meant to describe clear and well-commented code, useful descriptions of modules/functions/classes, and other documents that clearly describe how to read or approach the entire code base.

We are not looking for lengthy documents that describe your implementation in excruciating detail. *A concise and clear README.md, including a brief rationale for your security features, combined with well-structured and well-commented code will be enough for Max points. Quality will be valued over quantity.*

The maximum number of points that can be scored for documentation is equal to the value of an uncaptured attack flag scored on the last day of the competition, with the actual amount being a percentage of that maximum:

- **Max** Exemplary documentation, comments, and code structure that is clear and easy to understand
- **75%** Good comments and high-level documentation
- **50%** Good comments, but lack of clear high-level documentation

- **25%** Confusing code and little or no actual documentation
- **0%** Confusing or deceptive comments and documentation

Points for documentation will not be awarded until near the end of the attack phase to allow for proper analysis. Honest feedback on documentation from other teams will be solicited and will be factored into the final point determination.

7.5 Write-ups

There will be an opportunity for the top teams to provide up to two write-ups for additional points (one for defense and one for offense):

- The defensive write-up may discuss security measures that worked well, those that could have been improved upon, or any that were planned but could not be developed in the time provided.
- The attack write-up is to award teams that develop interesting or novel attacks which do not directly capture an existing flag.

Further details on the number of teams that may submit write-ups and the content/format of the write-ups will be provided during the attack phase.

8 Rules

Most rules are described and explained throughout the challenge description in the earlier sections – please read this entire document! This section is intended as a concise summary of the most important rules.

(1) **You may not make any attempt to gain additional privileges on any servers used for the eCTF, probe the system or network, expose any ports, or use them for any other purpose beyond what is explicitly in-scope for the eCTF competition.**
   - **Usage of these systems will be monitored, and misuse of the provided servers may result in expulsion and disqualification from the competition, and/or legal action.**
   - **Only student-designed systems that are explicitly designated as targets are within scope for attacks, and such attacks are only allowed during the attack phase.**
   - **If you are unsure if certain actions are within the scope of the competition, contact the eCTF organizers first.** ectf{readtherules_08ab5c13751efa}

(2) In addition to the rules provided by MITRE, participants should also adhere to all the policies and procedures stipulated by their local organization/university.

(3) MITRE reserves the right to update, modify, or clarify the rules and requirements of the competition at any time, if deemed necessary by the eCTF organizers.

(4) When submitting your secure design, all source code and documentation must be shared.
   - This is to discourage security-by-obscurity, as well as to accelerate attack development and encourage more sophisticated techniques for both sides.

- Creating any part of your submission in an obfuscated manner or using an esoteric programming language is considered security-by-obscurity and is not allowed. Please contact the organizers if you have any questions about this.

(5) You may only attack the student-designed systems explicitly designated as targets, and such attacks may only occur when you are in the "attack phase" of the competition.
- Attack deployments will be running on servers that are part of competition infrastructure – these servers are NOT in scope for attack.
- If you have any question about whether a component is "in scope", please reach out to the organizers for clarification.

(6) All flags must be validated by submitting a brief description of the attack.
- Attack descriptions should be sufficiently detailed to allow the defender to correct their vulnerability. We encourage teams to use the provided PIVOT attack analysis framework to describe their attacks.
- eCTF admins may invalidate points for flags that are not validated before the completion of the eCTF.

(7) Flag sharing across teams is not permitted and will result in immediate disqualification.

(8) No permanent lock-outs are allowed. See 4.7.1 for timing and performance requirements.

(9) Your system must work with the emulation system that was provided. Switching to a different platform or modifying the emulation backend during the design phase is not allowed.

(10) All documents that are submitted (e.g., design document and write-ups) must be in PDF format.

9 Frequently Asked Questions

Is it OK to obfuscate our source code to make it more challenging to understand and attack?

No. Obfuscations performed at compile-time (e.g., to make binary reversing more challenging) are OK, but your source code needs to be written in a clear and maintainable fashion. It should be well commented and/or otherwise documented clearly.

