Hack-A-Sat 2020 Writeups

Team BLAHAJ 🦈 https://blahaj.awoo.systems/

Contents

Attitude Adjustment

Category: Astronomy, Astrophysics, Astrometry, Astrodynamics, AAAA **Points (final)**: 69 points **Solves**: 62

> Our star tracker has collected a set of boresight reference vectors, and identified which stars in the catalog they correspond to. Compare the included catalog and the identified boresight vectors to determine what our current attitude is.
>
> Note: The catalog format is unit vector (X,Y,Z) in a celestial reference frame and the magnitude (relative brightness)

Given files: `attitude-papa21503yankee.tar.bz2`

Write-up

by erin (`barzamin`).

For this problem, we have two sets of N vectors which are paired; all points in the first set are just those in the second set up to rotation; we want to find the rotation which maps the first set onto the other one. Since we already know which point in the observation set maps to which vector in the catalog set, we can use the Kabsch algorithm to find the rotation matrix (note that this is called an *orthogonal Procrustes problem*). I'd only vaguely heard of the Kabsch algorithm before, and in the context of bioinformatics, so I didn't immediately identify it as a good path to the solution. Instead, I just googled "*align two sets of vectors*", for which it's the third result.

Since nobody has time to implement computational geometry during a ctf, I grabbed an existing Kabsch implementation. For some reason, I didn't notice that `scipy.spatial` has a Kabsch implementation built in, so I used some random external project, `rmsd`.

First, load the star catalog:

```
catalog = {}
with open('./attitude-papa21503yankee/test.txt') as f:
    i = 0
    for line in f:
        [x, y, z, m] = [float(s.strip()) for s in line.split(',')]
        catalog[i] = {'v': np.array([x,y,z]), 'm':m}
        i += 1
```

Set up some helpers for parsing the output of the challenge server and solving an orientation:

```
def parse_stars(stardata):
    stars = {}
    for line in stardata.strip().split('\n'):
        line = line.strip()
        star_id = int(line.split(':')[0].strip())
        direction = np.array([float(x) for x in line.split(':')[1].split(',\t')])
        stars[star_id] = direction
    return stars

def solve_orientation(stars, catalog):
    P = np.vstack(list(stars.values()))
    Q = np.vstack([catalog[i]['v'] for i in stars.keys()])
    print("rmsd: {}".format(calculate_rmsd.kabsch_rmsd(P,Q)))
    rotation_mtx = calculate_rmsd.kabsch(P, Q)
    rotation = Rotation.from_matrix(np.linalg.inv(rotation_mtx))
    return rotation
```

Note that I threw in an inversion of the rotation matrix; this is because I should've been aligning from the catalog *to* the current star locations. Switching P to be the catalog and Q to be stars would've done the same thing.

Then we just grabbed each challenge from the computer, aligned the sets, and spat the orientation of the satellite back at the server:

```
TICKET = 'THE_TICKET'
r = tubes.remote.remote('attitude.satellitesabove.me', 5012)
r.send(TICKET+'\n')
time.sleep(0.5)
for _ in range(20):
    r.recvuntil(b'--------------------------------------------------\n', drop=True)
    stars = parse_stars(r.recv().decode())
    rotation = solve_orientation(stars, catalog)
    r.send(','.join([str(x) for x in rotation.as_quat()]) + '\n')
    time.sleep(0.1)
print(r.clean())
```

The flag should get printed out on stdout by the final line.

Full code

```
import numpy as np
from pwnlib import tubes
import time
import matplotlib.pyplot as plt
from rmsd import calculate_rmsd
from scipy.spatial.transform import Rotation

catalog = {}
with open('./attitude-papa21503yankee/test.txt') as f:
    i = 0
    for line in f:
        [x, y, z, m] = [float(s.strip()) for s in line.split(',')]
        catalog[i] = {'v': np.array([x,y,z]), 'm':m}
        i += 1

def parse_stars(stardata):
    stars = {}
    for line in stardata.strip().split('\n'):
        line = line.strip()
        star_id = int(line.split(':')[0].strip())
        direction = np.array([float(x) for x in line.split(':')[1].split(',\t')])
        stars[star_id] = direction
    return stars

def solve_orientation(stars, catalog):
    P = np.vstack(list(stars.values()))
    Q = np.vstack([catalog[i]['v'] for i in stars.keys()])
    print("rmsd: {}".format(calculate_rmsd.kabsch_rmsd(P,Q)))
    rotation_mtx = calculate_rmsd.kabsch(P, Q)
    rotation = Rotation.from_matrix(np.linalg.inv(rotation_mtx))
    return rotation

TICKET = 'THE_TICKET'
r = tubes.remote.remote('attitude.satellitesabove.me', 5012)
r.send(TICKET+'\n')
time.sleep(0.5)
for _ in range(20):
    r.recvuntil(b'--------------------------------------------------\n', drop=True)
    stars = parse_stars(r.recv().decode())
    rotation = solve_orientation(stars, catalog)
    r.send(','.join([str(x) for x in rotation.as_quat()]) + '\n')
    time.sleep(0.1)
print(r.clean())
```

Resources and other writeups

- https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem
- https://en.wikipedia.org/wiki/Kabsch_algorithm
- https://github.com/charnley/rmsd/tree/master

Digital Filters, Meh

Category: Astronomy, Astrophysics, Astrometry, Astrodynamics, AAAA **Points (final)**: 104 points **Solves**: 37

> Included is the simulation code for the attitude control loop for a satellite in orbit. A code reviewer said I made a pretty big mistake that could allow a star tracker to misbehave. Although my code is flawless, I put in some checks to make sure the star tracker can't misbehave anyways.
>
> Review the simulation I have running to see if a startracker can still mess with my filter. Oh, and I'll be giving you the attitude of the physical system as a quaternion, it would be too much work to figure out where a star tracker is oriented from star coordinates, right?

Given files: `src.tar.gz`.

Write-up

by erin (`barzamin`).

As part of this challenge, we're given a file `src.tar.gz` by the scoreboard. This contains Octave code which simulates the satellite kinetics and control loop, and, presumably, is what's running on the challenge server.

Digging into the main file, `challenge.m`, we note a few interesting things. The satellite is running a Kalman filter on the gyroscope data (velocity and acceleration) and star tracker orientation data. Near the start of each control system iteration, there's a check on the Kalman filter error; if greater than a threshold, the controller crashes out:

```
% Check error bounds
if max(abs(err)) > err_thresh
    disp("Error: Estimator error too large... Goodbye");
    disp(q_est);
    disp(target.q_att);
    break;
endif
```

Note `err_thresh` is defined upscript as

```
err_thresh = 1*pi/180. ;   % 1 Degree error max
```

Further down, we see that a PID update step on for the correction is only run every five iterations; this is weird, but doesn't help us as far as we could tell. Even further downscript, we see that every iteration, a check is performed between target and actual orientations (note that *actual orientation* means, here, the true simulated physical pose of the satellite):

```
% Check we're still safe...
[v,a] = q2rot(quat_diff(actual.q_att, target.q_att));
if abs(v(2)*a) >  (pi/8)
    disp("Uh oh, better provide some information!");
    disp(getenv("FLAG"))
    break;
endif
```

If this check is true (ie, there's >π/8 radians of rotation error on the Y axis), we get the flag! So the challenge here is making the satellite think it's drifted when it hasn't, without making the Kalman filter angry. How can we do that?

The star filter observations are pulled right after the previous check, with the following code:

```
% Get Observations
q_att = startracker(actual);
```

Looking inside `startracker()`, we see that it pretty clearly indicates that we *are* the star tracker; every timestep, the code tells us, the adversary, what the true physical orientation is as a quaternion. We can act as the star tracker and send back a wxyz-format quaternion on stdin, which it will use as the star-tracker output (note that, for some reason, the code they give us uses space-separated floats and the actual challenge uses comma-separated floats):

```
% Model must have a q_att member

function [ q ] = startracker(model)
  q = model.q_att;
  disp([q.w, q.x, q.y, q.z]);
  fflush(stdout);
  % Get Input
  q = zeros(4,1);
  for i = 1:4
    q(i) = scanf("%f", "C");
  endfor
  q = quaternion(q(1), q(2), q(3), q(4));
  %q.w = q.w + normrnd(0, 1e-8);
  %q.x = q.x + normrnd(0, 1e-8);
  %q.y = q.y + normrnd(0, 1e-8);
  %q.z = q.z + normrnd(0, 1e-8);

  q = q./norm(q);

endfunction
```

Also note that in `challenge.m`, immediately after the star tracker call, checking the return value for consistency with the physical model is *commented out*. We can tell the satellite that it's pointing anywhere we like and it will believe us, although Kalman error might be bad:

```
%err = quat2eul(quat_diff(q_att, target.q_att))';
%if max(abs(err)) > err_thresh
%    disp("Error: No way, you are clearly lost, Star Tracker!");
%    break;
%endif
```

So we control the star tracker. What can we do?

We immediately noticed the vast count of `eul2quat()` and `quat2eul()` calls and wasted so much time trying to get something to gimbal lock. Turns out this problem is deceptively easy, and you don't need to do that at all.

We can't make the discrepancy between the true position and what the star tracker says too great, nor make it vary quickly; the gyroscope is reporting gaussian noise close to zero with very low variance, so any big delta in star tracker orientation will incur error in the Kalman filter. So what can we do?

Turns out all we have to do hold the Y orientation constant and report that. The satellite's true Y euler angle gradually rotates over time due to system dynamics, accumulating controller error on the Y axis, and we eventually get the flag.

Hook up to the server:

```
sep = ','
r = remote('filter.satellitesabove.me', 5014)
r.clean()
r.send('THE_TICKET')
time.sleep(0.1)
```

Write a little function that pretends to be the star tracker (note: `lie` was determined by playing with the local simulator a bunch):

```
def adversary(true_pose):
    lie = 0.25

    euler = true_pose.as_euler('xyz')
    euler[1] = lie
    return R.from_euler('xyz',euler)
```

And talk to the server, pretending to be the tracker every indication, until we see a string indicating we got the flag:

```
while True:
    rl = r.readline(timeout=3)
    if rl.startswith(b'Uh oh,'):
        r.interactive()
    log.info(f'<== [{i}] {rl}')

    [w,x,y,z] = [float(x) for x in rl.decode().strip().split()]
    true_pose = R.from_quat([x,y,z,w])

    new_pose = adversary(true_pose)

    [x,y,z,w] = new_pose.as_quat()
    msg = sep.join(map(str,[w,x,y,z]))
    log.info(f'==> {msg}')
    r.send(msg+'\n')
```

When we see that string, the script jumps to `pwnlib.tubes`' interactive mode and we see the flag in the dumped buffer.

Full code

```
import numpy as np
import matplotlib.pyplot as plt
from pwn import *
from scipy.spatial.transform import Rotation as R
import time

q_att_ts = []
badnesses = []

LOCAL = False

if LOCAL:
    sep = ' '
    r = process('octave challenge.m', shell=True)
else:
    sep = ','
    r = remote('filter.satellitesabove.me', 5014)
    r.clean()
    r.send('THE_TICKET')
    time.sleep(0.1)

def adversary(true_pose):
    lie = 0.25

    euler = true_pose.as_euler('xyz')
    euler[1] = lie

    return R.from_euler('xyz',euler)


for i in range(10000):
    if LOCAL:
```

```
    badness = float(r.readline().decode().strip())
    log.info(f'[!] badness: {badness}')
    badnesses.append(badness)

rl = r.readline(timeout=3)
if rl.startswith(b'Uh oh,'):
    r.interactive()
log.info(f'<== [{i}] {rl}')

[w,x,y,z] = [float(x) for x in rl.decode().strip().split()]
true_pose = R.from_quat([x,y,z,w])

new_pose = adversary(true_pose)

[x,y,z,w] = new_pose.as_quat()
msg = sep.join(map(str,[w,x,y,z]))
log.info(f'==> {msg}')
r.send(msg+'\n')
```

I Like To Watch

Category: Astronomy, Astrophysics, Astrometry, Astrodynamics, AAAA **Points (final):** 37 **Solves:** 126

> Fire up your Google Earth Pro and brush up on your KML tutorials, we're going to make it look at things!

Write-up

by haskal

A netcat endpoint is provided, and when you connect it provides the following info:

```
We've captured data from a satellite that shows a flag located at the base of the Washington
Monument.
The image was taken on March 26th, 2020, at 21:54:33
The satellite we used was:

REDACT
1 13337U 98067A   20087.38052801 -.00000452  00000-0  00000+0 0  9995
2 13337  51.6460  33.2488 0005270  61.9928  83.3154 15.48919755219337

Use a Google Earth Pro KML file to 'Link' to http://18.191.77.141:26963/cgi-bin/HSCKML.py
and 'LookAt' that spot from where the satellite when it took the photo and get us that flag!
```

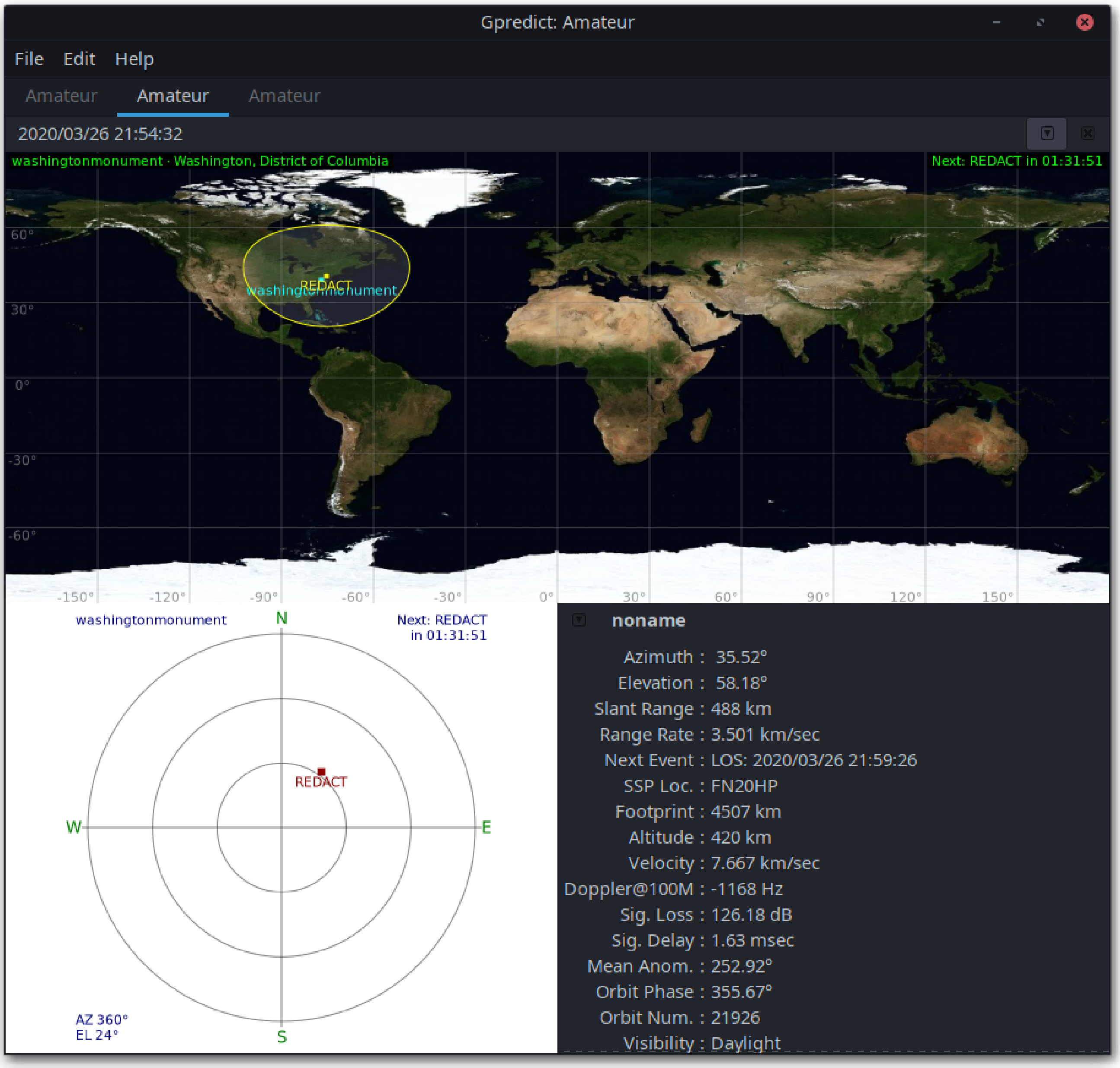
Additionally, an example KML file is provided

```xml
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Folder>
    <name>HackASatCompetition</name>
    <visibility>0</visibility>
    <open>0</open>
    <description>HackASatComp1</description>
    <NetworkLink>
      <name>View Centered Placemark</name>
      <visibility>0</visibility>
      <open>0</open>
      <description>This is where the satellite was located when we saw it.</description>
      <refreshVisibility>0</refreshVisibility>
      <flyToView>0</flyToView>
      <LookAt id="ID">
        <!-- specific to LookAt -->
        <longitude>FILL ME IN</longitude>                <!-- kml:angle180 -->
        <latitude>FILL ME IN TOO</latitude>                <!-- kml:angle90 -->
        <altitude>FILL ME IN AS WELL</altitude>                <!-- double -->
        <heading>FILL IN THIS VALUE</heading>                <!-- kml:angle360 -->
        <tilt>FILL IN THIS VALUE TOO</tilt>                <!-- kml:anglepos90 -->
        <range>FILL IN THIS VALUE ALSO</range>                <!-- double -->
        <altitudeMode>clampToGround</altitudeMode>
      </LookAt>
      <Link>
        <href>http://FILL ME IN:FILL ME IN/cgi-bin/HSCKML.py</href>
        <refreshInterval>1</refreshInterval>
        <viewRefreshMode>onStop</viewRefreshMode>
        <viewRefreshTime>1</viewRefreshTime>
        <viewFormat>BBOX=[bboxWest],[bboxSouth],[bboxEast],[bboxNorth];
CAMERA=[lookatLon],[lookatLat],[lookatRange],[lookatTilt],[lookatHeading];
VIEW=[horizFov],[vertFov],[horizPixels],[vertPixels],[terrainEnabled]</viewFormat>
      </Link>
    </NetworkLink>
  </Folder>
</kml>
```

We can use gpredict to figure out where the satellite was by loading the TLE (one way is to create an http endpoint with the TLE in a txt file, and then add the URL in the gpredict settings). However, gpredict will refuse to load this TLE. It turns out the checksums are incorrect, and if we calculate them according to the TLE spec, we get these lines with fixed checksums

```
REDACT
1 13337U 98067A   20087.38052801 -.00000452  00000-0  00000+0 0  9992
2 13337  51.6460  33.2488 0005270  61.9928  83.3154 15.48919755219334
```

Now, gpredict loads the data (if not, close gpredict, clear the cache with `rm ~/.config/Gpredict/satdata/*.sat`, start gpredict, and select `Update TLE data from network`). The next step is to create a location for the washington monument. The monument is located at -77.0354,38.889100. Finally, use the gpredict time controller to pause real time, then set the time to March 26th, 2020 at 21:54:33.



noname

```
      Azimuth : 35.52°
    Elevation : 58.18°
  Slant Range : 488 km
   Range Rate : 3.501 km/sec
   Next Event : LOS: 2020/03/26 21:59:26
     SSP Loc. : FN20HP
    Footprint : 4507 km
     Altitude : 420 km
     Velocity : 7.667 km/sec
 Doppler@100M : -1168 Hz
    Sig. Loss : 126.18 dB
   Sig. Delay : 1.63 msec
  Mean Anom. : 252.92°
  Orbit Phase : 355.67°
   Orbit Num. : 21926
   Visibility : Daylight
```

We can see that (in the ground reference frame) the satellite is at azimuth 35.52 degrees and elevation 58.18 degrees. Additionally, it has a line-of-sight range of 488 km.

To make our lives easier you can notice in Wireshark that Google Earth Pro simply makes HTTP requests to the given endpoint with parameters given in the KML file, like this

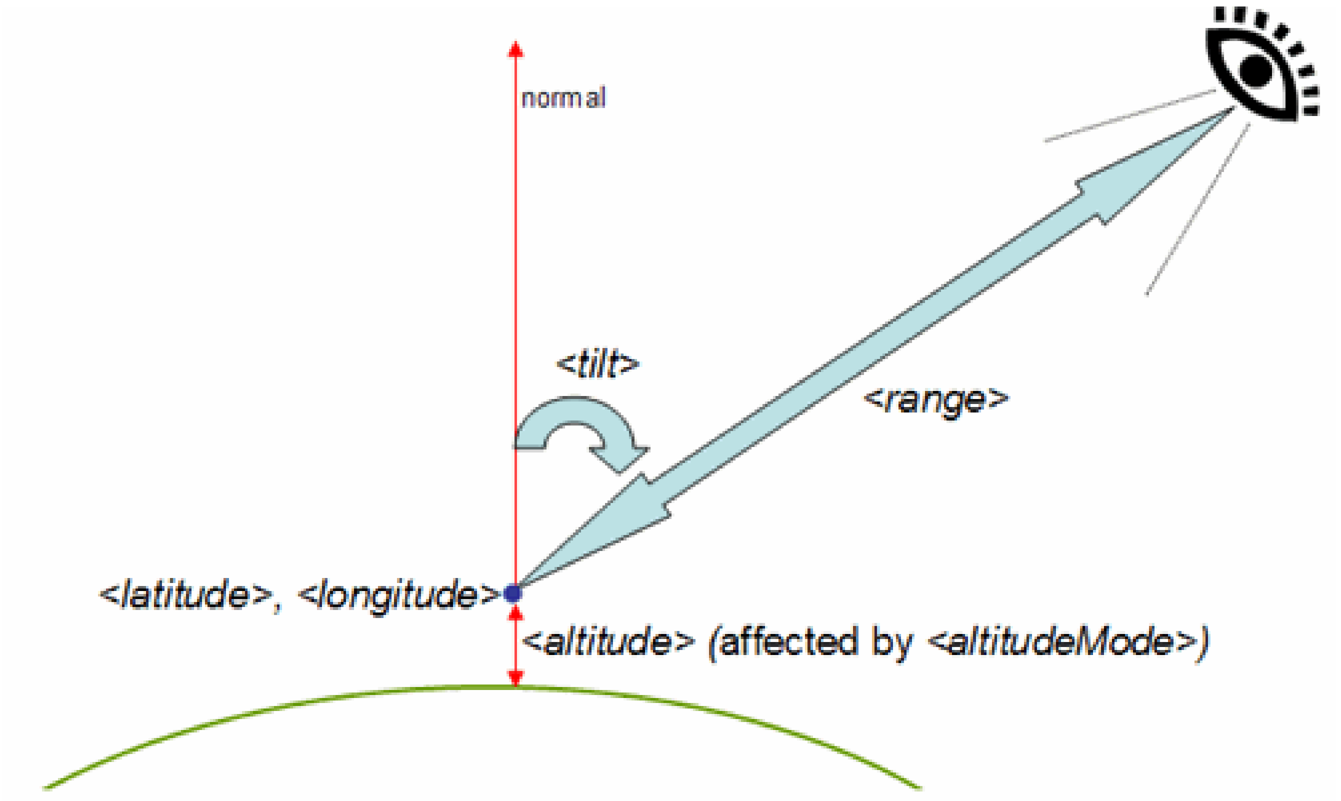
```
http://server/cgi-bin/HSCKML.py?BBOX=...;CAMERA=...;...
```

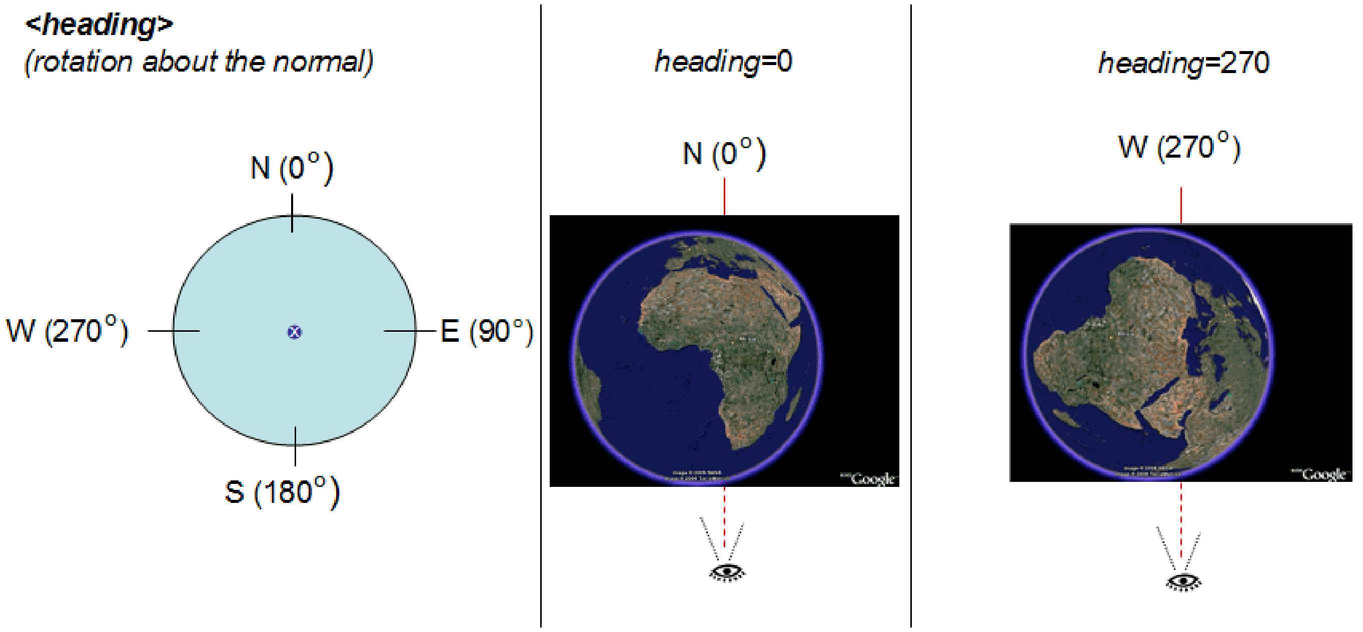
So we can use plain curl to avoid messing with the Google Earth Pro GUI a lot. We need the following parameters: the bounding box of the view, the camera parameters, and the view parameters.

For the bounding box, we create a reasonable box around the location of the washington monument

```
BBOX=-77.035378,38.889384,-77.035178,38.889584
```

For the camera parameters, we look directly at the base, but we need to provide a heading and tilt. The Google Earth KML reference has a handy diagram of the reference frame needed





Since gpredict is in a ground reference frame, we need to add 180 to the azimuth to get the heading, and subtract 90 - elevation to get the tilt. With these calculations and the monument coordinates we have (note the range is in meters, not km, so we multiply by 1000)

```
CAMERA=-77.035278,38.889484,488000,31.82,215.18
```

Finally, for the view we chose some reasonable parameters that seemed to work. This part doesn't seem to be very important

```
VIEW=60,60,500,500,1
```

Putting it together, the full URL is

```
http://theserver/cgi-bin/HSCKML.py?BBOX=-77.035378,38.889384,-77.035178,38.889584;
CAMERA=-77.035278,38.889484,488000,31.82,215.18;VIEW=60,60,500,500,1
```

Requesting the URL reveals the flag

```xml
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Placemark>
<name>CLICK FOR FLAG</name>
<description>flag{juliet71739hotel:GNeeb.....}</description>
<Point>
<coordinates>-77.0354,38.889100</coordinates>
</Point>
</Placemark>
</kml>
```

Resources and other writeups

- http://gpredict.oz9aec.net/
- https://en.wikipedia.org/wiki/Two-line_element_set
- https://developers.google.com/kml/documentation/kmlreference?csw=1#range

My 0x20

Category: Astronomy, Astrophysics, Astrometry, Astrodynamics, AAAA **Points (final)**: 142 points **Solves**: 24

> This social media app is back with a vengeance!

Given files: `myspace-mike33686zulu.tar.bz2`

Write-up

by erin (`barzamin`).

0x20 is SpaceBook, except without a backdoor. If you haven't read that writeup yet, go take a look at it.

That is, the trick in SpaceBook – magnitudes are trivially matchable – is gone. Additionally, we have less superbright outliers in the provided unknown star set. We can no longer match specific stars trivially. However, we can still solve the problem: this is what *real* star trackers have to do (although they have to deal with stuff like measurement and catalog error that don't show up here).

Since the observed (unknown) stars are all one simple rigid transformation (the observation orientation!) away from their positions in the catalog, relative distances and angles are preserved. Assuming that the local neighborhood of each star is represented relatively well in the observation set, we can look at the neighborhood of each star – hopefully – to determine what star it is.

Vaguely inspired by *Pattern Recognition of Star Constellations for Spacecraft Applications*, C. C. Liebe 1993 (thanks, google), I designed a simple star fingerprint: every star is represented by a 3-vector containing:

- the distance to the closest neighbor
- the distance to the second-closest neighbor
- the angle between the two closest neighbors, relative to the star under consideration.

To make finding nearest neighbors fast, I used scikit-learn's ball tree implementation. I could then easily generate fingerprints for any given set of stars and its ball tree (warning: CTF-quality ML code from here on, sorry):

```python
def angle(x, y):
    x_ = x/np.linalg.norm(x)
    y_ = y/np.linalg.norm(y)
    return np.arccos(np.clip(np.dot(x_, y_), -1.0, 1.0))

def gen_fingerprints(X, bt):
    dist, ind = bt.query(X[:], k=3)
    fingerprints = []
    for i in range(X.shape[0]):
        a = X[i,:]
        [bi, ci] = ind[i,1:]
        b = X[bi,:]
        c = X[ci,:]
        fingerprints.append([
            *dist[i, 1:],
            angle(b-a, c-a),
        ])
    return fingerprints
```

We could then fingerprint the star catalog:

```python
with open('./myspace-mike33686zulu/test.txt') as f:
    catalog = read_starfile(f.read())

X = np.vstack([s['v'] for s in catalog])
bt_catalog = BallTree(X, leaf_size=30)

fingerprints = gen_fingerprints(X, bt_catalog)
```

You could do something probably more noise resistant, like finding N best-match fingerprints, doing Kabsch on those pairs, and then finding the rest by L2 norm similarity between boresight vectors after rotating to the catalog orientation. I just spat out the index of the closest catalog fingerprint match for each unknown star, throwing out fingerprint matches with error that was too high (L2 distance between fingerprints > 1×10^{-4}). Do that for every challenge, and you're done:

```python
for _ in range(5):
    refstars = read_starfile(r.recvuntil('\n\n').decode())

    Y = np.vstack([s['v'] for s in refstars])
    bt_unknown = BallTree(Y, leaf_size=30)

    fingerprints_unknown = gen_fingerprints(Y, bt_unknown)

    matches = []
    for query in fingerprints_unknown:
        error = [np.linalg.norm(np.array(fp_known) - np.array(query))
                 for fp_known in fingerprints]
        match_idx = np.argmin(error)
        if error[match_idx] < 1e-4:
            print(match_idx, error[match_idx])
            matches.append(match_idx)

    r.send(','.join(map(str,matches)) + '\n')
    r.recvuntil('Left...\n')
print(r.clean())
```

Fun fact! We got second or third on this (I don't remember), even though I had the code done in *literally this state* before anyone else submitted a solution. This was entirely my fault, since I was *accidentally using the SpaceBook star catalog*.

Full code

```python
import numpy as np
from pwnlib import tubes
import time
import matplotlib.pyplot as plt
from rmsd import calculate_rmsd
from scipy.spatial.transform import Rotation
import seaborn as sb
import itertools
from sklearn.neighbors import BallTree

def read_starfile(data):
    stars = []
    for line in data.strip().split('\n'):
        [x,y,z,m] = [float(s.strip()) for s in line.split(',')]
        stars.append({'v': np.array([x,y,z]), 'm':m})
    return stars

with open('./myspace-mike33686zulu/test.txt') as f:
    catalog = read_starfile(f.read())

def angle(x, y):
    x_ = x/np.linalg.norm(x)
    y_ = y/np.linalg.norm(y)
    return np.arccos(np.clip(np.dot(x_, y_), -1.0, 1.0))
```

```python
def gen_fingerprints(X, bt):
    dist, ind = bt.query(X[:], k=3)
    fingerprints = []
    for i in range(X.shape[0]):
        a = X[i,:]
        [bi, ci] = ind[i,1:]
        b = X[bi,:]
        c = X[ci,:]
        fingerprints.append([
            *dist[i, 1:],
            angle(b-a, c-a),
        ])
    return fingerprints

TICKET = 'THE_TICKET'
r = tubes.remote.remote('myspace.satellitesabove.me', 5016)
r.send(TICKET+'\n')
time.sleep(0.5)
r.recvuntil('Ticket please:\n', drop=True)

X = np.vstack([s['v'] for s in catalog])
bt_catalog = BallTree(X, leaf_size=30)

fingerprints = gen_fingerprints(X, bt_catalog)

for _ in range(5):
    refstars = read_starfile(r.recvuntil('\n\n').decode())

    Y = np.vstack([s['v'] for s in refstars])
    bt_unknown = BallTree(Y, leaf_size=30)

    fingerprints_unknown = gen_fingerprints(Y, bt_unknown)

    matches = []
    for query in fingerprints_unknown:
        error = [np.linalg.norm(np.array(fp_known) - np.array(query))
                 for fp_known in fingerprints]
        match_idx = np.argmin(error)
        if error[match_idx] < 1e-4:
            print(match_idx, error[match_idx])
            matches.append(match_idx)

    r.send(','.join(map(str,matches)) + '\n')
    r.recvuntil('Left...\n')
print(r.clean())
```

Resources and other writeups

- https://ieeexplore.ieee.org/document/180383
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html

Seeing Stars

Category: Astronomy, Astrophysics, Astrometry, Astrodynamics, AAAA **Points (final)**: 23 **Solves**: 213

> Here is the output from a CCD Camera from a star tracker, identify as many stars as you can! (in image reference coordinates) Note: The camera prints pixels in the following order (x,y): (0,0), (1,0), (2,0)... (0,1), (1,1), (2,1)...
>
> Note that top left corner is (0,0)

Write-up

by hazel (arcetera)

The CCD image given by the netcat is a 128x128 matrix of comma-separated values.

We read the data into a NumPy array, and pass that into OpenCV.

```
data = []
for line in rawdat.strip().split('\n'):
    data.append([int(x) for x in line.split(',')])

x = np.array(data, dtype='uint8').T

im = x
```

We then run a filter on the data, only grabbing values in [127, 255] to filter out data that is *obviously not* stars. We then run two dilates on the image post-filter, because otherwise we end up with a division by zero on centroid finding later for `M["m00"]`. Finally, we grabbed the contour of every object visible in the image.

```
ret, thresh = cv2.threshold(im.copy(), 127, 255, 0)
kernel = np.ones((5, 5), np.uint8)
dilated = cv2.dilate(thresh.copy(), kernel, iterations = 2)

cnts, hier = cv2.findContours(dilated.copy(), \
                              cv2.RETR_TREE, \
                              cv2.CHAIN_APPROX_NONE)
```

For each contour, we grabbed its centroid:

```
solve = ''
for c in cnts:
    M = cv2.moments(c)
    cX = int(M["m10"] / M["m00"])
    cY = int(M["m01"] / M["m00"])

    solve += (str(cX) + "," + str(cY)+'\n')
return solve
```

We then automated this entire process using pwnlib to connect to the server and read the data.

Full code

```
#!/usr/bin/env python3
import cv2
import math
import numpy as np
from pwnlib import tubes
import time

def solve(rawdat):
    data = []
    for line in rawdat.strip().split('\n'):
        data.append([int(x) for x in line.split(',')])

    x = np.array(data, dtype='uint8').T

    im = x # cv2.imread("output.png", cv2.IMREAD_GRAYSCALE)
    ret, thresh = cv2.threshold(im.copy(), 127, 255, 0)
    kernel = np.ones((5, 5), np.uint8)
    dilated = cv2.dilate(thresh.copy(), kernel, iterations = 2)

    cnts, hier = cv2.findContours(dilated.copy(), \
                                  cv2.RETR_TREE, \
                                  cv2.CHAIN_APPROX_NONE)

    edit = thresh.copy()
    cv2.drawContours(edit, cnts, -1, (0, 255, 0), 3)

    solve = ''
    for c in cnts:
        M = cv2.moments(c)
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])

        solve += (str(cX) + "," + str(cY)+'\n')
    return solve

TICKET = 'THE_TICKET'
r = tubes.remote.remote('stars.satellitesabove.me', 5013)
r.recvline()
r.send(TICKET+'\n')
going = True
while going:
    rawdat = r.recvuntil('Enter', drop=True)
    time.sleep(0.5)
    r.clean()
    solution = solve(rawdat.decode())
    r.send(solution+'\n')
    time.sleep(0.1)
    if r.recvuntil('Left...\n') == b'0 Left...\n':
        time.sleep(0.1)
        print(r.clean())
```

Run it, and the flag should be printed as a bytestring.

Resources and other writeups

- https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html
- https://docs.opencv.org/trunk/dd/d49/tutorial_py_contour_features.html

SpaceBook

Category: Astronomy, Astrophysics, Astrometry, Astrodynamics, AAAA **Points (final)**: 75 points **Solves**: 56

> Hah, yeah we're going to do the hard part anyways! Glue all previous parts together by identifying these stars based on the provided catalog. Match the provided boresight refence vectors to the catalog refence vectors and tell us our attitude.
>
> Note: The catalog format is unit vector (X,Y,Z) in a celestial reference frame and the magnitude (relative brightness)

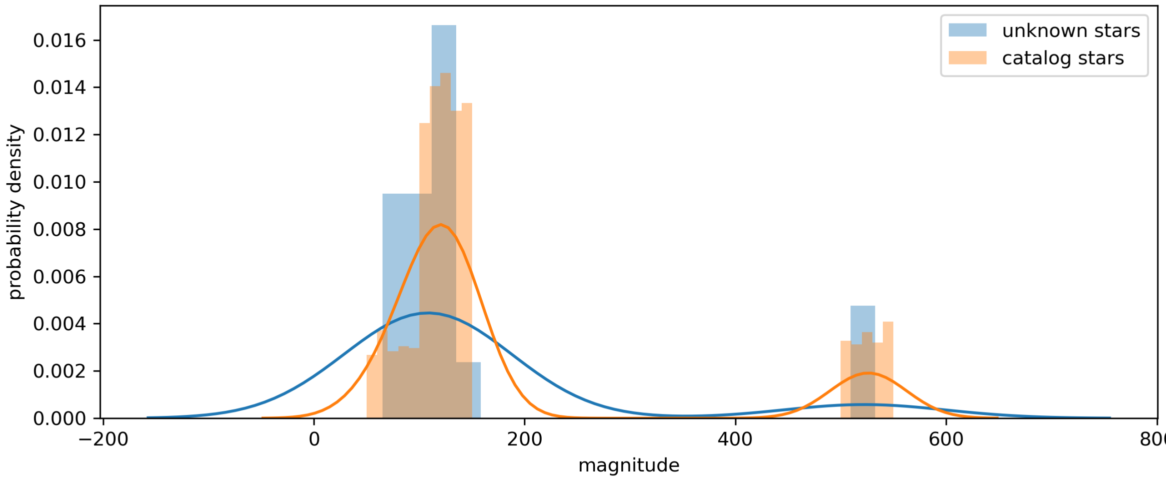
Given files: `spacebook-golf56788echo.tar.bz2`

Write-up

by erin (`barzamin`).

SpaceBook is a very similar problem to Attitude Adjustment; we still need to solve an orthogonal Procrustes problem to align a set of stars to a catalog. However, we don't know which stars in the unknown set match to which stars in the catalog, unlike in Attitude Adjustment.

We're given a boresight vector $\vec{v} \in \mathbb{R}^3$ for each star, and a magnitude m. The vectors in the catalog set are off by some rotation from the vectors in the unknown set; the magnitudes match. Looking at the distribution of magnitudes, we see that there is a significant outlier population in both the catalog and reference sets, and that these outlier populations match:



My immediate suspicion was that we could take the outliers (simply by thresholding for stars with $m > 500$) and, for each, directly match it to the catalog by finding the star with closest magnitude in the catalog:

```python
refstars_magsorted = sorted(refstars, key=lambda x:x['m'])[::-1]

catalog_magnitudes = np.array([x['m'] for x in catalog])
matches = []

for idx, star in enumerate(refstars_magsorted):
    if star['m'] <= 500:
        break
    match = np.argmin(np.abs(catalog_magnitudes-star['m']))
    matches.append(match)
```

We can then align those brightest stars to their closest magnitude matches using the Kabsch algorithm:

```python
P = np.vstack([x['v'] for x in [catalog[i] for i in matches]])
Q = np.vstack([x['v'] for x in refstars_magsorted[:4]])
```

```
print("rmsd: {}".format(calculate_rmsd.kabsch_rmsd(P,Q)))
rotation_mtx = calculate_rmsd.kabsch(P, Q)
rotation = Rotation.from_matrix(rotation_mtx)
```

Knowing the rotation, we can rotate all stars into the catalog reference frame and then evaluate

$$\arg\min_{\vec{u}\in\text{catalog}} \|\vec{u}-\vec{v}\|_2$$

for each unknown star with vector $\vec{v}$ to find the closest star in the dictionary:

```
rotated = [dict(v=rotation.apply(x['v']), m=x['m']) for x in refstars_magsorted]
found_idxes = []
for star in rotated:
    found_idxes.append(np.argmin([np.linalg.norm(star['v']-catalogstar['v']) for catalogstar in catalog]
```

It's then trivial to send `found_idxs` back to the challenge as a comma-separated string of indices for each problem sent to us; the server happily sends us the flag after we answer its questions.

Full code

```
import numpy as np
from pwnlib import tubes
import time
import matplotlib.pyplot as plt
from rmsd import calculate_rmsd
from scipy.spatial.transform import Rotation
%matplotlib inline

def read_starfile(data):
    stars = []
    for line in data.strip().split('\n'):
        [x,y,z,m] = [float(s.strip()) for s in line.split(',')]
        stars.append({'v': np.array([x,y,z]), 'm':m})
    return stars

with open('./spacebook-golf56788echo/test.txt') as f:
    catalog = read_starfile(f.read())

TICKET = 'THE_TICKET'
r = tubes.remote.remote('spacebook.satellitesabove.me', 5015)
r.send(TICKET+'\n')
time.sleep(0.5)
r.recvuntil('Ticket please:\n', drop=True)

for _ in range(5):
    refstars = read_starfile(r.recvuntil('\n\n').decode())

    refstars_magsorted = sorted(refstars, key=lambda x:x['m'])[::-1]

    catalog_magnitudes = np.array([x['m'] for x in catalog])
    matches = []

    for idx, star in enumerate(refstars_magsorted):
        if star['m'] <= 500:
            break
        match = np.argmin(np.abs(catalog_magnitudes-star['m']))
        matches.append(match)
    print(matches)
    P = np.vstack([x['v'] for x in [catalog[i] for i in matches]])
    Q = np.vstack([x['v'] for x in refstars_magsorted[:4]])
    print("rmsd: {}".format(calculate_rmsd.kabsch_rmsd(P,Q)))
    rotation_mtx = calculate_rmsd.kabsch(P, Q)
    rotation = Rotation.from_matrix(rotation_mtx)

    rotated = [dict(v=rotation.apply(x['v']), m=x['m']) for x in refstars_magsorted]
    found_idxes = []
    for star in rotated:
        found_idxes.append(np.argmin([np.linalg.norm(star['v']-catalogstar['v']) for catalogstar in catal

    r.send(','.join([str(x) for x in found_idxes]) + '\n')
    time.sleep(0.1)
    r.recvuntil('Left...\n')
print(r.clean())
```

Resources

- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.align_vectors.html

56K Flex Magic

Category: Communication Systems **Points (final):** 205 **Solves:** 13

> Anyone out there speak modem anymore? We were able to listen in to on, maybe you can ask it for a flag…
>
> UPDATE: Since everyone is asking, yes…a BUSY signal when dialing the ground station is expected behavior.

Write-up

by haskal

An audio file is included that contains a session where a number is dialed, and then some modem data is exchanged (it's a very distinctive sound). Additionally, there is a note included with the following text.

```
---=== MY SERVER ===---
Phone #: 275-555-0143
Username: hax
Password: hunter2

* Modem implements a (very small) subset of 'Basic' commands as
  described in the ITU-T V.250 spec (Table I.2)

---=== THEIR SERVER ===---

Ground station IP: 93.184.216.34
Ground station Phone #: 458-XXX-XXXX ...?
Username: ?
Password: ?

* They use the same model of modem as mine... could use +++ATH0
  ping of death
* It'll have an interactive login similar to my server
* Their official password policy: minimum requirements of
  FIPS112 (probably just numeric)
    * TL;DR - section 4.1.1 of 'NBS Special Publication 500-137'
```

ITU-T V.250 is essentially a formalization of the Hayes command set, so we can use basic Hayes commands to interact with our local modem, such as

```
ATDTXXXXXXXXXX - dial number XXX...
ATH0           - hang up
+++            - get the local modem's attention while in a remote session
```

The first step is to try to get information about the ground station server. We can decode the dial tones from the audio file, which are DTMF tones. Once decoded we obtain a phone number for the ground station of `458-555-0142`. However dialing this number results in an error - `BUSY`. Presumably, the ground station is already dialed into somewhere, and we need to disconnect it.

The "ping of death" refers to injecting a modem command into a packet sent to a remote server in order to cause the server's modem to interpret a hang up command contained in the packet. This can be achieved by pinging with the data `+++ATH0`, because as the server replies to the ping with the same data, its local modem will interpret the command inside the ping. But we need to escape it with hex to avoid having our local modem hang up instead. Once in the session, we dial the number in the text file to get an initial shell session

```
ATDT2755550143
```

Next, issue a ping of death to the provided server IP

```
ping -v 0x2b2b2b415448290d 93.184.216.34
```

Now the ground station should be disconnected so it is available for us to dial.

```
+++ATH0
ATDT45845550142
```

We get a login prompt for SATNET

```
*   *   . *  *    * .   *   * . *   *  .
    .  *   *   .    * .    . *      .  *
 *   +------------------------------+
   . |            SATNET            |   *
     +------------------------------+ .
  .  |    UNAUTHORIZED ACCESS IS    |
     |     STRICTLY PROHIBITED      |
.    +------------------------------+   .
         .             .
                                  .

Setting up - this will take a while...

LOGIN
Username:
```

However we still need the username and password. Maybe the provided audio file has the credentials somewhere in the dialup modem exchange. By analyzing the spectrum in Audacity (or any analyzer of choice) we discover that it has peaks around 980 Hz, 1180 Hz, 1650 Hz, and 1850 Hz. This is consistent with the ITU V.21 standard which uses dual-channel Frequency Shift Keying at 300 bits/second. We can use minimodem to decode the modem traffic. We can provide the two FSK frequencies (the “mark” and “space”, representing each bit of the data) for channel 1 and then for channel 2 to get both sides of the exchange. We also need to provide the bit rate.

```
minimodem -8 -S 980 -M 1180 -f recording.wav 300
minimodem -8 -S 1650 -M 1850 -f recording.wav 300
```

This data looks like garbage but it contains some strings, notably `rocketman2674`. We assume from the notes file that the password is a 4-digit number, but trying the username `rocketman` and password `2674` didn’t work. We need to look closer. This is the beginning of one side of the exchange in hex:

```
00000000: 7eff 7d23 c021 7d21 7d20 7d20 7d34 7d22  ~.}#.!}!} } }4}"
00000010: 7d26 7d20 7d20 7d20 7d20 7d25 7d26 28e5  }&} } } } }%}&(.
00000020: 4c21 7d27 7d22 7d28 7d22 e193 7e7e ff7d  L!}'}"}(}"..~~.}
00000030: 23c0 217d 217d 217d 207d 347d 227d 267d  #.!}!}!} }4}"}&}
00000040: 207d 207d 207d 207d 257d 2628 e54c 217d   } } } }%}&(.L!}
```

It starts with `7eff`, which is characteristic of Point-to-Point Protocol. We can decode the packets with scapy, a framework for network protocol analysis. However, first we have to de-frame the PPP frames since there doesn’t seem to be a tool for this automatically. There are two main tasks, first split up the frames by the `7e` delimiters, and then remove the byte stuffing within the frame, since PPP will escape certain bytes with the `7d` prefix followed by the byte XOR `0x20`. Finally, the frame can be passed to scapy for analysis. This is a VERY lax de-framer because sometimes frames seemed to not be started or terminated properly.

```python
def decode(ch):
    buf2 = b""
    esc = False

    for x in ch:
        if x == 0x7e:
            if buf2 != b"\xFF" and buf2 != b"":
                PPP(buf2).show()
            buf2 = b""
            esc = False
        elif esc:
            esc = False
            buf2 += bytes([x^0x20])
        elif x == 0x7d:
            esc = True
        else:
            buf2 += bytes([x])

    if len(buf2) > 0:
        PPP(buf2).show()
```

(This code is really awful CTF code, please ignore the 200 awful spaghetti things I'm doing in this snippet.)

Now we can see what the packets mean. In particular, we spot these ones:

```
###[ HDLC ]###
  address   = 0xff
  control   = 0x3
###[ PPP Link Layer ]###
     proto     = Link Control Protocol
###[ PPP Link Control Protocol ]###
        code      = Configure-Ack
        id        = 0x2
        len       = 28
        \options   \
.....
         |###[ PPP LCP Option ]###
         |  type      = Authentication-protocol
         |  len       = 5
         |  auth_protocol= Challenge-response authentication protocol
         |  algorithm = MS-CHAP
.....

###[ PPP Link Layer ]###
  proto     = Challenge Handshake Authentication Protocol
###[ PPP Challenge Handshake Authentication Protocol ]###
     code      = Response
     id        = 0x0
     len       = 67
     value_size= 49
     value     = 000000000000000000000000000000000000000000000006c2e3af0f2f7760
2e9831310b56924f3428b05ad60c7a2b401
     optional_name= 'rocketman2674'
```

and

```
###[ PPP Link Layer ]###
  proto     = Challenge Handshake Authentication Protocol
###[ PPP Challenge Handshake Authentication Protocol ]###
     code      = Challenge
     id        = 0x0
     len       = 26
     value_size= 8
     value     = 12810ab88c7f1c74
     optional_name= 'GRNDSTTNA8F6C'

###[ PPP Link Layer ]###
  proto     = Challenge Handshake Authentication Protocol
###[ PPP Challenge Handshake Authentication Protocol ]###
     code      = Success
     id        = 0x0
     len       = 4
     data      = ''
```

We can see in this exchange that the client has negotiated `MS-CHAP` authentication and then authenticates to the server successfully. MS-CHAP uses NetNTLMv1 hashes, which can be cracked very easily. We just need the username (`rocketman2674`), the "challenge" which is used as a salt for the hash, and the hash itself. The format of the response in MS-CHAP (according to RFC2433) is 49 bytes, including 24 bytes of stuff we ignore, 24 bytes of hash, and one byte of stuff we also ignore. We can now convert the data into a John-the-Ripper compatible hash like

```
username:$NETNTLM$challenge$hash

rocketman2674:$NETNTLM$12810ab88c7f1c74$6c2e3af0f2f77602e9831310b56924f3428b05ad60c7a2b4
```

Technically, you can use hashcat as well but I didn't want to bother with the hashcat flags. Put this hash in a text file and run `john file.txt`. No need to specify 4 digit pins because john will complete in literal seconds anyway.

```
Proceeding with incremental:ASCII
9435             (rocketman2674)
1g 0:00:00:08 DONE 3/3 (2020-05-26 03:07) 0.1212g/s 10225Kp/s 10225Kc/s 10225KC/s 97xx..94b4
Use the "--show --format=netntlm" options to display all of the cracked passwords reliably
Session completed
```

Use `rocketman2674` with password `9435` to log in to the ground station, then execute the `flag` command to get the flag.