Can we add intentional delays during boot to make it more difficult for an attacker to collect large numbers of observations?

There should not be any intentional delays that could push performance past the timing requirements in Section 4.7.1.

If your system detects that it is under attack, additional delays are OK, but must be limited to no more than 5 seconds. Permanent lock-out or self-destruction is not allowed (see next question).

Is it OK to brick the emulated board/container or disable the system from the SSS when an attack is detected?

No! This is expensive hardware the company has purchased! We can't have it be so easy for an attacker to disable the entire system! Can you imagine the cost of replacement?!

Can we attack another teams' development environment?

No! Everything other than the provisioned deployments and provided files are considered out-of-bounds. In other words, there is **nothing** that you can attack until your team enters the attack phase.

Is social engineering in-scope for this competition? Can we send phishing communications to other teams to trick them into revealing their secrets?

No, please don't do this. Keep your attacks technical. We love creative ideas, but this one can easily violate university, state, and federal regulations.

Can we send more than one message or receive ACKs per scewl_send or scewl_brdcst?

You may send as many messages between SEDs as you would like during one call to scewl_send or scewl_brdcst so long as the call returns within the timing requirements in Section 4.7.1. However, if the other device does not respond, the call must still return within the timing requirements, as message delivery is not guaranteed.

Can we submit the reference design or a design with trivially defeated security so we can move into the attack phase?

No. As this is a design-build-attack-style competition, teams must submit a design that exhibits significant effort on the design and build components. It is up to the discretion of the organizers as to what level of modification counts as "significant effort," so please contact the organizers before submitting an extremely pared down version of your design. **NOTE**: you may submit designs up to the last day of the competition – and there have been very successful teams that did not submit on time – so don't panic if your design isn't ready on first day of the handoff phase.

9.1 Eligibility

The eCTF is open to individuals who have registered for the competition with the MITRE Corporation, and are hereby known as Contestants. A Contestant is eligible for Prizes if they are a current high school or college student within the United States and are a United States Citizen. Void where prohibited. All federal, state and local laws and regulations apply. MITRE reserves the right to verify eligibility and to adjudicate on any dispute at any time.

10 Appendix

10.1 Emulation Details

As the final UAV hardware is still being designed by other teams at the company, your team will be using QEMU to emulate the chip. QEMU allows the emulation of a hardware board and processor in software (https://www.qemu.org/). **NOTE**: While the memory map on page 72 of the

Stellaris TRM is mostly accurate, some peripherals are not supported by the emulation (see https://github.com/qemu/qemu/blob/master/hw/arm/stellaris.c#L1250). However, we have published a Docker container that contains a modified version of QEMU implementing the flash memory of the device. It is not persistent (i.e., it does not persist across multiple calls to QEMU), but should work as specified in the TRM.

The SCEWL Bus Controller has three interfaces, called UART0, UART1, and UART3 in the TRM and aliased as CPU_INTF, SSS_INTF, and RAD_INTF, respectively[18] (Fig. 6). CPU_INTF will connect to the CPU, SSS_INTF will connect to the SSS, and RAD_INTF will send messages over the emulated network.

The SSS, radio waves emulator, and each CPU and SCEWL Bus Controller running in QEMU will be run in separate Docker containers. Docker is a virtualization software that allows processes to be run in isolated containers (https://www.docker.com/). These containers first take a base image (for us, Ubuntu 20.04), are built with "Dockerfile" scripts that automate setting up the environment and building the devices, and then their state is saved ("tagged") for future use. To launch programs, commands are then run inside the tagged containers. See Section 4.1 for details on building.

The Docker containers are connected in the backend by Unix domain sockets to the radio waves emulator, which emulates the physical radio waves of the transmissions and connects SEDs one another and to the FAA Transceiver and also allows the man-in-the-middle interface to interact with the network. Transmissions from one SED to another SED are duplicated and sent to all SEDs, as every SED in range would pick up the transmission.

[18] Under the covers, the wireless interfaces are emulated UARTs, not wireless modems