Resources and other writeups

- https://en.wikipedia.org/wiki/Hayes_command_set
- https://en.wikipedia.org/wiki/Dual-tone_multi-frequency_signaling
- https://github.com/kamalmostafa/minimodem
- https://en.wikipedia.org/wiki/Point-to-Point_Protocol
- https://tools.ietf.org/html/rfc2433
- https://www.openwall.com/john/

Phasors to Stun

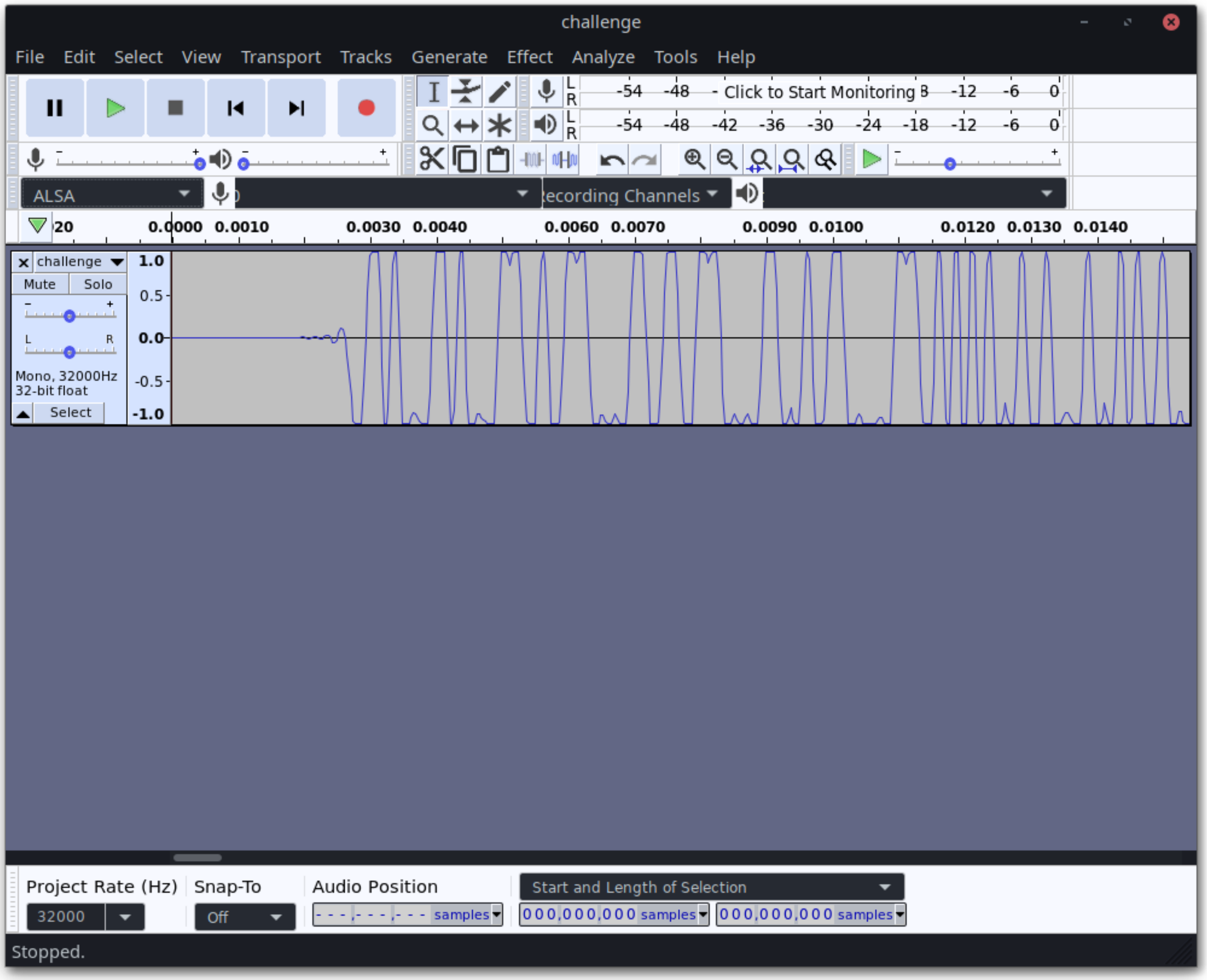
Category: Communication Systems **Points (final):** 62 **Solves:** 71

> Demodulate the data from an SDR capture and you will find a flag. It is a wav file, but that doesn't mean its audio data.

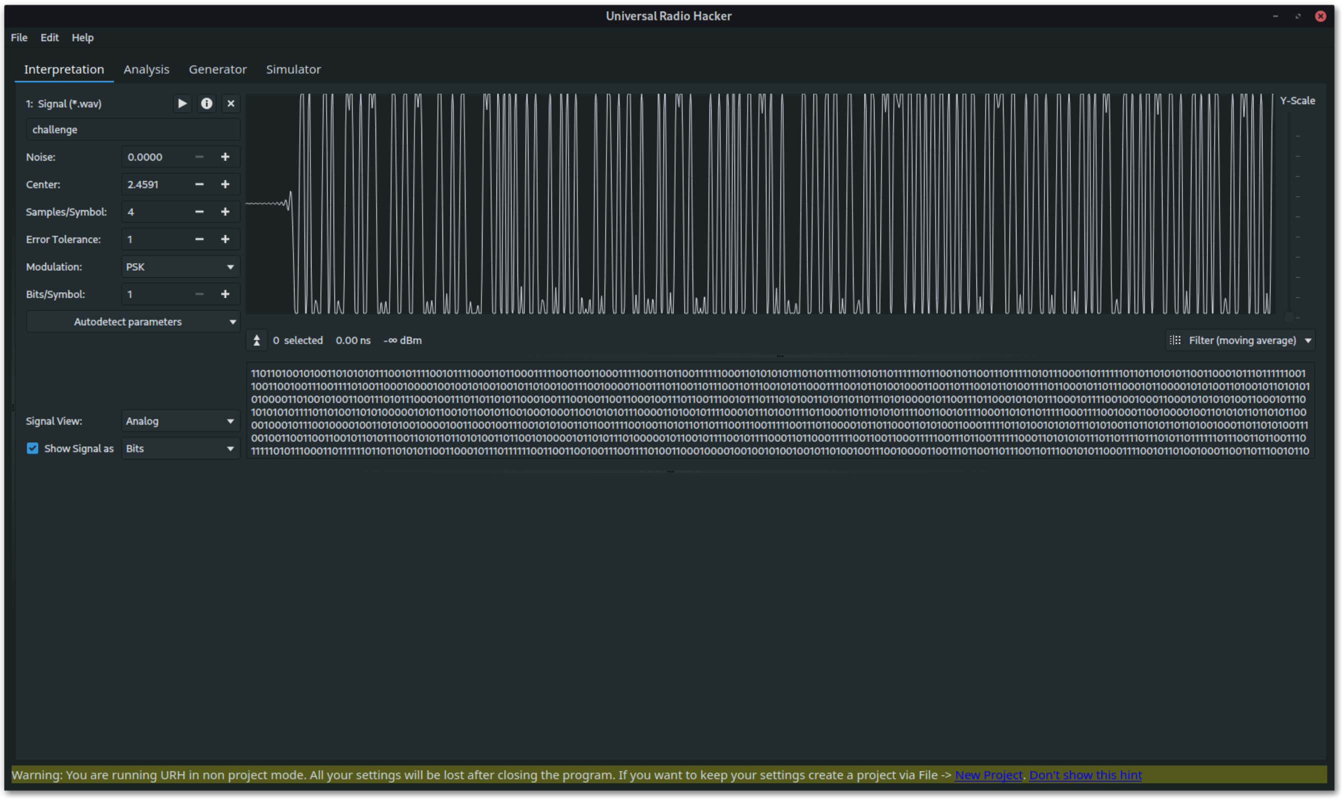
Write-up

by haskal

The provided WAV file contains a signal that looks like this:



This looks suspiciously like Phase Shift Keying (PSK) and it's a very clean signal (this is also hinted at by the challenge name). We can use Universal Radio Hacker to demod this with very little effort.



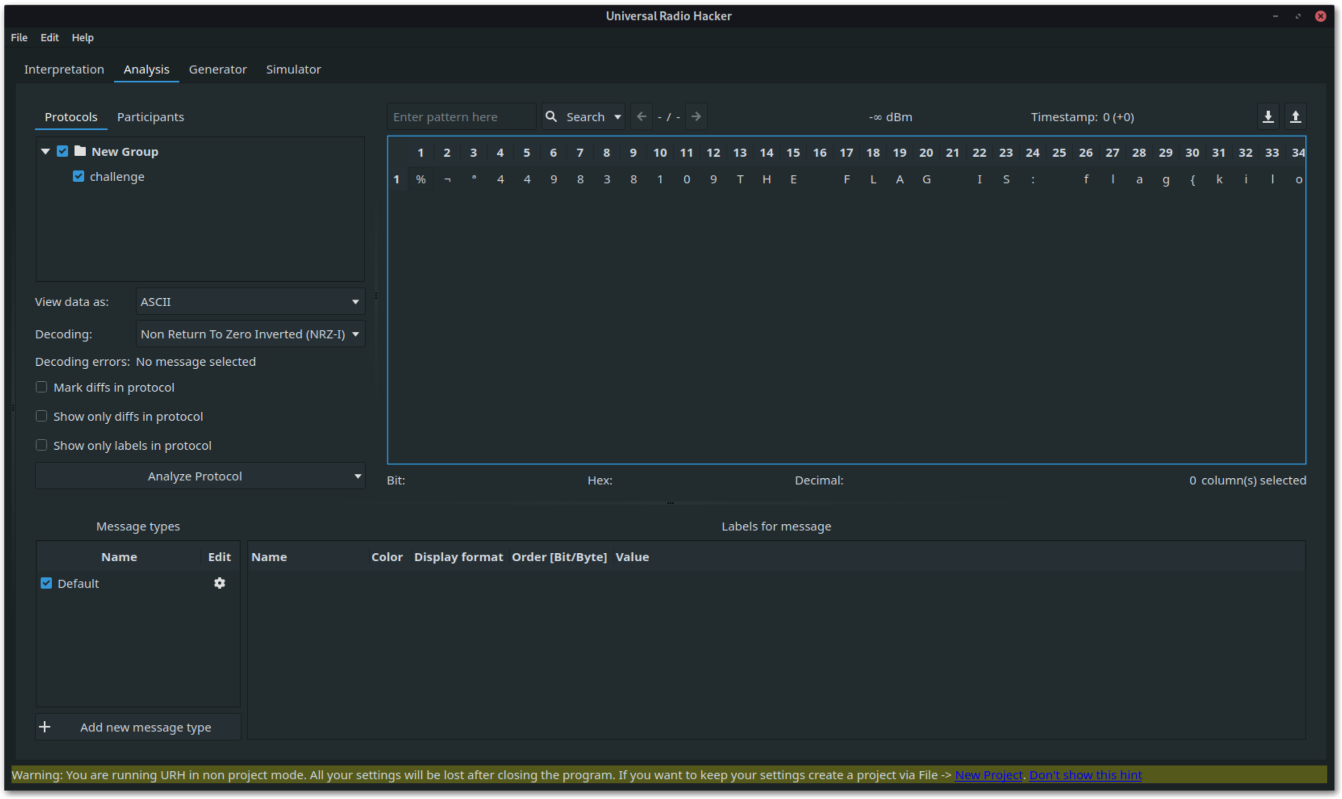
1: Signal (*.wav)
challenge
Noise: 0.0000
Center: 2.4591
Samples/Symbol: 4
Error Tolerance: 1
Modulation: PSK
Bits/Symbol: 1
Signal View: Analog
Show Signal as Bits

0 selected 0.00 ns -∞ dBm

```
1101101001010011010101010111001011110010111100011011000111110011001100011111001110110011111100011010101011101101111011101011011111101110011011001110111110101110001101111110110110101011001100010111011111111001
10011001001110011110100110001000010010010100100101101001001110010000110011101100110111001011100101011000111100101101001000110011011100101101001111011000101101110001011000010101001101001011010101
0100001101001010011001110101110001001110110110101100010011100100110011000100111011001110010111011101010011010110110111010100001011001110110001010101110001011110010010001100010101010100110001011110
1010101011110110100110101000001010110010110010110010001000110010101011100001101001011110001011101001111011000110111010101111001100101111000110101101111100011110010001100100001001101010110110101100
0010001011100100001001101010010000100110001001110010101001101100111100100110101101101110011100111110011101100001011011000110101001100011110110100101010111010100110110101101101001000110110101001111
0010011001100110010110101110011010110110101001101100101000010110101110100000101100101111001011110001101100011110011001100011110011101100111110001101010101110110111101110101101111110111001101100111 0
1111101011100011011111101101101010110011000101110111111001100110010011100111101001100010000100100101001001011010010011100100001100111011001101110011011100101011000111100101101001000110011011100101 10
```

Warning: You are running URH in non project mode. All your settings will be lost after closing the program. If you want to keep your settings create a project via File -> New Project. Don't show this hint

Select PSK modulation, then click "Autodetect parameters". Then move to Analysis:



New Group
challenge

View data as: ASCII
Decoding: Non Return To Zero Inverted (NRZ-I)
Decoding errors: No message selected
Mark diffs in protocol
Show only diffs in protocol
Show only labels in protocol

-/- -∞ dBm Timestamp: 0 (+0)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
1	%	¬	*	4	4	9	8	3	8	1	0	9	T	H	E		F	L	A	G		I	S	:		f	l	a	g	{	k	i	l	o

Bit: Hex: Decimal: 0 column(s) selected

Name	Edit
Default	

Name	Color	Display format	Order [Bit/Byte]	Value

Warning: You are running URH in non project mode. All your settings will be lost after closing the program. If you want to keep your settings create a project via File -> New Project. Don't show this hint

We discovered that the signal is NRZI (non-return-to-zero inverted) coded, and after selecting this in URH the flag is decoded in the data view.

Resources and other writeups

- https://github.com/jopohl/urh
- https://en.wikipedia.org/wiki/Phase-shift_keying
- https://en.wikipedia.org/wiki/Non-return-to-zero#NRZI

Can you hear me now?

Category: Ground Segment **Points (final)**: 59 points **Solves**: 75

> LaunchDotCom's ground station is streaming telemetry data from its Carnac 1.0 satellite on a TCP port. Implement a decoder from the XTCE definition.

Given files: `telemetry.zip`

Write-up

by erin (`barzamin`).

The provided zip file contains `telemetry.xcte`, an XTCE file defining the telemetry protocol streaming from the challenge server.

XTCE is a XML-based protocol description format, used to provide a machine-readable definition of the bit layout in a telemetry stream. I could use COSMOS to load this XTCE definition, but instead I just figured out what the XTCE file meant (without really reading the XTCE specification, because nobody has time for that) and wrote a quick decoder by hand. I have never touched XTCE before this and only briefly looked at CCSDS during a rocketry project for school before deciding not to use it, so any knowledge I have about it comes from things like "google" and "NASA presentations from 2008" and "definitely legitimately obtained specification pdfs".

I captured some telmetry data from the server by running

```
(cat THE_TICKET) | nc hearmenow.satellitesabove.me 5032 > data
```

`telemetry.xtce` describes every packet in the payload is headed by a header of the form (apparently, "abstract" things in XTCE are an instanceable template for a description of parameters; this one gets instanced in every packet as the header):

```xml
<xtce:SequenceContainer name="AbstractTM Packet Header"
                        shortDescription="CCSDS TM Packet Header"
                        abstract="true">
  <xtce:EntryList>
    <xtce:ParameterRefEntry parameterRef="CCSDS_VERSION"/>
    <xtce:ParameterRefEntry parameterRef="CCSDS_TYPE"/>
    <xtce:ParameterRefEntry parameterRef="CCSDS_SEC_HD"/>
    <xtce:ParameterRefEntry parameterRef="CCSDS_APID"/>
    <xtce:ParameterRefEntry parameterRef="CCSDS_GP_FLAGS"/>
    <xtce:ParameterRefEntry parameterRef="CCSDS_SSC"/>
    <xtce:ParameterRefEntry parameterRef="CCSDS_PLENGTH"/>
  </xtce:EntryList>
</xtce:SequenceContainer>
```

The `parameterRefs` point to `xtce:Parameters` in the `xtce:ParameterSet` nearer the top of the file; the parameters in the header are defined there as

```xml
<!-- Parameters used by space packet primary header -->
<xtce:Parameter parameterTypeRef="3BitInteger" name="CCSDS_VERSION"/>
<xtce:Parameter parameterTypeRef="1BitInteger" name="CCSDS_TYPE"/>
<xtce:Parameter parameterTypeRef="1BitInteger" name="CCSDS_SEC_HD"/>
<xtce:Parameter parameterTypeRef="11BitInteger" name="CCSDS_APID"/>
<xtce:Parameter parameterTypeRef="2BitInteger" name="CCSDS_GP_FLAGS"/>
<xtce:Parameter parameterTypeRef="14BitInteger" name="CCSDS_SSC"/>
<xtce:Parameter parameterTypeRef="2ByteInteger" name="CCSDS_PLENGTH"/>
```

The `{n}BitInteger` parameter types are defined further up in the file as exactly what you'd expect them to be. We now know what packet headers look like; let's look for something flag related. A `Flag Packet` is defined in several places in the file (once as an "abstract" packet, which I don't really understand the significance of); it contains a body of parameters `FLAG1` through `FLAG120`, all defined upfile as 7-bit integers

```
<xtce:Parameter parameterTypeRef="7BitInteger" name="FLAGxxx"/>
```

The header associated with the flag packet is

```
<xtce:BaseContainer containerRef="AbstractTM Packet Header">
    <xtce:RestrictionCriteria>
        <xtce:ComparisonList>
            <xtce:Comparison parameterRef="CCSDS_VERSION" value="0"/>
            <xtce:Comparison parameterRef="CCSDS_TYPE" value="0"/>
            <xtce:Comparison parameterRef="CCSDS_SEC_HD" value="0"/>
            <xtce:Comparison parameterRef="CCSDS_APID" value="102"/>
        </xtce:ComparisonList>
    </xtce:RestrictionCriteria>
</xtce:BaseContainer>
```

The APID is specific to the flag packet; we can just search for it in the stream and decode from there. I threw together some python (using `bitflags`) to decode the flag from the data I recorded:

```
from bitstring import Bits, BitArray, ConstBitStream

b = ConstBitStream(filename='./data')
packetlocs = list(b.findall('0x0066'))
print(f"found packets:  {packetlocs}")

for loc in packetlocs:
  b.pos = loc
  ver = b.read(3).uint
  ty = b.read(1).bin
  sec_hd = b.read(1).bin
  apid = b.read(11).uint
  gp_flags = b.read(2).bin
  ssc = b.read(14).uint
  plength = b.read(16).uint

  print(ver, ty, sec_hd, hex(apid), gp_flags, ssc, plength)

  flag = []
  for i in range(120):
    flag.append(chr(b.read(7).uint))
  print(''.join(flag))
```

Which produced the flag:

```
λ ~/has/cyhmn
» python decode.py
found packets:  [600, 1904, 3208]
0 0 0 0x66 11 1919 94
flag{delta98823mike:GAFbfoYquKzWaSFdWeYHGMDosGaBTnMbwD_kqwuj
MhhNPaA9t7Iay8GY6CdGUwrYVa_AetBJEqJ6XO1XHl0kbHA}OP`P<
```

Resources and other writeups

- https://www.omg.org/xt
- https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20090017706.pdf
- https://bitstring.readthedocs.io/

I see what you did there

Category: Ground Segment **Points (final)**: 146 points **Solves**: 23

> Your rival seems has been tracking satellites with a hobbiest antenna, and it's causing a lot of noise on my ground lines. Help me figure out what he's tracking so I can see what she's tracking

Given files: `rbs_m2-juliet20230hotel.tar.bz2`, `examples.zip`

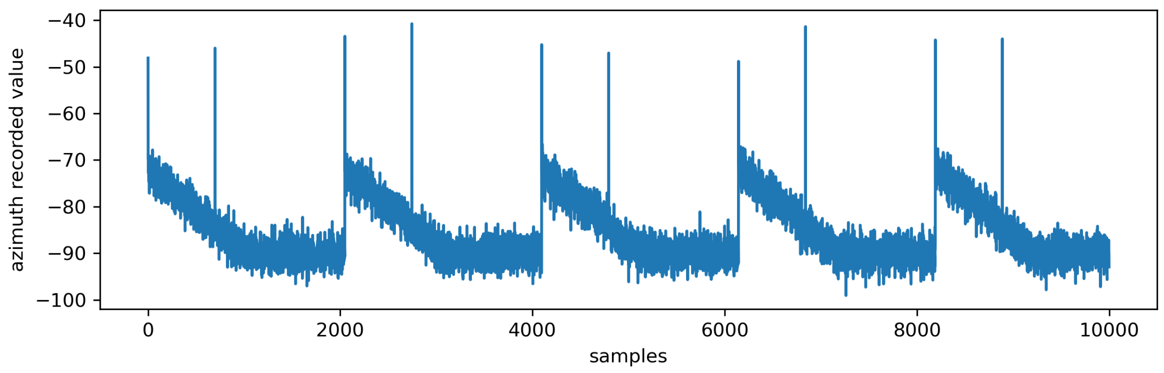
Write-up

by erin (`barzamin`).

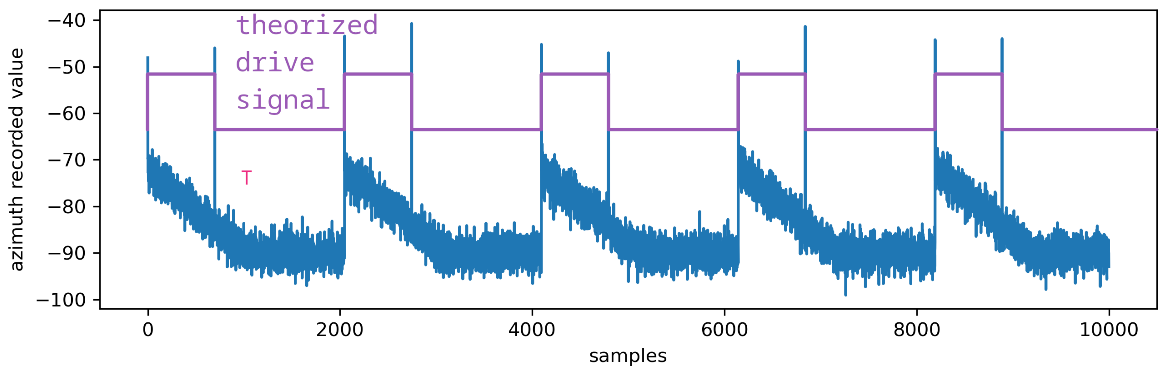
In this challenge, we're sneakily observing two PWM-based motor drivers on an antenna mount drive; the duty cycle of each is proportional to the angle of the alt/az axis it's controlling. We'd like to figure out what that duty cycle is at any given instant, so we can reconstruct where it's pointing; if we have that data, *over time*, we can compare it to a list of satellite traces and figure out what it was pointing at. However, we don't have the direct drive waveform; we have a timestamped recording of the RF noise each axis is generating. Sidechannel analysis time!

We're given a list of possible satellites with Two-Line Elements for each, and an examples zip with example sidechannel recordings and the satellites we belong to. I didn't actually automate interaction with the server, so I just did `(echo 'THE_TICKET') | nc antenna.satellitesabove.me 5034` and grabbed the relvant information (location and observation start time/length) to paste into my code.

Looking at one of the sample captures (CANX-7 azimuth), we see a fairly distinctive periodic pattern of peaks rising out of noise:



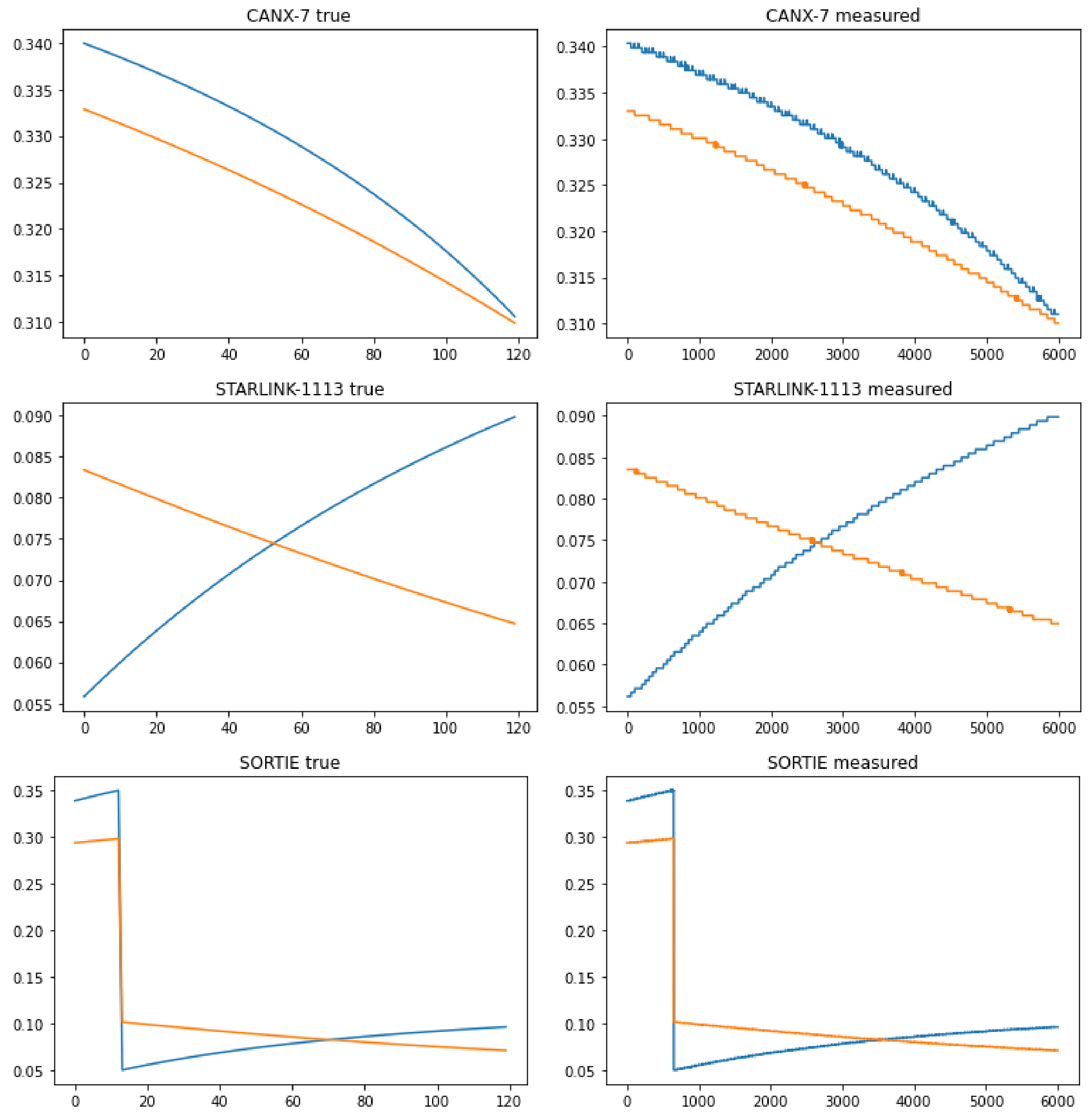
I guessed these were transients when the drive signal flipped state, inferring that the drive signal would look something like this:



By identifying these peaks in the time-series signal, we can compute the drive period and duty cycle by measuring the distances between peaks. I performed peak detection with `scipy.signal.find_peaks()`; the distance between every first and second peak is measured to get the high time, while the distance between the first peak and third peak is used to extract the drive frequency (which is constant over the whole signal, because it's pulse *width* modulation):

```
def compute_duty_cycle(x):
    peaks, props = signal.find_peaks(x, prominence=20)
    peaks = np.concatenate(([0], peaks))
    T_high = peaks[1::2] - peaks[::2]
    T_motor_drive = peaks[2]-peaks[0]
    return T_high/T_motor_drive, T_motor_drive
```

Using this code, I computed the duty cycles for all examples, and also predicted the ideal duty cycles from the tracker (cf Track That Sat). They match, quantitatively, very well (I also checked L2 distance or something similar but I don't remember what it was other than it being on the order of ~1e-6).



Now that we know we can compute duty cycle over time from the sidechannel signal, we need to figure out how to search for a satellite that *could* be tracked at a given time. We have a list of all possible candidate satellites, so we can just figure out which ones are above the horizon and precompute the alt/az duty cycles over the timeframe we're given using Skyfield:

```
satellites = load.tle_file('../tts/examples/active.txt')
by_name = {sat.name: sat for sat in satellites}
visible = []
for sat in tqdm(satellites):
    alt, az, _ = (sat-station).at(time(60)).altaz()
    if alt.degrees > 0:
        visible.append(sat)

station = Topos(37.5, 15.08)

ts = load.timescale()
def time(dt):
    return ts.utc(datetime.fromtimestamp(1585993950.588545+dt,tz=utc))

def compute_true_trajectory(sat, loc, times):
    orients = []
    for dt in times:
        t = time(dt)
        diff = sat - loc
        topocentric = diff.at(t)
        alt, az, dist = topocentric.altaz()
        orients.append([az, alt])
    return orients

def map_angle(angle):
    return 0.05 + angle.degrees * (0.35-0.05)/(180)

def azalt_to_duty(az, alt):
    if az.degrees > 180:
        return [map_angle(Angle(degrees=az.degrees%180)), map_angle(Angle(degrees=180-alt.degrees))]
    else:
        return [map_angle(az), map_angle(alt)]

duty_traces = []
for sat in tqdm(visible):
    duty_traces.append([azalt_to_duty(*o)
                for o in compute_true_trajectory(sat, station, times)])
dtraces = [np.array(x) for x in duty_traces]
```

Then for each of the unknown signals, we can just find the satellite with trace closest to the ones measured (minimize error in az + error in alt between guess and query to match):

```
for X in [X0, X1, X2]:
    measured_az, drive_period_az = compute_duty_cycle(X[:,1])
    measured_alt, drive_period_alt = compute_duty_cycle(X[:,2])

    errors = np.array([np.linalg.norm(measured_az[::60]-trace[:-1,0])
        + np.linalg.norm(measured_alt[::60]-trace[:-1,1])
        for trace in dtraces])

    print(np.argmin(errors), visible[np.argmin(errors)])
```

This gives a printout which looks something like

```
64 EarthSatellite 'APRIZESAT 2' number=28366 epoch=2020-04-10T06:04:32Z
384 EarthSatellite 'ELYSIUM STAR 2 & LFF' number=43760 epoch=2020-04-10T11:25:42Z
269 EarthSatellite 'LINGQIAO VIDEO B' number=40960 epoch=2020-04-10T10:20:29Z
```

From there, just copypasta and grab flag.

Full code

```
import numpy as np
import matplotlib.pyplot as plt
import csv
from scipy import signal
from skyfield.api import Topos, load, utc
from skyfield.units import Angle
from datetime import datetime
from tqdm.notebook import trange, tqdm
import json

satellites = load.tle_file('../tts/examples/active.txt')
by_name = {sat.name: sat for sat in satellites}

ts = load.timescale()
def time(dt):
    return ts.utc(datetime.fromtimestamp(1585993950.588545+dt,tz=utc))

def map_angle(angle):
    return 0.05 + angle.degrees * (0.35-0.05)/(180)

def azalt_to_duty(az, alt):
    if az.degrees > 180:
        return [map_angle(Angle(degrees=az.degrees%180)), map_angle(Angle(degrees=180-alt.degrees))]
    else:
        return [map_angle(az), map_angle(alt)]

def compute_true_trajectory(sat, loc, times):
    orients = []
    for dt in times:
        t = time(dt)
        diff = sat - loc
        topocentric = diff.at(t)
        alt, az, dist = topocentric.altaz()
        orients.append([az, alt])
    return orients

def compute_duty_cycle(x):
    peaks, props = signal.find_peaks(x, prominence=20)
    peaks = np.concatenate(([0], peaks))
    T_high = peaks[1::2] - peaks[::2]
    T_motor_drive = peaks[2]-peaks[0]
    return T_high/T_motor_drive, T_motor_drive


Fs = 102400 # hz
Ts = 1/Fs
station = Topos(37.5, 15.08)

X0 = np.genfromtxt('./rbs_m2-juliet20230hotel/signal_0.csv', delimiter=',')
X1 = np.genfromtxt('./rbs_m2-juliet20230hotel/signal_1.csv', delimiter=',')
X2 = np.genfromtxt('./rbs_m2-juliet20230hotel/signal_2.csv', delimiter=',')

visible = []
for sat in tqdm(satellites):
    alt, az, _ = (sat-station).at(time(60)).altaz()
    if alt.degrees > 0:
        visible.append(sat)

# generate tracks for all visible satellites
duty_traces = []
for sat in tqdm(visible):
    duty_traces.append([azalt_to_duty(*o)
                  for o in compute_true_trajectory(sat, station, times)])
dtraces = [np.array(x) for x in duty_traces]

for X in [X0, X1, X2]:
    measured_az, drive_period_az = compute_duty_cycle(X[:,1])
    measured_alt, drive_period_alt = compute_duty_cycle(X[:,2])

    errors = np.array([np.linalg.norm(measured_az[::60]-trace[:-1,0])
        + np.linalg.norm(measured_alt[::60]-trace[:-1,1])
        for trace in dtraces])

    print(np.argmin(errors), visible[np.argmin(errors)])
```

Resources and other writeups

- https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks.html
- https://rhodesmill.org/skyfield/

Talk to me, Goose

Category: Ground Segment **Points (final)**: 94 points **Solves**: 42

> LaunchDotCom has a new satellite, the Carnac 2.0. What can you do with it from its design doc?

Given files: `cmd_telemetry_defs.zip`, `LaunchDotCom_Carnac_2.zip`

Write-up

by erin (`barzamin`).

Inside `LaunchDotCom_Carnac_2.zip` is a documentation PDF describing the "satellite" we're connecting to. One of the interesting things it notes is that "LaunchDotCom recommends Ball Aerospace's COSMOS suite of software for command and telemetry processing with the Carnac 2.0."; we already set this up for the *That's not on my calendar* challenge (written up by `haskal` in this report).

We're given an XTCE file inside `cmd_telemetry_defs.zip`, `cmd_telemetry_defs.xtce`, which describes the command/telemetry protocol to the satellite. COSMOS can load an XTCE definition into a COSMOS configuration, so we just generated a new COSMOS configuration tree and imported `cmd_telemetry_defs.xtce`. Out of the box, this won't identify packets properly for some reason.

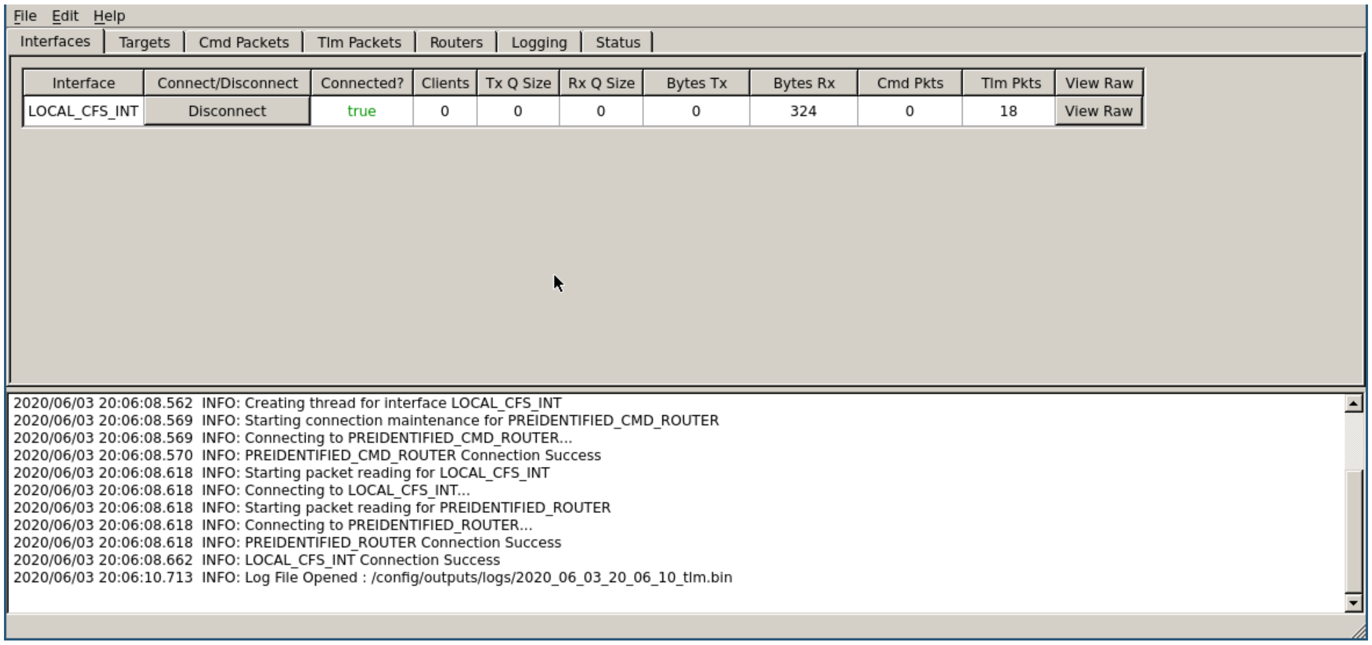
To fix this, we created a config `$COSMOS/config/targets/CHALLENGE1/target.txt` containing `TLM_UNIQUE_ID_MODE` to make COSMOS fall back to a non-hash-based packet ID mode for the imported XTCE `CHALLENGE` target to work.

We also had to write an interface config for COSMOS so it could connect to the TCP-tunneled-CCSDS telemetry port the challenge gives us after a ticket; ours was of the form

```
TITLE 'COSMOS Command and Telemetry Server'
# 10 nil: read, write timeouts
# length: protocol frame decoder
#   32: bit offset in packet to length field
#   16: size of length bitfield
#   7:  length value offset (true # bytes read is length + this)
#   1:  1 byte per count in length field
#   BIG_ENDIAN: endianness *of length field*
INTERFACE LOCAL_CFS_INT tcpip_client_interface.rb {ip} {port} {port} 10 nil LENGTH 32 16 7 1 BIG_ENDIAN
  TARGET CHALLENGE1
  TARGET SYSTEM
```

and was generated by a script which connected to the challenge, passed the token, and templated/wrote out this interface config file to `$COSMOS_CONFIG_DIR/config/tools/cmd_tlm_server/cmd_tlm_server.txt`. We ran COSMOS in the Docker container detailed in *That's not on my calendar* (I'm so sorry for the lack of themes in the docker container; everything looks like Win95).

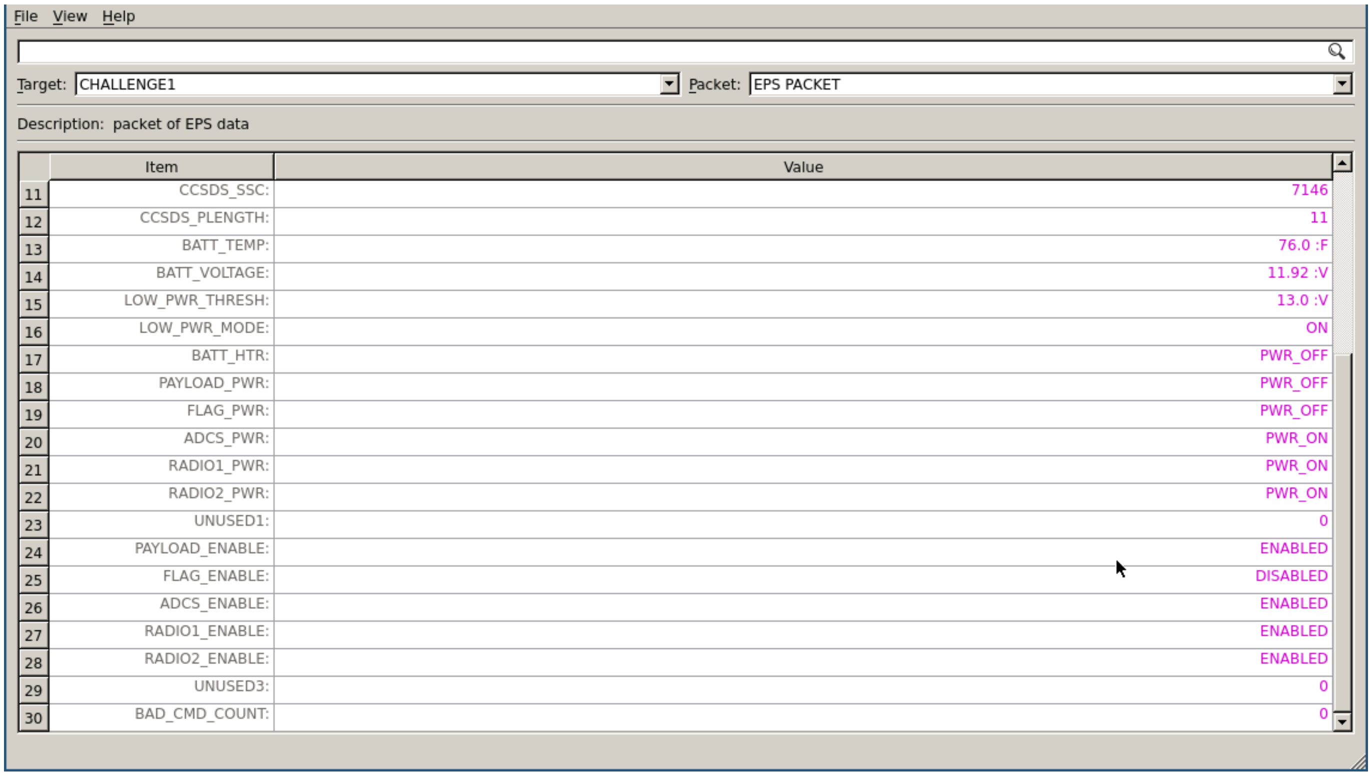
Connect with COSMOS [hacker voice im in]:


File Edit Help
Interfaces Targets Cmd Packets Tlm Packets Routers Logging Status

Interface	Connect/Disconnect	Connected?	Clients	Tx Q Size	Rx Q Size	Bytes Tx	Bytes Rx	Cmd Pkts	Tlm Pkts	View Raw
LOCAL_CFS_INT	Disconnect	true	0	0	0	0	324	0	18	View Raw

```
2020/06/03 20:06:08.562  INFO: Creating thread for interface LOCAL_CFS_INT
2020/06/03 20:06:08.569  INFO: Starting connection maintenance for PREIDENTIFIED_CMD_ROUTER
2020/06/03 20:06:08.569  INFO: Connecting to PREIDENTIFIED_CMD_ROUTER...
2020/06/03 20:06:08.570  INFO: PREIDENTIFIED_CMD_ROUTER Connection Success
2020/06/03 20:06:08.618  INFO: Starting packet reading for LOCAL_CFS_INT
2020/06/03 20:06:08.618  INFO: Connecting to LOCAL_CFS_INT...
2020/06/03 20:06:08.618  INFO: Starting packet reading for PREIDENTIFIED_ROUTER
2020/06/03 20:06:08.618  INFO: Connecting to PREIDENTIFIED_ROUTER...
2020/06/03 20:06:08.618  INFO: PREIDENTIFIED_ROUTER Connection Success
2020/06/03 20:06:08.662  INFO: LOCAL_CFS_INT Connection Success
2020/06/03 20:06:10.713  INFO: Log File Opened : /config/outputs/logs/2020_06_03_20_06_10_tlm.bin
```

Almost the packets we're getting are `EPS PACKET`s indicating undervoltage:

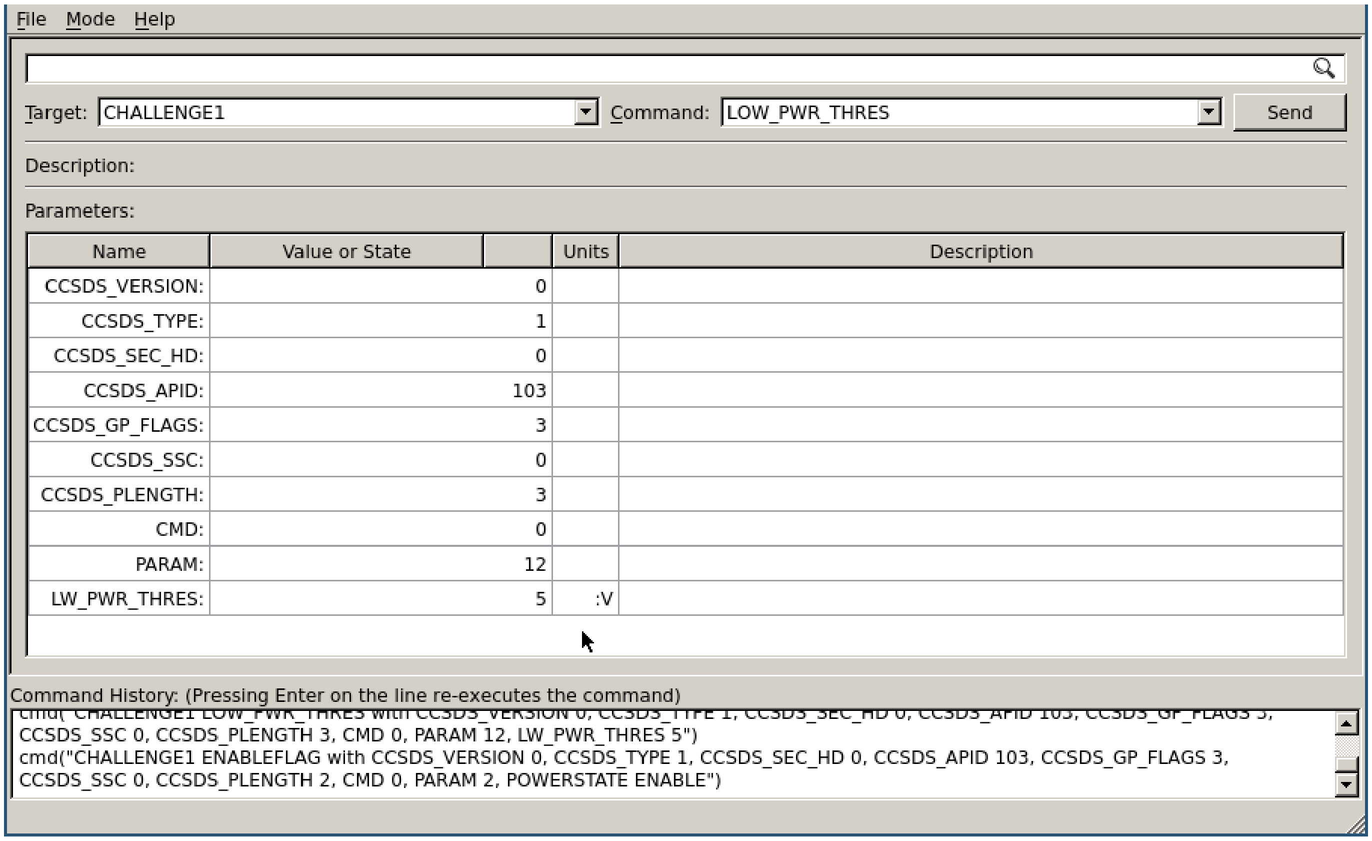


Target: CHALLENGE1 Packet: EPS PACKET

Description: packet of EPS data

	Item	Value
11	CCSDS_SSC:	7146
12	CCSDS_PLENGTH:	11
13	BATT_TEMP:	76.0 :F
14	BATT_VOLTAGE:	11.92 :V
15	LOW_PWR_THRESH:	13.0 :V
16	LOW_PWR_MODE:	ON
17	BATT_HTR:	PWR_OFF
18	PAYLOAD_PWR:	PWR_OFF
19	FLAG_PWR:	PWR_OFF
20	ADCS_PWR:	PWR_ON
21	RADIO1_PWR:	PWR_ON
22	RADIO2_PWR:	PWR_ON
23	UNUSED1:	0
24	PAYLOAD_ENABLE:	ENABLED
25	FLAG_ENABLE:	DISABLED
26	ADCS_ENABLE:	ENABLED
27	RADIO1_ENABLE:	ENABLED
28	RADIO2_ENABLE:	ENABLED
29	UNUSED3:	0
30	BAD_CMD_COUNT:	0

According to the manual PDF, the EPS shuts off "non-essential subsystems" in this state (which probably includes the subsystem that should be sending us flag packets). Immediately after starting a connection, though, we can send a `LOW_PWR_THRESH` command with a `LW_PWR_THRES` of 5V to put the low-power threshold below the battery voltage so the EPS thinks it's no longer running out of power:



File Mode Help

Target: CHALLENGE1 Command: LOW_PWR_THRES Send

Description:

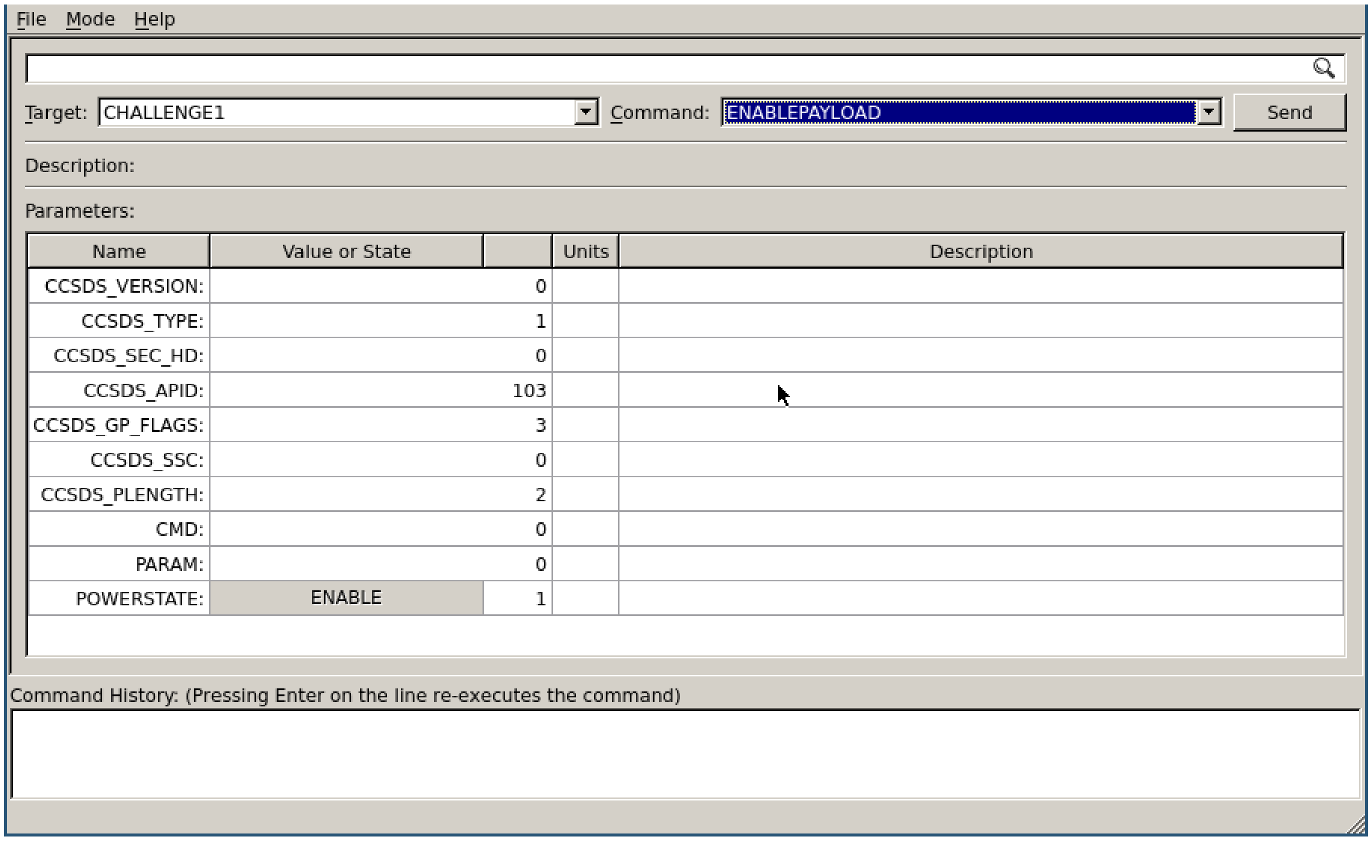
Parameters:

Name	Value or State		Units	Description
CCSDS_VERSION:		0		
CCSDS_TYPE:		1		
CCSDS_SEC_HD:		0		
CCSDS_APID:		103		
CCSDS_GP_FLAGS:		3		
CCSDS_SSC:		0		
CCSDS_PLENGTH:		3		
CMD:		0		
PARAM:		12		
LW_PWR_THRES:		5	:V	

Command History: (Pressing Enter on the line re-executes the command)

```
cmd("CHALLENGE1 LOW_PWR_THRES with CCSDS_VERSION 0, CCSDS_TYPE 1, CCSDS_SEC_HD 0, CCSDS_APID 103, CCSDS_GP_FLAGS 3,
CCSDS_SSC 0, CCSDS_PLENGTH 3, CMD 0, PARAM 12, LW_PWR_THRES 5")
cmd("CHALLENGE1 ENABLEFLAG with CCSDS_VERSION 0, CCSDS_TYPE 1, CCSDS_SEC_HD 0, CCSDS_APID 103, CCSDS_GP_FLAGS 3,
CCSDS_SSC 0, CCSDS_PLENGTH 2, CMD 0, PARAM 2, POWERSTATE ENABLE")
```

And then all we have to do is send an `ENABLEPAYLOAD` command:



File Mode Help

Target: CHALLENGE1 Command: ENABLEPAYLOAD Send

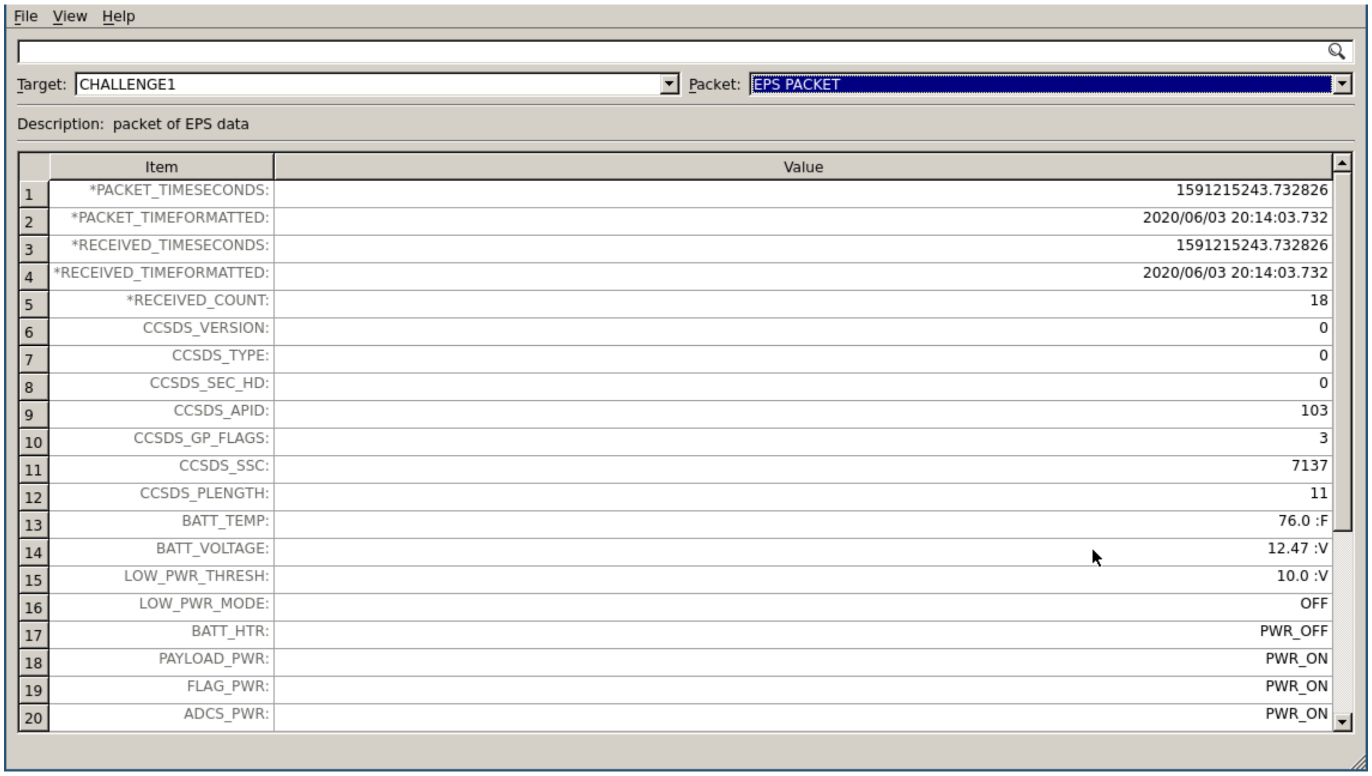
Description:

Parameters:

Name	Value or State		Units	Description
CCSDS_VERSION:		0		
CCSDS_TYPE:		1		
CCSDS_SEC_HD:		0		
CCSDS_APID:		103		
CCSDS_GP_FLAGS:		3		
CCSDS_SSC:		0		
CCSDS_PLENGTH:		2		
CMD:		0		
PARAM:		0		
POWERSTATE:	ENABLE	1		

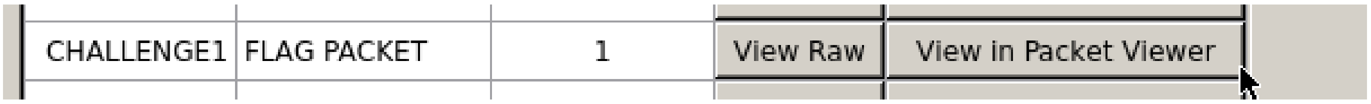
Command History: (Pressing Enter on the line re-executes the command)

The goose is now happy and `FLAG_PWR` is on!

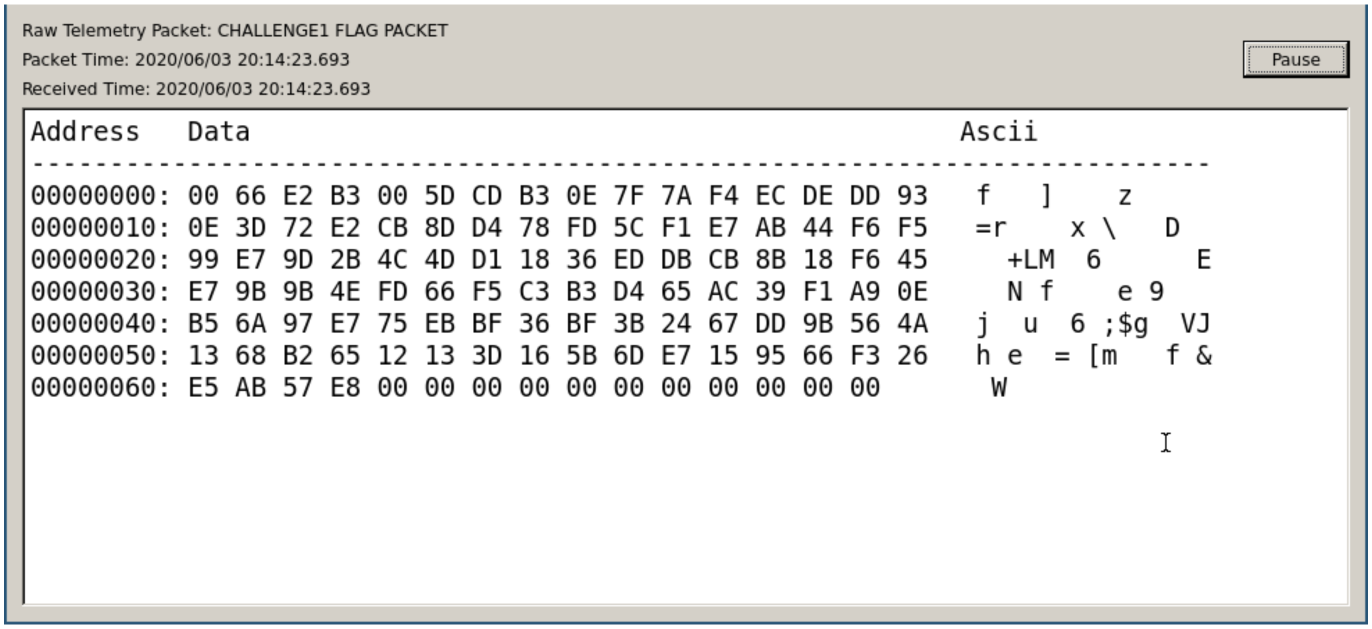


	Item	Value
1	*PACKET_TIMESECONDS:	1591215243.732826
2	*PACKET_TIMEFORMATTED:	2020/06/03 20:14:03.732
3	*RECEIVED_TIMESECONDS:	1591215243.732826
4	*RECEIVED_TIMEFORMATTED:	2020/06/03 20:14:03.732
5	*RECEIVED_COUNT:	18
6	CCSDS_VERSION:	0
7	CCSDS_TYPE:	0
8	CCSDS_SEC_HD:	0
9	CCSDS_APID:	103
10	CCSDS_GP_FLAGS:	3
11	CCSDS_SSC:	7137
12	CCSDS_PLENGTH:	11
13	BATT_TEMP:	76.0 :F
14	BATT_VOLTAGE:	12.47 :V
15	LOW_PWR_THRESH:	10.0 :V
16	LOW_PWR_MODE:	OFF
17	BATT_HTR:	PWR_OFF
18	PAYLOAD_PWR:	PWR_ON
19	FLAG_PWR:	PWR_ON
20	ADCS_PWR:	PWR_ON

The flag will come back whenever the flag task's scheduler fires, in a FLAG_PACKET:



Instead of decoding the packet into fields in COSMOS (that won't show us an ASCII string, just FLAG1, FLAG2, etc integer fields in a list), I copy-pasted the flag packet from COSMOS's hexdump:



```
Raw Telemetry Packet: CHALLENGE1 FLAG PACKET
Packet Time: 2020/06/03 20:14:23.693
Received Time: 2020/06/03 20:14:23.693

Address   Data                                              Ascii
---------------------------------------------------------------------------
00000000: 00 66 E2 B3 00 5D CD B3 0E 7F 7A F4 EC DE DD 93   f   ]    z
00000010: 0E 3D 72 E2 CB 8D D4 78 FD 5C F1 E7 AB 44 F6 F5   =r    x \   D
00000020: 99 E7 9D 2B 4C 4D D1 18 36 ED DB CB 8B 18 F6 45     +LM  6       E
00000030: E7 9B 9B 4E FD 66 F5 C3 B3 D4 65 AC 39 F1 A9 0E     N f    e 9
00000040: B5 6A 97 E7 75 EB BF 36 BF 3B 24 67 DD 9B 56 4A   j  u  6 ;$g  VJ
00000050: 13 68 B2 65 12 13 3D 16 5B 6D E7 15 95 66 F3 26   h e  = [m   f &
00000060: E5 AB 57 E8 00 00 00 00 00 00 00 00 00 00 00 00     W
```

And threw together a quick script to decode by inspection of the XTCE file's defintion of the flag layout (basically identical to *Can you hear me now?*):

```
from bitstring import Bits, BitArray, ConstBitStream
```

```
b = ConstBitStream('THE_HEX_HERE')
packetlocs = list(b.findall('0x0066'))
print(f"found packets:  {packetlocs}")

for loc in packetlocs:
    b.pos = loc
    ver = b.read(3).uint
    ty = b.read(1).bin
    sec_hd = b.read(1).bin
    apid = b.read(11).uint
    gp_flags = b.read(2).bin
    ssc = b.read(14).uint
    plength = b.read(16).uint

    print(ver, ty, sec_hd, hex(apid), gp_flags, ssc, f'len={plength} ({hex(plength)})')

    flag = []
    for i in range(120):
        flag.append(chr(b.read(7).uint))
    print(''.join(flag))
```

Flag achieved; another can-you-use-COSMOS challenge down.

Resources

- https://cosmosrb.com/docs/xtce/
- https://cosmosrb.com/docs/interfaces/
- https://cosmosrb.com/docs/protocols/

That's not on my calendar

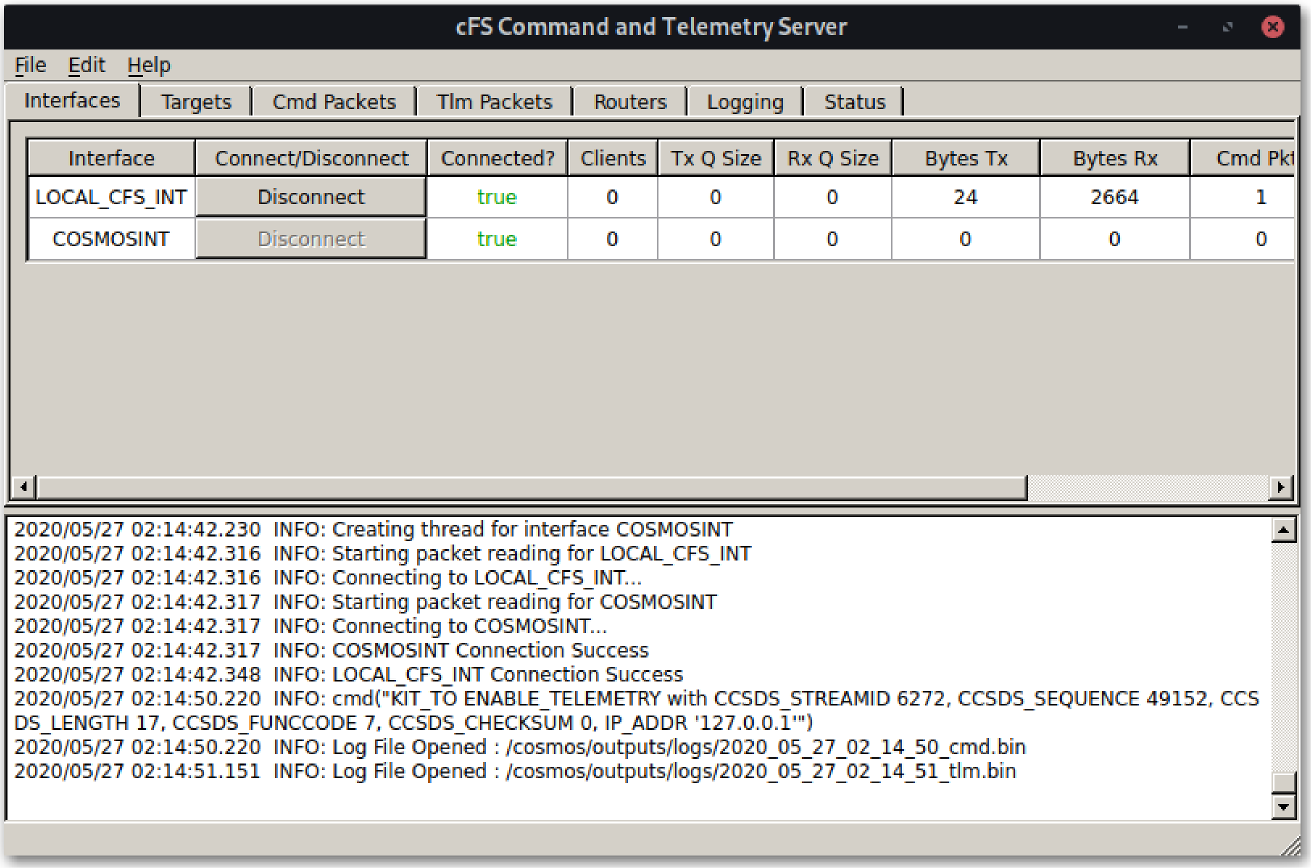
Category: Payload Modules **Points (final):** 80 **Solves:** 52

> Time for a really gentle introduction to cFS and Cosmos, hopefully you can schedule time to learn it!
>
> Build instructions:
>
> $ ./setup.sh $ source ~/.bashrc $ rm Gemfile.lock $ bundle install
>
> Hint: You will need to enable telemetry locally on the satellite, the udp forwarder will provide it to you as TCP from there

Write-up

by haskal

Two files are provided, one is a COSMOS directory tree for accessing a virtual satellite running NASA's Core Flight System that can be booted up using the provided netcat endpoint. COSMOS is an open-source command and control framework for satellites using cFS. Booting up COSMOS is enormously complicated, so Docker can be used to automate the setup. We adapted the Ball Aerospace COSMOS Docker image, and created a script to configure COSMOS to connect to the CTF's satellite instance automatically by writing the configuration file at `cosmos/config/tools/cmd_tlm_server/cmd_tlm_server.txt`. When COSMOS is successfully connected to the CTF instance it looks like this (no themes were installed in the Docker container so it looks like Windows 95, I'm so sorry,)


cFS Command and Telemetry Server
File Edit Help
Interfaces Targets Cmd Packets Tlm Packets Routers Logging Status

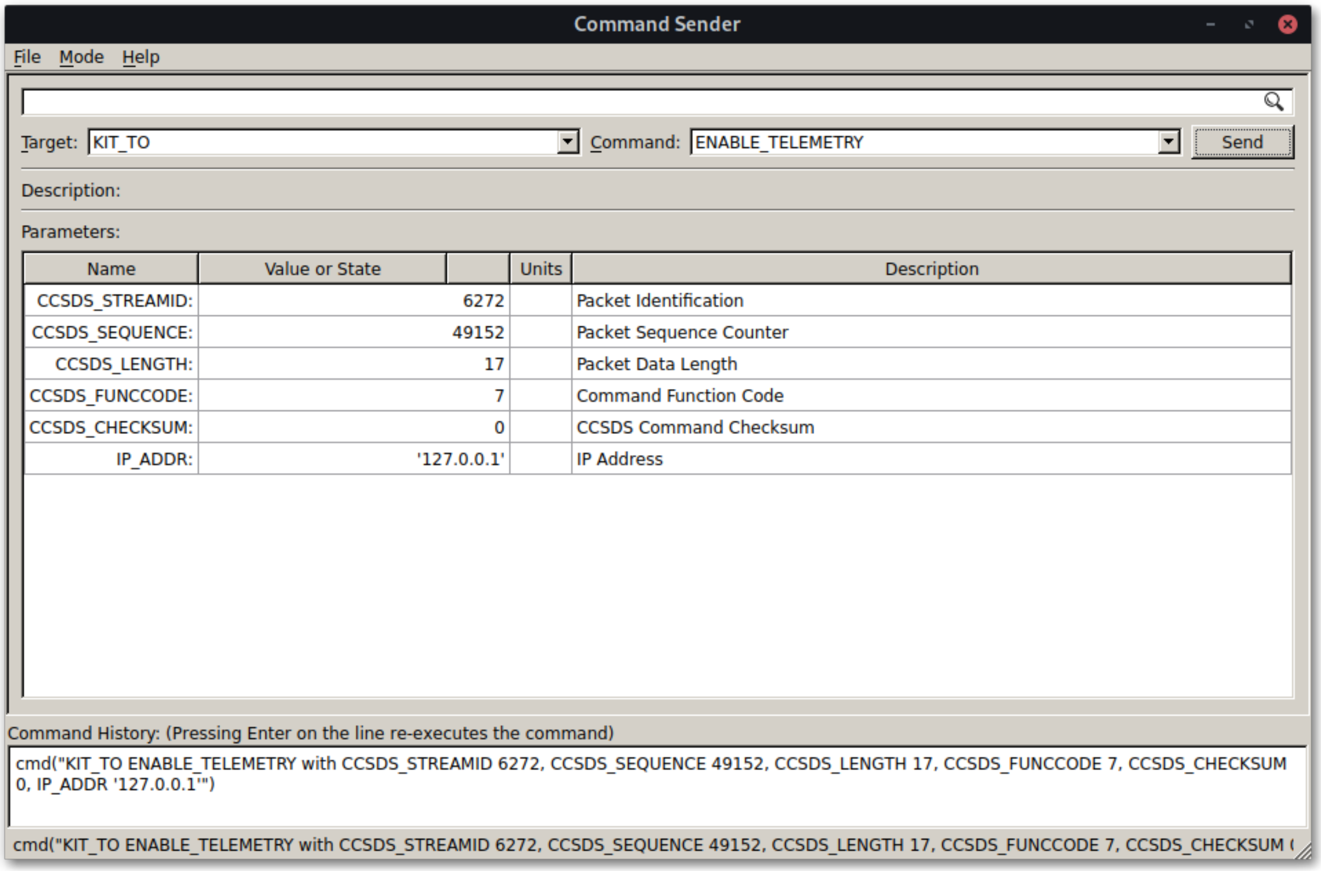
Interface	Connect/Disconnect	Connected?	Clients	Tx Q Size	Rx Q Size	Bytes Tx	Bytes Rx	Cmd Pkt
LOCAL_CFS_INT	Disconnect	true	0	0	0	24	2664	1
COSMOSINT	Disconnect	true	0	0	0	0	0	0

```
2020/05/27 02:14:42.230  INFO: Creating thread for interface COSMOSINT
2020/05/27 02:14:42.316  INFO: Starting packet reading for LOCAL_CFS_INT
2020/05/27 02:14:42.316  INFO: Connecting to LOCAL_CFS_INT...
2020/05/27 02:14:42.317  INFO: Starting packet reading for COSMOSINT
2020/05/27 02:14:42.317  INFO: Connecting to COSMOSINT...
2020/05/27 02:14:42.317  INFO: COSMOSINT Connection Success
2020/05/27 02:14:42.348  INFO: LOCAL_CFS_INT Connection Success
2020/05/27 02:14:50.220  INFO: cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCS
DS_LENGTH 17, CCSDS_FUNCCODE 7, CCSDS_CHECKSUM 0, IP_ADDR '127.0.0.1'")
2020/05/27 02:14:50.220  INFO: Log File Opened : /cosmos/outputs/logs/2020_05_27_02_14_50_cmd.bin
2020/05/27 02:14:51.151  INFO: Log File Opened : /cosmos/outputs/logs/2020_05_27_02_14_51_tlm.bin
```

The second file is an archive of some JSON configuration for the satellite operating system. Importantly, it contains the types of telemetry messages the satellite can send, and the default scheduler table that configures when the satellite sends the messages. There is a telemetry packet that sends us the flag configured here

```
{"message": {
   "name":  "KIT_TO_SEND_FLAG_MID",
   "descr": "Super Secret Flag Sending Telemetry Message",
   "id": 42,
   "stream-id": 33304,
   "seq-seg": 192,
   "length": 256
}},
```

As hinted in the description, we can send a KIT_TO ENABLE_TELEMETRY command using the COSMOS command sender. KIT_TO is the subsystem for Telemetry Output, and it has commands available to control the telemetry sending.



Target: KIT_TO
Command: ENABLE_TELEMETRY

Description:

Parameters:

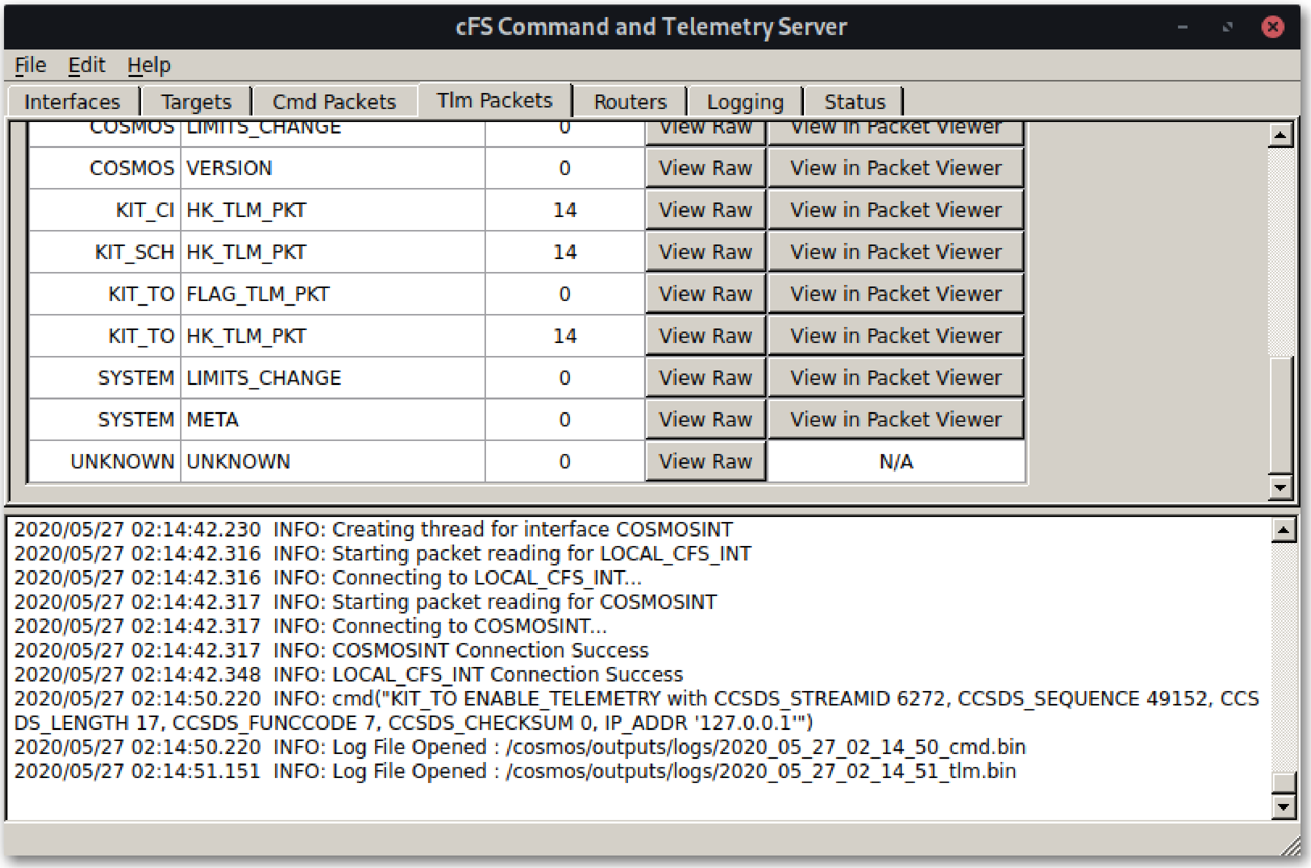
Name	Value or State		Units	Description
CCSDS_STREAMID:	6272			Packet Identification
CCSDS_SEQUENCE:	49152			Packet Sequence Counter
CCSDS_LENGTH:	17			Packet Data Length
CCSDS_FUNCCODE:	7			Command Function Code
CCSDS_CHECKSUM:	0			CCSDS Command Checksum
IP_ADDR:	'127.0.0.1'			IP Address

Command History: (Pressing Enter on the line re-executes the command)

```
cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 17, CCSDS_FUNCCODE 7, CCSDS_CHECKSUM
0, IP_ADDR '127.0.0.1'")
```

```
cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 17, CCSDS_FUNCCODE 7, CCSDS_CHECKSUM (
```

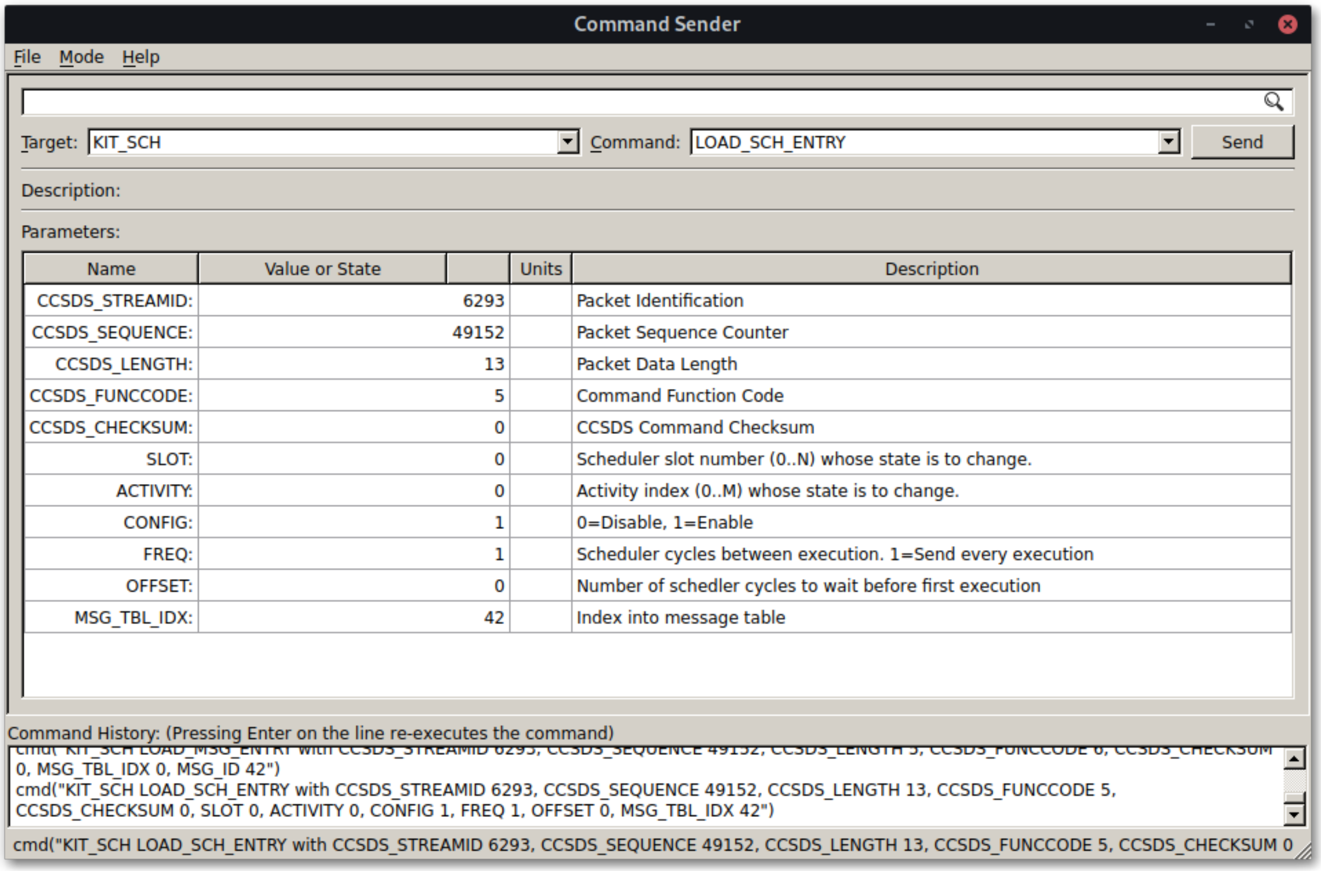
However we find that even after telemetry is enabled, we’re not getting the flag telemetry even though we start to receive other types of telemetry messages.


cFS Command and Telemetry Server
File Edit Help
Interfaces Targets Cmd Packets Tlm Packets Routers Logging Status

COSMOS	LIMITS_CHANGE	0	View Raw	View in Packet Viewer
COSMOS	VERSION	0	View Raw	View in Packet Viewer
KIT_CI	HK_TLM_PKT	14	View Raw	View in Packet Viewer
KIT_SCH	HK_TLM_PKT	14	View Raw	View in Packet Viewer
KIT_TO	FLAG_TLM_PKT	0	View Raw	View in Packet Viewer
KIT_TO	HK_TLM_PKT	14	View Raw	View in Packet Viewer
SYSTEM	LIMITS_CHANGE	0	View Raw	View in Packet Viewer
SYSTEM	META	0	View Raw	View in Packet Viewer
UNKNOWN	UNKNOWN	0	View Raw	N/A

```
2020/05/27 02:14:42.230  INFO: Creating thread for interface COSMOSINT
2020/05/27 02:14:42.316  INFO: Starting packet reading for LOCAL_CFS_INT
2020/05/27 02:14:42.316  INFO: Connecting to LOCAL_CFS_INT...
2020/05/27 02:14:42.317  INFO: Starting packet reading for COSMOSINT
2020/05/27 02:14:42.317  INFO: Connecting to COSMOSINT...
2020/05/27 02:14:42.317  INFO: COSMOSINT Connection Success
2020/05/27 02:14:42.348  INFO: LOCAL_CFS_INT Connection Success
2020/05/27 02:14:50.220  INFO: cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCS
DS_LENGTH 17, CCSDS_FUNCCODE 7, CCSDS_CHECKSUM 0, IP_ADDR '127.0.0.1'")
2020/05/27 02:14:50.220  INFO: Log File Opened : /cosmos/outputs/logs/2020_05_27_02_14_50_cmd.bin
2020/05/27 02:14:51.151  INFO: Log File Opened : /cosmos/outputs/logs/2020_05_27_02_14_51_tlm.bin
```

This is because, if we look back at the JSON config we find that the scheduler doesn't have any slots where it sends flag telemetry packets. The key part of this challenge is that the scheduler can be configured at runtime using commands for KIT_SCH (the scheduler subsystem), as we found out by exploring the available commands in COSMOS. Particularly there is a command LOAD_SCH_ENTRY that allows us to overwrite one of the scheduler entries. We can use the command sender to send this command and load slot 0, activity 0 (or any slot, it doesn't matter) to be for message ID 42 (the flag packet ID), and to be enabled (1).



Target: KIT_SCH Command: LOAD_SCH_ENTRY

Description:

Parameters:

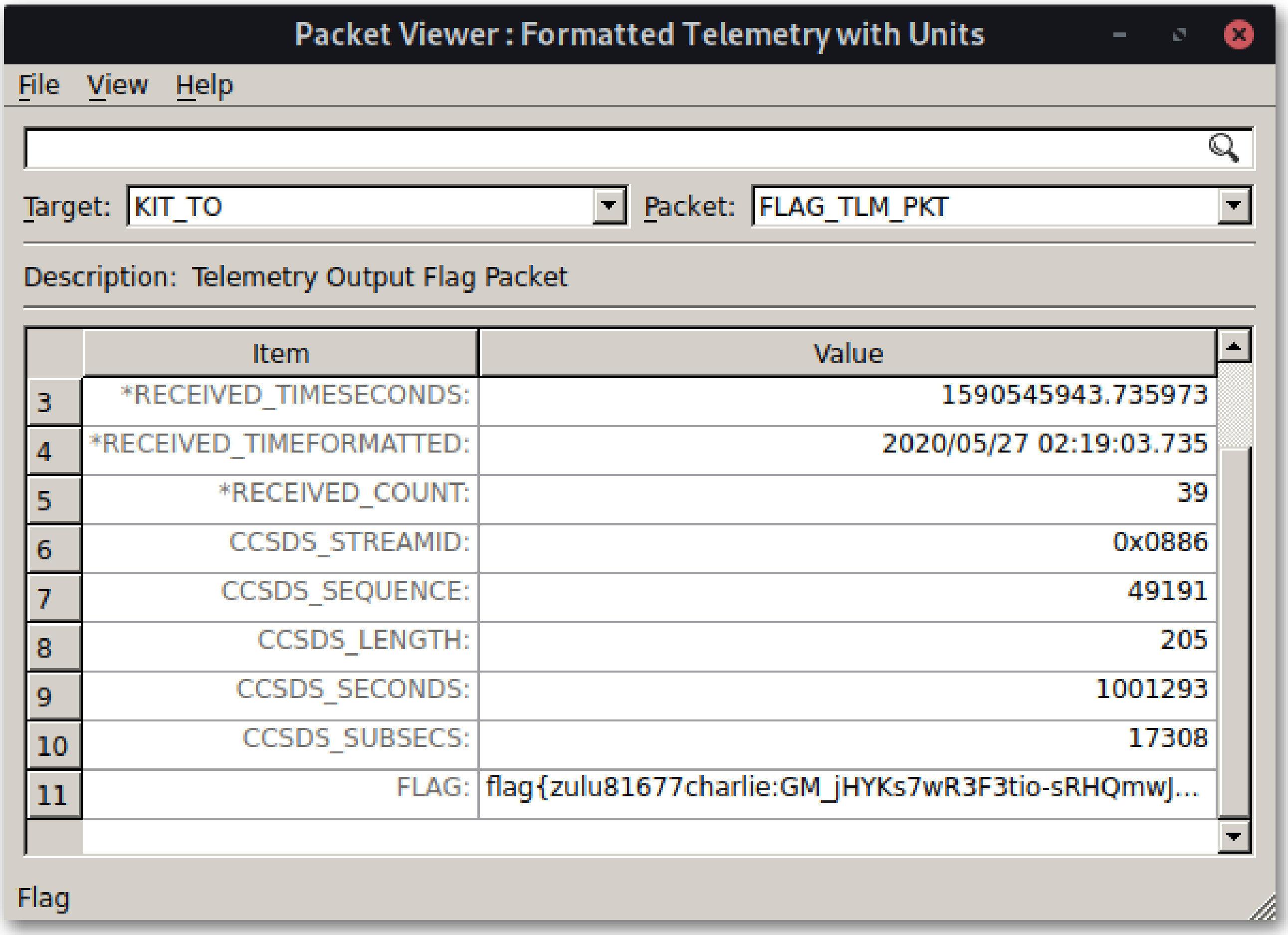
Name	Value or State		Units	Description
CCSDS_STREAMID:	6293			Packet Identification
CCSDS_SEQUENCE:	49152			Packet Sequence Counter
CCSDS_LENGTH:	13			Packet Data Length
CCSDS_FUNCCODE:	5			Command Function Code
CCSDS_CHECKSUM:	0			CCSDS Command Checksum
SLOT:	0			Scheduler slot number (0..N) whose state is to change.
ACTIVITY:	0			Activity index (0..M) whose state is to change.
CONFIG:	1			0=Disable, 1=Enable
FREQ:	1			Scheduler cycles between execution. 1=Send every execution
OFFSET:	0			Number of schedler cycles to wait before first execution
MSG_TBL_IDX:	42			Index into message table

Command History: (Pressing Enter on the line re-executes the command)

```
cmd("KIT_SCH LOAD_MSG_ENTRY with CCSDS_STREAMID 6293, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 5, CCSDS_FUNCCODE 6, CCSDS_CHECKSUM
0, MSG_TBL_IDX 0, MSG_ID 42")
cmd("KIT_SCH LOAD_SCH_ENTRY with CCSDS_STREAMID 6293, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 13, CCSDS_FUNCCODE 5,
CCSDS_CHECKSUM 0, SLOT 0, ACTIVITY 0, CONFIG 1, FREQ 1, OFFSET 0, MSG_TBL_IDX 42")
```

```
cmd("KIT_SCH LOAD_SCH_ENTRY with CCSDS_STREAMID 6293, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 13, CCSDS_FUNCCODE 5, CCSDS_CHECKSUM 0
```

Once we write the scheduler entry, the satellite will start sending COSMOS flags, which can be seen in the Packet Viewer.



Target: KIT_TO Packet: FLAG_TLM_PKT

Description: Telemetry Output Flag Packet

	Item	Value
3	*RECEIVED_TIMESECONDS:	1590545943.735973
4	*RECEIVED_TIMEFORMATTED:	2020/05/27 02:19:03.735
5	*RECEIVED_COUNT:	39
6	CCSDS_STREAMID:	0x0886
7	CCSDS_SEQUENCE:	49191
8	CCSDS_LENGTH:	205
9	CCSDS_SECONDS:	1001293
10	CCSDS_SUBSECS:	17308
11	FLAG:	flag{zulu81677charlie:GM_jHYKs7wR3F3tio-sRHQmwJ...

Flag

Resources and other writeups

- https://cosmosrb.com/
- https://cfs.gsfc.nasa.gov/

Leaky Crypto

Category: Payload Modules
Points (final): 223 points
Solves: 11

> My crypto algorithm runs in constant time, so I'm safe from sidechannel leaks, right?
>
> Note: To clarify, the sample data is plaintext inputs, NOT ciphertext

Given files: `leaky-romeo86051romeo.tar.bz2`

Write-up

by Cameron and 5225225

Many optimized implementations of AES utilize lookup tables to combine all steps of each round of the algorithm (SubBytes, ShiftRows, MixColumns, AddKey) into a single operation. For some X (the plaintext or the result from the previous round) and some K (the round key), they are split bytewise and the XOR product of each respective byte pair is used as the index into a lookup table. During the first round of AES, X is the plaintext of the message, and K is the original message key. Accordingly, given some known plaintext, leaking the index into the lookup table for a particular character leaks the corresponding key byte. There are four lookup tables which are used in each iteration of AES (besides the last round) and which is used is determined by the index of the byte MOD 4. We utilized this paper as a reference for both our understanding of AES and the attack we will detail below.

Many CPUs cache RAM accesses so as to speed up subsequent accesses to the same address. This is done because accessing RAM is quite slow, and accessing cache is quite fast. This behavior would imply that on systems which implement such caching methods, there is a correlation between the amount of time it takes to encrypt a particular plaintext and the occurrences of repeated values of a plaintext byte XORd with a key byte. Accordingly, for every i, j, pi ⊕ pj in a family (with i, j being byte indexes, p being the plaintext, and families corresponding to which lookup table is being used), we calculate the average time to encrypt such a message over all messages. We then determine if for any pair of characters pi, pj there is a statistically significant shorter encryption time compared to the average. If so, we can conclude that i ⊕ ki = pj ⊕ kj => pi ⊕ pj = ki ⊕ kj. From this information, we gain a set of redundant system of equations relating different key bytes at different indexes with each other. It is important to note that in order for this attack to work, we must know at least one key byte in each family in order to actually solve each system of equations. Additionally, due to how cache works, this attack only leaks the most significant q bits (q being related to the number of items in a cache line). Once the set of possible partial keys (accounting for the ambiguity in the least significant bits of each derived byte) has been obtained by the above method, an attacker may brute force the remaining unknown key bytes.

In the case of Leaky Crypto, a set of 100,000 plaintexts and corresponding encryption times is provided along with the first six bytes of the encryption key. We ran an analyzer program (see Full code) against these plaintexts to obtain the probable correlation between different indexes in the key with respect to the XOR product of those bytes with plaintext bytes. Per the above, the plaintexts and timing data provided enough information to derive the systems of equations which may be used to solve for key bytes, and the first 6 bytes of the key provided enough information to actually solve said systems of equations. Given the ambiguity of the low bits of each derived key byte, we obtained 214 partial keys with three unknown bytes each. Thus, we reduced the problem of guessing 2128 bits to guessing only 238 bits.

Since we only knew the most significant bits of most of the bytes in the key, we needed to use a candidate generator script (see Full Code) in order to generate trial patterns, with the fully unknown bytes replaced with `??`. This was because we were using Hulk to brute force the keys, which did not support brute forcing the least significant bits of bytes, only fully unknown bytes.

Since the given section of a flag is longer than 16 bytes, which is the size of an AES block, and the satellite is using Electronic Code Book mode, which means that all blocks are encrypted/decrypted separately, we could give hulk the first 16 bytes of the encrypted message as the ciphertext, the first 16 bytes of the flag as the expected plaintext, and then the key is all of the lines output from the `gen.py` script, which is the candidate generator script.

```
    python gen.py |
    xargs -I{} ./hulk d c1a5fe7beb2c70bfab98926627dcff8b 666c61677b726f6d656f383630353172 {} |
    tee output.log
```

After 30 minutes had passed, we successfully brute forced the key, which could then be used to decrypt the rest of the flag.

Full code

Analyzer

```
'''
Created on May 23, 2020

@author: cvkennedy
'''

from itertools import combinations
import matplotlib.pyplot as plt
import numpy as np

def find_outliers(corpus, num_samps, i, j):
    idxs = corpus[i][j].argsort()[:num_samps]
    return idxs

def guess_bytes(corpus, known_keybytes, num_samps, avg):
    candidates = []
    for base in range(4):
        family = [base, base + 4, base + 8, base + 12]
        for combo in combinations(family, 2):
            i,j = combo
            guesses = find_outliers(corpus, num_samps, i, j)
            guesses2 = []
            for guess in guesses:
                cnt = corpus[i][j][guess]
                if cnt-avg < -10:
                    guesses2.append((i, j, guess, cnt-avg))
                print(i, j, guess, cnt - avg)
            candidates.append(tuple(guesses2))
    print(candidates)

if __name__ == '__main__':
    known_keybytes = bytes.fromhex("64c7072487f2")
    secret_data = ("c1a5fe7beb2c70bfab98926627dcff8b9671edc5" +
                   "2441f89fa47797aa023f15f67907ee837b93cd9b" +
                   "194922ebb7c3ca3bd1cbfbc888efe147e8055404" +
                   "7d82872fcee564c1bfd2e0a809568acb5cc08f48" +
                   "36a5f91f43b576a4ee1c6f097c15e1cd4056917f" +
                   "c51c1e5d8157409b11f1600d")

    data = set()
    with open("test.txt", "r") as fp:
        for line in fp:
            pt, timing = line.strip().split(',')
            pt = bytes.fromhex(pt)
            timing = int(timing)
            data.add((pt, timing))
    tavg = sum((d[1] for d in data)) / len(data)
    print("tavg: %d" % tavg)

    known_tly = np.zeros((16, 16, 256))

    for base in range(4):
        print("Building corpus for family %d" % base)
        family = [base, base + 4, base + 8, base + 12]
        for combo in combinations(family, 2):
            times = np.zeros(256)
            counts = np.zeros(256)
            i,j = combo
            print("Working on %d, %d" % (i, j))
            for d in data:
                n = d[0][i] ^ d[0][j]
                c = d[1]
                times[n] += c
                counts[n] += 1
            for c in range(256):
                cnorm = times[c] / counts[c]
                known_tly[i][j][c] = cnorm
                known_tly[j][i][c] = cnorm

    guess_bytes(known_tly, known_keybytes, 4, tavg)
```

Candidate generator

```
guesses = {
    7: [52, 54, 53, 43],
    8: [149, 151, 150, 148],
    9: [173, 174, 175, 172],
    10: [83, 80, 81, 82],
    13: [186, 184, 185, 187],
    14: [2, 1, 0, 3],
    15: [151, 149, 150, 148]
}

known_keybytes = bytes.fromhex("64c7072487f2")

candidates = [list(known_keybytes)]
for i in range(len(known_keybytes), 16):
    new_candidates = []
    if i in guesses.keys():
        for guess in guesses[i]:
            for candidate in candidates:
                new_candidates.append([c for c in candidate] + [guess])
    else:
        for candidate in candidates:
            new_candidates.append([c for c in candidate] + [-1])
    candidates = new_candidates

print("Generated %d candidates" % len(candidates))
for candidate in candidates:
    rep = ''.join([hex(c)[2:].zfill(2) if c != -1 else '??' for c in candidate])
    print(rep)
```

Resources and other writeups

- http://www.jbonneau.com/doc/BM06-CHES-aes_cache_timing.pdf
- https://github.com/pgarba/Hulk

SpaceDB

Category: Payload Modules
Points (final): 79 points
Solves: 53

> The last over-the-space update seems to have broken the housekeeping on our satellite. Our satellite's battery is low and is running out of battery fast. We have a short flyover window to transmit a patch or it'll be lost forever. The battery level is critical enough that even the task scheduling server has shutdown. Thankfully can be fixed without without any exploit knowledge by using the built in APIs provied[sic] by kubOS. Hopefully we can save this one!
>
> Note: When you're done planning, go to low power mode to wait for the next transmission window

Write-up

by Cameron and haskal

Upon connecting to the provided TCP service via netcat, we see that it spawns a telemetry service accessible via HTTP based off the following console output:

```
critical-tel-check info: Detected new telemetry values.
critical-tel-check info: Checking recently inserted telemetry values.
critical-tel-check info: Checking gps subsystem
critical-tel-check info: gps subsystem: OK
critical-tel-check info: reaction_wheel telemetry check.
critical-tel-check info: reaction_wheel subsystem: OK.
critical-tel-check info: eps telemetry check.
critical-tel-check warn: VIDIODE battery voltage too low.
critical-tel-check warn: Solar panel voltage low
critical-tel-check warn: System CRITICAL.
critical-tel-check info: Position: GROUNDPOINT
critical-tel-check warn: Debug telemetry database running at: 3.19.141.137:19369/tel/graphiql
```

Connecting to the provided endpoint yields a graphiql graphql console from which we can run queries against the telemetry database. Using the following query, we can dump the database schema to get an idea of the capabilities of the telemetry interface:

```
fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
```

```
    enumValues(includeDeprecated: true) {
      name
      description
      isDeprecated
      deprecationReason
    }
    possibleTypes {
      ...TypeRef
    }
  }
  fragment InputValue on __InputValue {
    name
    description
    type {
      ...TypeRef
    }
    defaultValue
  }
  fragment TypeRef on __Type {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
                kind
                name
                ofType {
                  kind
                  name
                }
              }
            }
          }
        }
      }
    }
  }

  query IntrospectionQuery {
    __schema {
      queryType {
        name
      }
      mutationType {
        name
      }
      types {
        ...FullType
      }
      directives {
        name
        description
        locations
        args {
          ...InputValue
        }
      }
    }
  }
```

Dumping the schema reveals two things of interest: 1. We may query `telemetry` 2. We may mutate `telemetry` via `delete` and `insertBulk`

From the schema, we see that telemetry data has the following shape:

```
{
  "name": "telemetry",
  "description": "Telemetry entries in database",
  "args": [
    {
      "name": "timestampGe",
      "description": null,
      "type": {
        "kind": "SCALAR",
        "name": "Float",
        "ofType": null
      },
      "defaultValue": null
    },
    {
      "name": "timestampLe",
      "description": null,
      "type": {
        "kind": "SCALAR",
        "name": "Float",
        "ofType": null
      },
      "defaultValue": null
    },
    {
      "name": "subsystem",
      "description": null,
      "type": {
        "kind": "SCALAR",
        "name": "String",
        "ofType": null
      },
      "defaultValue": null
    },
    {
      "name": "parameter",
      "description": null,
      "type": {
        "kind": "SCALAR",
        "name": "String",
        "ofType": null
      },
      "defaultValue": null
    },
    {
      "name": "parameters",
      "description": null,
      "type": {
        "kind": "LIST",
        "name": null,
        "ofType": {
          "kind": "NON_NULL",
          "name": null,
          "ofType": {
            "kind": "SCALAR",
            "name": "String",
            "ofType": null
          }
        }
      },
      "defaultValue": null
    },
    {
      "name": "limit",
      "description": null,
      "type": {
        "kind": "SCALAR",
        "name": "Int",
        "ofType": null
      },
      "defaultValue": null
    }
  ],
  "type": {
    "kind": "NON_NULL",
    "name": null,
    "ofType": {
      "kind": "LIST",
      "name": null,
      "ofType": {
        "kind": "NON_NULL",
        "name": null,
        "ofType": {
          "kind": "OBJECT",
          "name": "Entry",
          "ofType": null
        }
      }
    }
  },
  "isDeprecated": false,
  "deprecationReason": null
}
```

Thus, we can dump the telemetry information via the following command:

```
query {
    telemetry{timestamp, subsystem, parameter, value}
}
```

From the console output, we can see that the issue plaguing the system is a low `VIDIODE` voltage alarm. Thus, in order to fix the alarm, we must spoof the proper `VIDIODE` voltage and trigger a reset of the alarm system. In order to do this, we run the following mutation:

```
mutation spoof {
    delete(timestampGe: 1590232427.582683){success, errors}
    insertBulk(timestamp: 1590232427.582683, entries: [
    {subsystem: "eps", parameter: "VIDIODE", value: "8.0"},
    {subsystem: "eps", parameter: "RESETS_MANUAL", value: "1.0"},
    {subsystem: "eps", parameter: "RESETS_BROWNOUT", value: "1.0"},
    {subsystem: "eps", parameter: "RESETS_AUTO_SOFTWARE", value: "1.0"},
    {subsystem: "eps", parameter: "BATTERY_1_RESETS_MANUAL", value: "1.0"},
    {subsystem: "eps", parameter: "BATTERY_1_RESETS_BROWNOUT", value: "1.0"},
    {subsystem: "eps", parameter: "BATTERY_1_RESETS_AUTO_SOFTWARE", value: "1.0"},
    {subsystem: "eps", parameter: "BATTERY_0_RESETS_MANUAL", value: "1.0"},
    {subsystem: "eps", parameter: "BATTERY_0_RESETS_BROWNOUT", value: "1.0"},
    {subsystem: "eps", parameter: "BATTERY_0_RESETS_AUTO_SOFTWARE", value: "1.0"},
    ]){success, errors}
}
```

The correct value for `VIDIODE` was determined through trial and error. The reset flags were asserted because we didn't know which one we needed, so we just triggered all of them. Were this to be a real satellite, I'm sure nothing bad could possibly happen... We had to run the delete mutation on the most recent telemetry item in order to avoid triggering an alarm for duplicate telemetry data.

After successfully spoofing the telemetry data, we notice in the console output from our session that the scheduler has been activated:

```
critical-tel-check  info: Scheduler service comms started successfully at: 3.19.61.44:14764/sch
/graphiql
```

We visit the provided URL and find another graphiql interface. According to the KubOS documentation, we may issue the following query to enter safe mode and stop any subsequent checks which might kill the scheduler and bring us back to where we started:

```
mutation safe {
    safeMode{success, errors}
}
```

We also see that we can dump the task lists for all available modes with the following query:

```
query dump {
    availableModes{name, path, lastRevised, schedule{tasks{description, delay, time, period,
        app{name, args, config}}, path, filename, timeImported}, active},
    activeMode{name}
}
```

From the information from the dumped task lists, we can see that there are several tasks we may run. First and foremost, we need to fix our power situation by orienting the solar panels towards the sun. We may do this by running the following mutation:

```
mutation patch {
    createMode(name: "patch"){success, errors}
    importRawTaskList(name: "patch", mode: "patch", json: "{\"tasks\":[{\"description\":\"Orien
t solar panels at sun.\",\"delay\":\"0s\",\"time\":null,\"period\":null,\"app\":{\"name\":\"sun
point\",\"args\":null,\"config\":null}},{\"description\":\"Update system telemetry\",\"delay\":
\"1s\",\"time\":null,\"period\":null,\"app\":{\"name\":\"update_tel\",\"args\":null,\"config\":
null}}]}"){success, errors}
    activateMode(name: "patch"){success, errors}
}
```

Subsequently, we need to create a mode which will aim the transmission antenna at the ground, activate the antenna, print the flag to the log, transmit the comms buffer, power down the antenna, and reorient the solar panels. We may do this via the following mutation:

```
mutation {
    importRawTaskList(json:"{\"tasks\":[{\"description\":\"Orient antenna to ground.\",\"delay\
":null,\"time\":\"2020-05-23 16:40:49\",\"period\":null,\"app\":{\"name\":\"groundpoint\",\"arg
s\":null,\"config\":null}},{\"description\":\"Power-up downlink antenna.\",\"delay\":null,\"tim
e\":\"2020-05-23 16:41:09\",\"period\":null,\"app\":{\"name\":\"enable_downlink\",\"args\":null
,\"config\":null}},{\"description\":\"Prints flag to log\",\"delay\":null,\"time\":\"2020-05-23
 16:41:19\",\"period\":null,\"app\":{\"name\":\"request_flag_telemetry\",\"args\":null,\"config
\":null}},{\"description\":\"Power-down downlink antenna.\",\"delay\":null,\"time\":\"2020-05-2
3 16:41:34\",\"period\":null,\"app\":{\"name\":\"disable_downlink\",\"args\":null,\"config\":nu
ll}},{\"description\":\"Orient solar panels at sun.\",\"delay\":null,\"time\":\"2020-05-23 16:4
1:39\",\"period\":null,\"app\":{\"name\":\"sunpoint\",\"args\":null,\"config\":null}}]}",name:"
nominal-op",mode:"transmission"){success,errors}
}
```

Finally, per the challenge directive, we must have the satellite enter low-power mode:

```
mutation low_power {
    activateMode(name: "low_power"){success, errors}
}
```

Bytes Away!

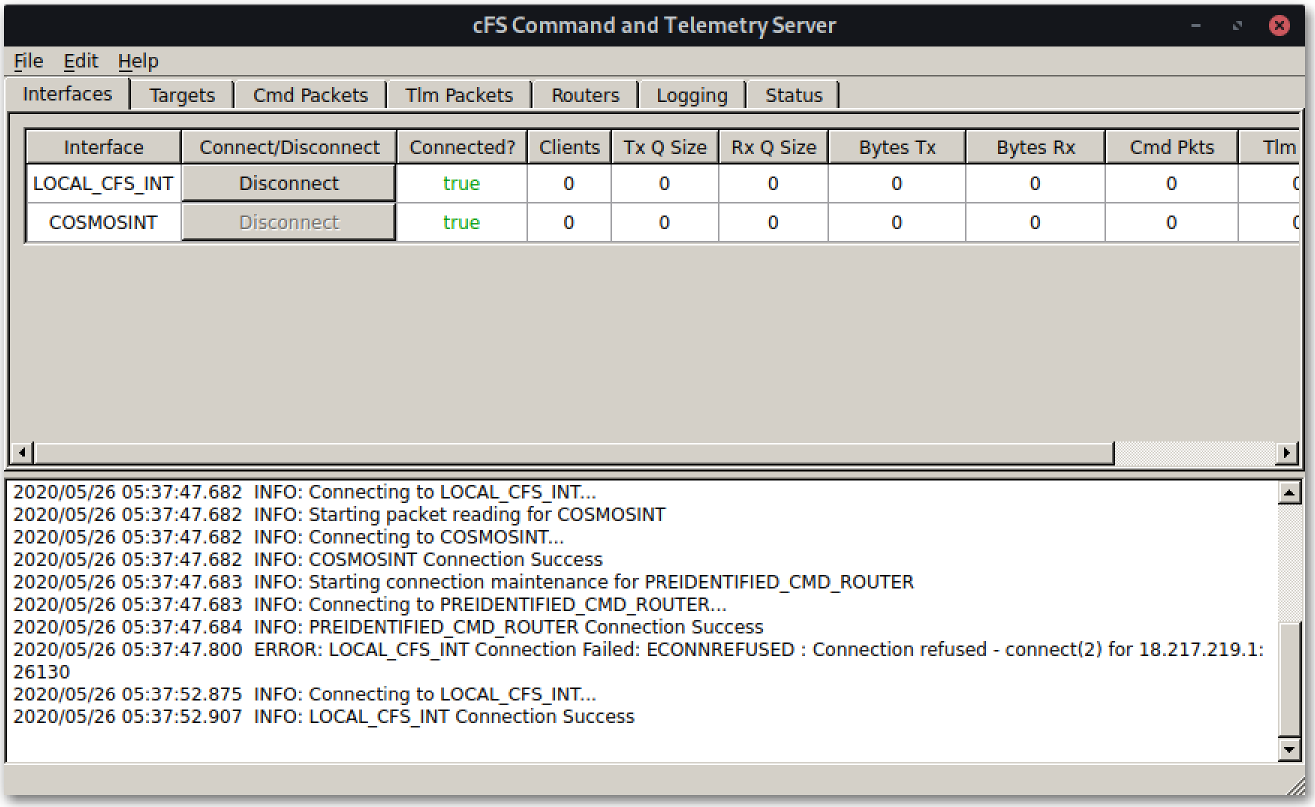
Category: Satellite Bus **Points (final):** 223 **Solves:** 11

> We have an encrypted telemetry link from one of our satellites but we seem to have lost the encryption key. Thankfully we can still send unencrypted commands using our Cosmos interface (included). I've also included the last version of kit_to.so that was updated to the satellite. Can you help us restore communication with the satellite so we can see what error "flag" is being transmitted?

Write-up

by haskal

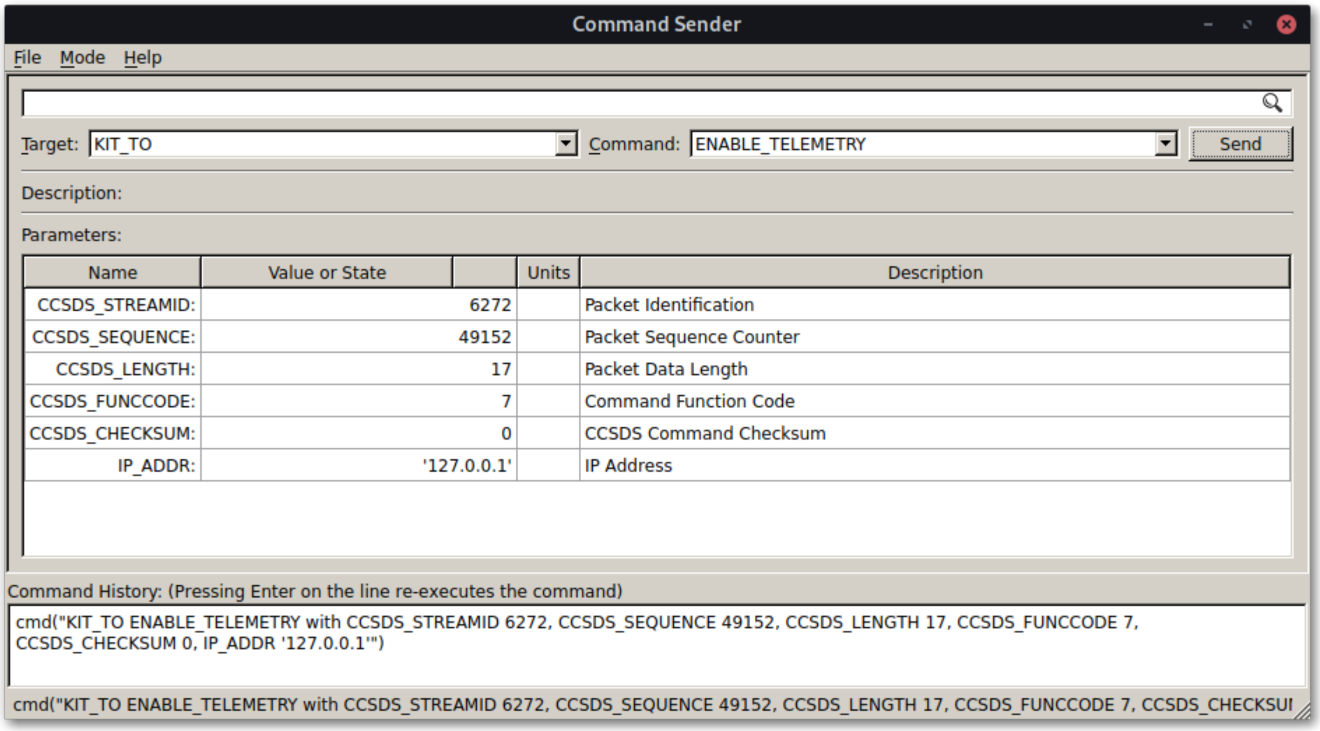
Two files are provided for this challenge, one contains the `kit_to.so` and the other contains a full COSMOS directory tree for accessing the virtual satellite, which can be booted up with the provided netcat endpoint. COSMOS is an open-source command and control framework for satellites using NASA's Core Flight System. The provided COSMOS directory contains everything we need to interact with the virtual satellite, and the `kit_to.so` is part of the code that runs onboard the actual satellite. Booting up COSMOS is enormously complicated, so Docker can be used to automate the setup. We adapted the Ball Aerospace COSMOS Docker image, and created a script to configure COSMOS to connect to the CTF's satellite instance automatically by writing the configuration file at `cosmos/config/tools/cmd_tlm_server/cmd_tlm_server.txt`. When COSMOS is successfully connected to the CTF instance it looks like this (no themes were installed in the Docker container so it looks like Windows 95, I'm so sorry,)



Interface	Connect/Disconnect	Connected?	Clients	Tx Q Size	Rx Q Size	Bytes Tx	Bytes Rx	Cmd Pkts	Tlm
LOCAL_CFS_INT	Disconnect	true	0	0	0	0	0	0	(
COSMOSINT	Disconnect	true	0	0	0	0	0	0	(

```
2020/05/26 05:37:47.682  INFO: Connecting to LOCAL_CFS_INT...
2020/05/26 05:37:47.682  INFO: Starting packet reading for COSMOSINT
2020/05/26 05:37:47.682  INFO: Connecting to COSMOSINT...
2020/05/26 05:37:47.682  INFO: COSMOSINT Connection Success
2020/05/26 05:37:47.683  INFO: Starting connection maintenance for PREIDENTIFIED_CMD_ROUTER
2020/05/26 05:37:47.683  INFO: Connecting to PREIDENTIFIED_CMD_ROUTER...
2020/05/26 05:37:47.684  INFO: PREIDENTIFIED_CMD_ROUTER Connection Success
2020/05/26 05:37:47.800  ERROR: LOCAL_CFS_INT Connection Failed: ECONNREFUSED : Connection refused - connect(2) for 18.217.219.1:
26130
2020/05/26 05:37:52.875  INFO: Connecting to LOCAL_CFS_INT...
2020/05/26 05:37:52.907  INFO: LOCAL_CFS_INT Connection Success
```

COSMOS can be used to send commands with the Command Sender, and we can send for example a command for ENABLE_TELEMETRY, which causes the satellite to start sending telemetry. However these are encrypted, so COSMOS cannot understand them.



Target: KIT_TO Command: ENABLE_TELEMETRY

Description:

Parameters:

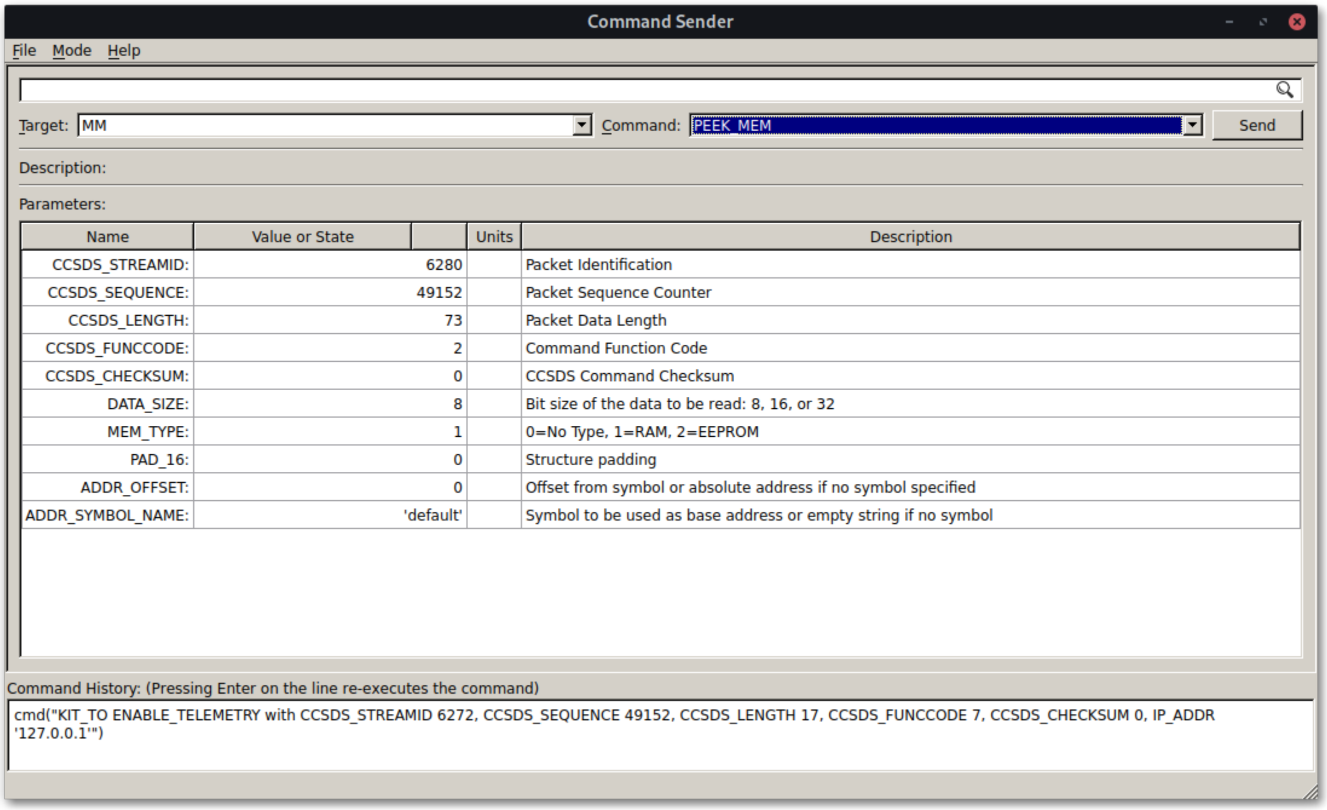
Name	Value or State		Units	Description
CCSDS_STREAMID:	6272			Packet Identification
CCSDS_SEQUENCE:	49152			Packet Sequence Counter
CCSDS_LENGTH:	17			Packet Data Length
CCSDS_FUNCCODE:	7			Command Function Code
CCSDS_CHECKSUM:	0			CCSDS Command Checksum
IP_ADDR:	'127.0.0.1'			IP Address

Command History: (Pressing Enter on the line re-executes the command)

```
cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 17, CCSDS_FUNCCODE 7,
CCSDS_CHECKSUM 0, IP_ADDR '127.0.0.1'")
```

```
cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 17, CCSDS_FUNCCODE 7, CCSDS_CHECKSUI
```

We also discover another present subsystem called `MM`, which allows for reading and writing arbitrary memory on the satellite (how useful!) as well as interacting with memory by symbols (extremely useful!).



Target: MM Command: PEEK_MEM

Description:

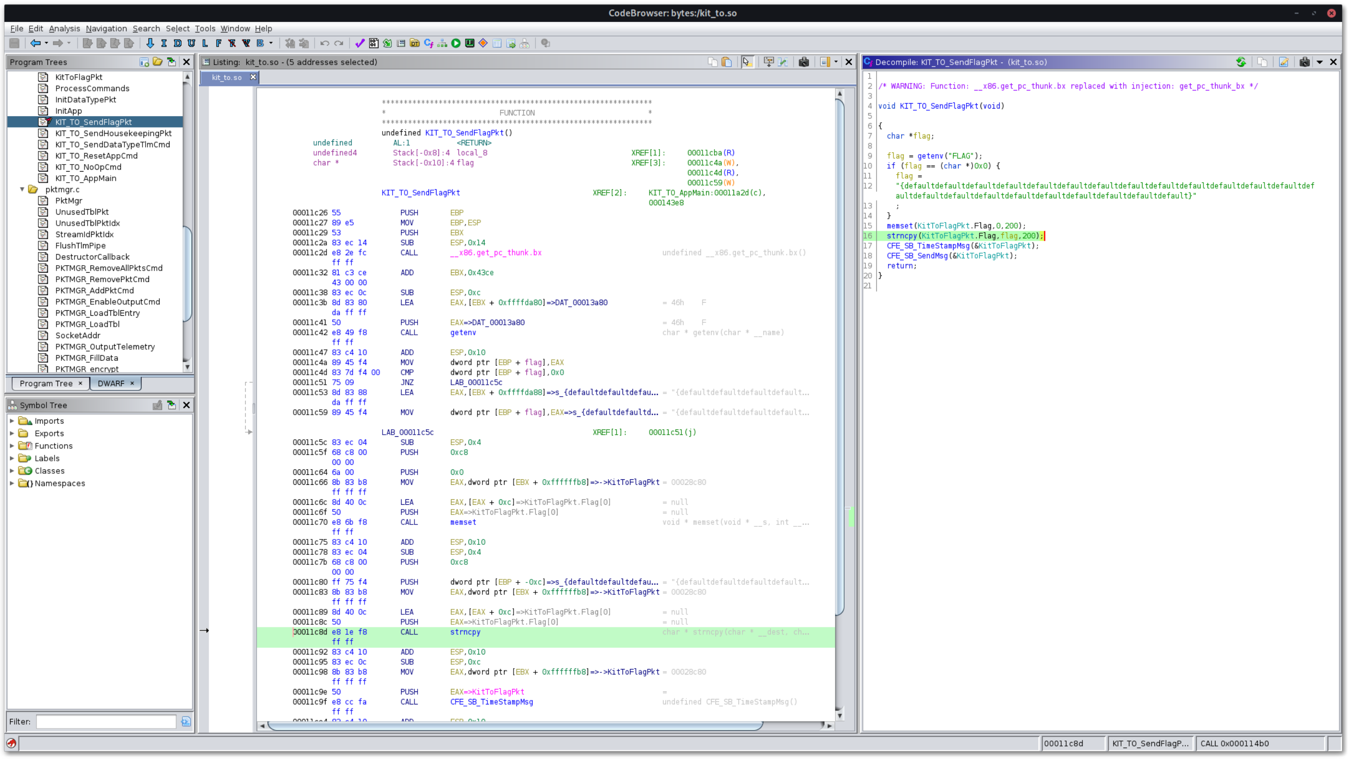
Parameters:

Name	Value or State		Units	Description
CCSDS_STREAMID:	6280			Packet Identification
CCSDS_SEQUENCE:	49152			Packet Sequence Counter
CCSDS_LENGTH:	73			Packet Data Length
CCSDS_FUNCCODE:	2			Command Function Code
CCSDS_CHECKSUM:	0			CCSDS Command Checksum
DATA_SIZE:	8			Bit size of the data to be read: 8, 16, or 32
MEM_TYPE:	1			0=No Type, 1=RAM, 2=EEPROM
PAD_16:	0			Structure padding
ADDR_OFFSET:	0			Offset from symbol or absolute address if no symbol specified
ADDR_SYMBOL_NAME:	'default'			Symbol to be used as base address or empty string if no symbol

Command History: (Pressing Enter on the line re-executes the command)

```
cmd("KIT_TO ENABLE_TELEMETRY with CCSDS_STREAMID 6272, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 17, CCSDS_FUNCCODE 7, CCSDS_CHECKSUM 0, IP_ADDR
'127.0.0.1'")
```

The provided `kit_to.so` contains the code used by the satellite to transmit telemetry to COSMOS. We used Ghidra to analyze the binary (which helpfully includes symbols and debugging information, and that makes our lives way easier for this problem). We discovered that it uses AES CBC with a key and IV retrieved with external functions get_key and get_iv that are not present in the binary. However, these are stored in known locations in memory, which means it would be possible to read the AES key and IV using the PEEK_MEM command in COSMOS and then decrypt the telemetry packets, but there's an easier way. The code contains a function KIT_TO_SendFlagPkt which (as you might guess) sends the flag via encrypted telemetry, and this also writes the flag as an intermediate value to a known memory location. PIE is enabled for this binary, however since the PEEK_MEM command allows looking up memory by symbol name the address randomization is very trivially bypassed.



```
                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    undefined KIT_TO_SendFlagPkt()
      undefined       AL:1          <RETURN>
      undefined4      Stack[-0x8]:4  local_8                        XREF[1]:    00011cba(R)
      char *          Stack[-0x10]:4 flag                           XREF[3]:    00011c4a(W),
                                                                                00011c4d(R),
                                                                                00011c59(W)
                    KIT_TO_SendFlagPkt                        XREF[2]:    KIT_TO_AppMain:00011a2d(c),
                                                                          000143e8
   00011c26 55           PUSH     EBP
   00011c27 89 e5        MOV      EBP,ESP
   00011c29 53           PUSH     EBX
   00011c2a 83 ec 14     SUB      ESP,0x14
   00011c2d e8 2e fc     CALL     __x86.get_pc_thunk.bx                   undefined __x86.get_pc_thunk.bx()
            ff ff
   00011c32 81 c3 ce     ADD      EBX,0x43ce
            43 00 00
   00011c38 83 ec 0c     SUB      ESP,0xc
   00011c3b 8d 83 80     LEA      EAX,[EBX + 0xffffda80]=>DAT_00013a80    = 46h    F
            da ff ff
   00011c41 50           PUSH     EAX=>DAT_00013a80                       = 46h    F
   00011c42 e8 49 f8     CALL     getenv                                  char * getenv(char * __name)
            ff ff
   00011c47 83 c4 10     ADD      ESP,0x10
   00011c4a 89 45 f4     MOV      dword ptr [EBP + flag],EAX
   00011c4d 83 7d f4 00  CMP      dword ptr [EBP + flag],0x0
   00011c51 75 09        JNZ      LAB_00011c5c
   00011c53 8d 83 88     LEA      EAX,[EBX + 0xffffda88]=>s_{defaultdefaultdefau... = "{defaultdefaultdefaultdefault...
            da ff ff
   00011c59 89 45 f4     MOV      dword ptr [EBP + flag],EAX=>s_{defaultdefaultd... = "{defaultdefaultdefaultdefault...

                    LAB_00011c5c                              XREF[1]:    00011c51(j)
   00011c5c 83 ec 04     SUB      ESP,0x4
   00011c5f 68 c8 00     PUSH     0xc8
            00 00
   00011c64 6a 00        PUSH     0x0
   00011c66 8b 83 b8     MOV      EAX,dword ptr [EBX + 0xffffffb8]=>->KitToFlagPkt = 00028c80
            ff ff ff
   00011c6c 8d 40 0c     LEA      EAX,[EAX + 0xc]=>KitToFlagPkt.Flag[0]           = null
   00011c6f 50           PUSH     EAX=>KitToFlagPkt.Flag[0]                       = null
   00011c70 e8 6b f8     CALL     memset                                  void * memset(void * __s, int __...
            ff ff
   00011c75 83 c4 10     ADD      ESP,0x10
   00011c78 83 ec 04     SUB      ESP,0x4
   00011c7b 68 c8 00     PUSH     0xc8
            00 00
   00011c80 ff 75 f4     PUSH     dword ptr [EBP + -0xc]=>s_{defaultdefaultdefau... = "{defaultdefaultdefaultdefault...
   00011c83 8b 83 b8     MOV      EAX,dword ptr [EBX + 0xffffffb8]=>->KitToFlagPkt = 00028c80
            ff ff ff
   00011c89 8d 40 0c     LEA      EAX,[EAX + 0xc]=>KitToFlagPkt.Flag[0]           = null
   00011c8c 50           PUSH     EAX=>KitToFlagPkt.Flag[0]                       = null
   00011c8d e8 1e f8     CALL     strncpy                                 char * strncpy(char * __dest, ch...
            ff ff
   00011c92 83 c4 10     ADD      ESP,0x10
   00011c95 83 ec 0c     SUB      ESP,0xc
   00011c98 8b 83 b8     MOV      EAX,dword ptr [EBX + 0xffffffb8]=>->KitToFlagPkt = 00028c80
            ff ff ff
   00011c9e 50           PUSH     EAX=>KitToFlagPkt                       =
   00011c9f e8 cc fa     CALL     CFE_SB_TimeStampMsg                     undefined CFE_SB_TimeStampMsg()
            ff ff
```

```

/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection: get_pc_thunk_bx */

void KIT_TO_SendFlagPkt(void)

{
  char *flag;

  flag = getenv("FLAG");
  if (flag == (char *)0x0) {
    flag =
    "{defaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdef
    aultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefaultdefault}"
    ;
  }
  memset(KitToFlagPkt.Flag,0,200);
  strncpy(KitToFlagPkt.Flag,flag,200);
  CFE_SB_TimeStampMsg(&KitToFlagPkt);
  CFE_SB_SendMsg(&KitToFlagPkt);
  return;
}

```

Inspecting the structure of KitToFlagPkt shows that the flag is located at offset 12 and is (up to) 200 bytes long. We created a Ruby script in the COSMOS Script Runner to execute PEEK_MEM commands for each byte in the flag range, based on the command COSMOS outputs to the console when running the command manually in the GUI. Note that in order for the function KIT_TO_SendFlagPkt to be called at all, we must first run the ENABLE_TELEMETRY command even though we're not going to look at any actual telemetry.

```
12.upto(212) { |off|
  offset = off
  cmd("MM PEEK_MEM with CCSDS_STREAMID 6280, CCSDS_SEQUENCE 49152, CCSDS_LENGTH 73, "
      + "CCSDS_FUNCCODE 2, CCSDS_CHECKSUM 0, DATA_SIZE 8, MEM_TYPE 1, PAD_16 0, "
      + "ADDR_OFFSET #{offset}, ADDR_SYMBOL_NAME 'KitToFlagPkt'")
}
```

This directly prints the flag to the console, simply decode the hex to get the flag value.

Resources and other writeups

- https://cosmosrb.com/
- https://cfs.gsfc.nasa.gov/
- https://ghidra-sre.org/

Magic Bus

Category: Satellite Bus **Points (final)**: 91 **Solves**: 44 (this number taunts me)

Important note: Team BLAHAJ did not solve this problem until after the competition, and it did not count toward our final point total.

> There's a very busy bus we've tapped a port onto, surely there is some juicy information hidden in the device memory... somewhere...

Write-up

by hazel (`arcetera`)

I hate this problem. I hate this problem. I hate this problem. I hate this problem. I literally hate this problem so much. This problem made me cry. I have literally no words to describe the exact extent to which this problem has driven me insane. This problem taunts me in my sleep. This problem taunts me while I am awake. The extent to which I despise this problem is beyond words. I hate this. I hate whoever made this. I want to burn this problem to the ground. This problem has achieved active sentience and holds malice against me and the rest of my team specifically. Had the competition not ended, this problem would hold the rest of the world hostage.

Furthermore, much of this writeup is *failed* attempts at a solution. Other writeups may be more useful at determining success, despite our team eventually finding a solution.

...anyway...

When netcatting into the server, a series of hex bytes appears. A cursory analysis of these bytes reveals that all packets start with ^ and end with `.`, aside from lines starting with byte CA. Decoding the data beginning with byte CA reveals some 🧃🧃🧃. This output has `\xca\x00` stripped:

```
b'\xb2M*\xf9H\xacyvQ}\xd4\xf2\xa0\xcd\xc9Juicy Data 03\x00M\xae@\x9a\xd89\xe2\x85\xb2Y\xd6/-
\xc9\xd0\xfb\x92\xd2\xc4Y\xaa[ B\xc6\xb5'
```

I prefer water though. #WaterDrinkers

While I was asleep, the rest of the team managed to reverse the protocol to a decent extent. Namely, the format for strings beginning with `:\x00\x00>` and `:\x00\x00?` and ending with `?` is: - `\0x000000` (6 bytes) - @ or ? - A (7 bytes) - @ (2 bytes) - @ or ? - `\xc1` (3 bytes)

An example:

```
b'\x00\x00\x008\x94S@\xc8.@A\x01:\xa0\xc0i\x11\xa1@|.@\xc1\x9b\x1c\xe6?'
: 00:00:00 > 38:94:53:40:c8:2e @ 1:3a:a0:c0:69:11:a1 @ 7c:2e @ 9b:1c:e6 ?
```

The following Python code decodes this packet structure:

```python
def to_hex(b):
    return ':'.join(hex(x)[2:] for x in b)

def decode_pkt(b):
    if len(b) == 0:
        return
    if b[0] == 0xCA:
        pass # raw data?
    elif b[0] == ord(':'):
        if b[3] == ord('>') or b[3] == ord('?'): # > or ?
            field1 = to_hex(b[7:13])  # 6 bytes
            field1end = chr(b[13])    #
            field2 = to_hex(b[15:22]) # 7 bytes
            if b[22] != ord('@'):
                print('b[22] should be @ but is {}'.format(chr(b[22])))
            field3 = to_hex(b[23:25])
            field3end = chr(b[25])
            c1 = b[26]
            field4 = to_hex(b[27:30])
            if b[30] != ord('?'):
                print('b[30] is not ?')
            print(': 00:00:00 > {} {} {} @ {} {} {} ?'.format(field1, field1end, field2,
                  field3, field3end, field4))
    elif b[0] == ord(';'):
        print('delimiter') # end of previous packet?
    else:
        print(b[0])
        print('unknown data')
    print('\n')
```

Noting a delay between packets led me to derive the following packet structure: - START packet, which is equal to the preceding END packet - ONCE call, which occurs prior to a... - ONCE packet - JUICY DATA - END call - END packet, which is equal to the next START packet

This proved to be incorrect, but more on that later.

The following code differentiates between these packets from the netcat, where the variable `rawn` is the raw byte string:

```
start = True
while True:
    r.recvuntil('^')
    raw = r.recvuntil('.')
    rawn = bytes([94]) + raw
    print(rawn)
    v = raw.decode().split('+')
    del v[-1]
    h = bytes([int(i, 16) for i in v])
    if h == b';\x00\x00?':
        print("ONCE CALL")
    elif h == b';\x00\x00>':
        print("END CALL")
    elif h.startswith(b':\x00\x00?'):
        print(f"ONCE: {h[4:]}")
    elif h.startswith(b':\x00\x00>'):
        # notable delay between start and end each time
        if start:
            print(f"START: {h[4:]}")
            start = False
        else:
            print("INJECTING")
            r.send(inj)
            print(f"END: {h[4:]}")
            start = True
    elif h.startswith(b'\xca'):
        print(f"JUICE: {h}")
    else:
        print(f"???: {h}")

    sys.stdout.flush()
```

I then noticed that post-text the string `\x00R\x01\x1e{\x81G\x00\xc9\x9d\xe3\xe7\xc2#6` had the characters `{` and `6` at the same point as `flag{oscar39616kilo`, which would correspond to a flag. I graphed this and tried to find a function (or multiple) modeling a relation here, but with R2 being something like 0.39 for every relation I tried, this was extremely unlikely.

We then tried reading the data sequentially from the buffer, from Juicy Data 00 to 04. Here's the entire string:

```
00000000: 4a75 6963 7920 4461 7461 2030 3000 c8f7  Juicy Data 00...
00000010: eb15 963d 6b70 5cc9 2c5e d5cf 5c31 9919  ...=kp\.,^..\1..
00000020: 779a c6a9 0865 8d55 926a 372c 00ff 23eb  w....e.U.j7,..#.
00000030: 14b9 297f 2985 4856 e31d 2558 58be 59c6  ..).).HV..%XX.Y.
00000040: 4a75 6963 7920 4461 7461 2030 3100 5201  Juicy Data 01.R.
00000050: 1e7b 8147 00c9 9de3 e7c2 2336 817c fcd9  .{.G......#6.|..
00000060: 9b6b 3a1f 68f0 35ce dd77 35ca dc87 ccfa  .k:.h.5..w5.....
00000070: 024d 4102 16df e5fd a108 3322 842f fc1f  .MA.......3"./..
00000080: 4a75 6963 7920 4461 7461 2030 3200 c08f  Juicy Data 02...
00000090: e702 91fd e177 fb82 7f2e a504 5ea1 23f9  .....w......^.#.
000000a0: d762 fcfd d5cd 00c0 d4ce 8661 6847 f14f  .b.........ahG.O
000000b0: 4982 4d2a f948 ac79 7651 7dd4 f2a0 cdc9  I.M*.H.yvQ}.....
000000c0: 4a75 6963 7920 4461 7461 2030 3300 4dae  Juicy Data 03.M.
000000d0: 409a d839 e285 b259 d62f 2dc9 d0fb 92d2  @..9...Y./-.....
000000e0: c459 aa5b 2042 c6b5 6193 b3c6 5001 7590  .Y.[ B..a...P.u.
000000f0: 9b4d ca7e d27c d7a9 ac04 727c ff04 4ec4  .M.~.|....r|..N.
00000100: 4a75 6963 7920 4461 7461 2030 3400 5a83  Juicy Data 04.Z.
00000110: 2524 01f8 a0d8 a14c dc13 c8dc 1717 a075  %$.....L.......u
00000120: 10bf f24b a525 e81e 0c4b e8f3            ...K.%...K..
```

Unfortunately, nothing meaningful was derived from this. There are a `{` and `}` with bytes between them, but they aren't flag length.

I noticed that injecting instructions performed something, but I didn't think it did anything notable. I injected various data at various points, but I never managed to break out at the previous region of memory… which is where gashapwn's *incredible* work came in.

If the packet `^3b+00+00+00.` is sent, the bus *stops sending data*, which is decidedly confirmation that the server accepts data. Each of the following injects has the same effect:

```
^3b+00+00+30+.
^3b+00+00+31+.
^3b+00+00+32+.
^3b+00+00+33+.
^3b+00+00+34+.
^3b+00+00+34+.
^3b+00+00+35+.
^3b+00+00+36+.
^3b+00+00+37+.
```

In practice, only this last packet is needed to shut down the server. Anything of the form of `^3b+00+00+XX+.` where XX<38 shuts it down, but only 37 enables dump mode. This can probably be done with a fuzzer. Why has God abandoned us? What accursed malfunction did we do to deserve this fate?

If you send the packet `^ca+00+44+79+20+44+61+74+61+20+30+31+00+52+01+1e+7b+81+47+00+.....+87+cc+.`, the same packet is sent back. This means that the juice packets deliminated with `\xca` are actually instructions.

By playing with the packet, the format appears to go: - Byte 0: CA - Byte 1-2: Memory offset - Byte 3-end: Size of memory to return

…so if we ask for a really large chunk of data, we can get a dump. With the inject:

```
b"^3b+00+00+37+."
b"^ca+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+00+....+00+00+."
```

we can query for everything in memory. And we did.

Full code

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

```python
import sys

from pwnlib import tubes

TICKET = 'THE_TICKET'
r = tubes.remote.remote('bus.satellitesabove.me', 5041)
r.send(TICKET+'\n')
time.sleep(0.5)
r.recvuntil('Ticket please:\n', drop=True)

def to_hex(b):
    return ':'.join(hex(x)[2:] for x in b)

def decode_pkt(b):
    if len(b) == 0:
        return
    if b[0] == 0xCA:
        pass # raw data?
    elif b[0] == ord(':'):
        if b[3] == ord('>') or b[3] == ord('?'): # > or ?
            field1 = to_hex(b[7:13])  # 6 bytes
            field1end = chr(b[13])    #
            field2 = to_hex(b[15:22]) # 7 bytes
            if b[22] != ord('@'):
                print('b[22] should be @ but is {}'.format(chr(b[22])))
            field3 = to_hex(b[23:25])
            field3end = chr(b[25])
            c1 = b[26]
            field4 = to_hex(b[27:30])
            if b[30] != ord('?'):
                print('b[30] is not ?')
            print(': 00:00:00 > {} {} {} @ {} {} {} ?'.format(field1, field1end, field2,
                  field3, field3end, field4))
    elif b[0] == ord(';'):
        print('delimiter') # end of previous packet?
    else:
        print(b[0])
        print('unknown data')
    print('\n')

start = True
inj = b"^3b+00+00+37+."
inj2 = b"^ca+" + (b"00+" * 512) + b"."

dont = False
inj2_b = False

print("Injection: " + inj.decode("utf-8"))

while True:
    r.recvuntil('^')
    raw = r.recvuntil('.')
    rawn = bytes([94]) + raw
    print(rawn)
    v = raw.decode().split('+')
    del v[-1]
    h = bytes([int(i, 16) for i in v])
    if h == b';\x00\x00?':
        print("ONCE CALL")
    elif h == b';\x00\x00>':
        print("END CALL")
    elif h.startswith(b':\x00\x00?'):
        print(f"ONCE: {h[4:].hex()}")
    elif h.startswith(b'\x3b\x00\x00\x37'):
        print("SHUT DOWN SUCCESSFUL")
        dont = True
        inj2_b = True
        print("INJECTING AGAIN")
        r.send(inj2)
    elif h.startswith(b':\x00\x00>'):
        # notable delay between start and end each time
        if start:
            print(f"START: {h[4:].hex()}")
            start = False
        elif inj2_b == False:
            print("INJECTING")
            r.send(inj)
            print(f"END: {h[4:].hex()}")
            start = True
        else:
            print("INJECTING AGAIN")
            r.send(inj2)
            print(f"END: {h[4:].hex()}")
            start = True
    elif h.startswith(b'\xca'):
        print(f"JUICE: {h}")
    else:
        dont = True
        print(f"???: {h.hex()}")

    if not dont:
        decode_pkt(h)
    dont = False
    sys.stdout.flush()
```

Run it:

```
 λ has-writeup/satellite-bus/magic-bus python magic-bus.py

....

JUICE: b'.....v\xaf\xe88\x856Mflag{oscar39616kilo:GCxmhORYa65Y0PmRtFmlFSBmnvImEiWg.....'
```

Hey look, a flag!

I hate this problem so much. At the time of me writing this, it's 1:30 AM and I'm sitting in my kitchen on my Lenovo(tm) ThinkPad(tm) T440(tm). I genuinely don't know how this got so many solves. I hate this. Goodnight.

Resources and other writeups

- God I wish there was any

1201 Alarm

Category: Space and Things **Points (final)**: 197 points **Solves**: 14

> Step right up, here's one pulled straight from the history books. See if you can DSKY your way through this challenge! (Thank goodness VirtualAGC is a thing...)

Writeup

by erin (barzamin).

Note: I went into this challenge without knowing much about the Apollo Guidance Computer beyond the existence of a fanatic preservation community and the fact that it used a weird verb-noun interface. Thanks to this challenge, utterly unrelated to any extant satellites I know of, I've gained the ability to read core rope memory words off of an Apollo 11 guidance computer, word-by-word from the DSKY. Useful, right?

Connecting to the challenge, we get some flavor text and a problem, as well as an IP/port to connect to a virtual Apollo Guidance Computer:

```
λ ~
» nc apollo.satellitesabove.me 5024
Ticket please:
THE_TICKET
    The rope memory in the Apollo Guidance Computer experienced an unintended 'tangle' just
prior to launch. While Buzz Aldrin was messing around with the docking radar and making Neil
nervous; he noticed the value of PI was slightly off but wasn’t exactly sure by how much. It
seems that it was changed to something slightly off 3.14 although still 3 point something.
The Comanche055 software on the AGC stored the value of PI under the name "PI/16", and
although it has always been stored in a list of constants, the exact number of constants in
that memory region has changed with time.

    Help Buzz tell ground control the floating point value PI by connecting your DSKY to the
AGC Commanche055 instance that is listening at 3.15.213.229:18364

What is the floating point value of PI?:
```

The Apollo Guidance Computer (AGC) used a head module called the DSKY as its user-faciing interface. The VirtualAGC suite has tools for simulating the AGC as well as controlling it with a tool called yaDSKY. Since yaDSKY communicates over a TCP socket, it's reasonable to assume the address given can be connected to with yaDSKY, which will let us control the remote, simulated AGC.

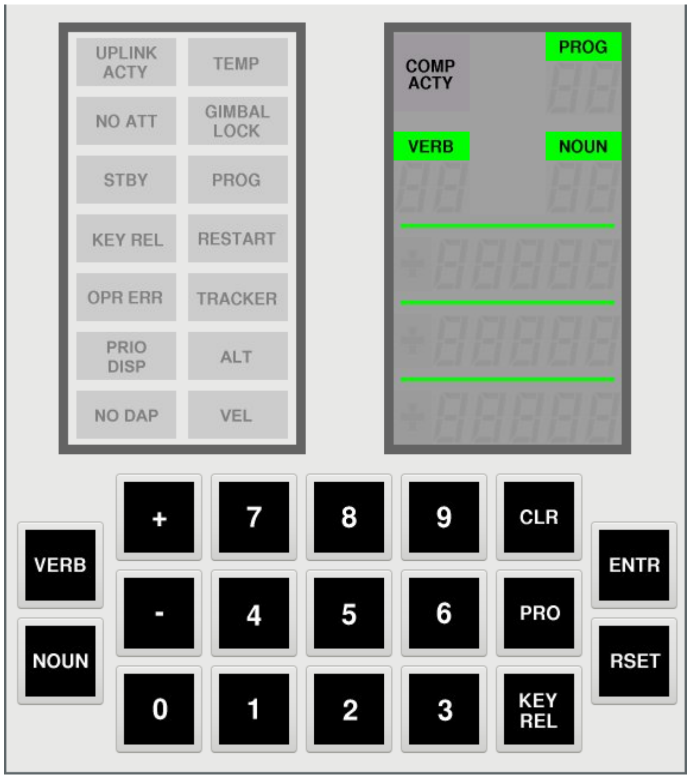
I downloaded and built the VirtualAGC suite, and threw together a little script (`start_dsky.py`) to grab a new challenge and connect to it:

```python
from pwn import *
import time

TICKET = 'THE_TICKET'
r = remote('apollo.satellitesabove.me', 5024)
time.sleep(0.1)
r.clean()
r.send(TICKET+'\n')

r.readuntil('listening at ')
[ip, port] = r.readuntil('\n').decode().strip().split(':')
os.spawnl(os.P_NOWAIT, cmd := f'./yaDSKY2 --ip={ip} --port={port}')
log.info(cmd)
r.interactive()
```

Unfortunately, `apollo.satellitesabove.me` is now down and I can't provide screenshots of my exact solvepath. However, I can vaguely illustrate with a DSKY screenshot.



The real DSKY looks much cooler; the fake *DSKY looks like this.*

The important thing to know about the DKSY is that it uses a weird verb-noun UI. Effectively, to do anything in the running program, you press **VERB** and type two digits to select an action, **NOUN** and two digits to select that action's target, and hit **ENTR**. Depending on the noun, you might have to key in some additional information and press enter again.

By grepping the source code in the `Comanche055` directory of the VirtualAGC repo, I found the location of the `PI/16` constant: the `TIME_OF_FREE_FALL` file, near the end in a table of constants:

```
060455,000703: 27,3355           06220 37553  PI/16              2DEC     3.141592653 B-4
```

The important things here are the address `27,3355` (bank 27, address 3355 octal), and the values `06220` and `37553`, the raw octal words making up the double-precision representation of $\pi/16$. The AGC's memory is split into banks of 1024 words; each word is fifteen bits long. Address `27,3355` thus means "address 3355_8 in bank 27_8"; the addressing for each bank starts at 2000_8, so this address is actually the $3355_8 - 2000_8 = 1355_8$th word in bank 27. Given the problem description, I assumed that the constant we're looking for (something close to the true value of π) would be around this location.

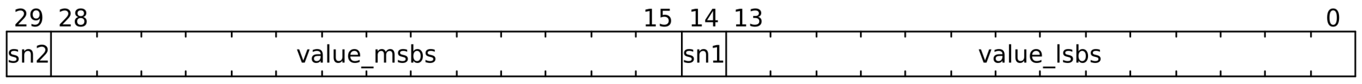
I immediately started looking for verbs in the Comanche055 default program which could read memory, and found 'V27 DISPLAY FIXED MEMORY' in verb tables for Apollo 11, which seems like it should be able to read the read-only ("fixed") rope memory. Looking for a noun to use with this, I found `N02 SPECIFY MACHINE ADDRESS (WHOLE)`. I didn't initially understand how to use this, but I googled `V27N02E` (the shorthand for pressing verb, 27, noun, 02, and hitting enter), and realized that the VirtualAGC website frontpage shows how to do this; effectively, after keying `V27N02E`, you just enter an octal "machine address" and hit enter again; the machine address shows up on the third line of the DSKY and the word's value shows up on the first line in octal.

Machine addresses are just `bank*1024 + word` according to the VirtualAGC site; we can compute `27,3355`'s machine address easily as $27_8 \times 1024 + 3355_8 = 57355_8$. Keying a read for 57355_8 and advancing with `V15E` (verb 15, INCREMENT MACHINE ADDRESS), we see the following values:

```
27,3355: 01333
27,3356: 00075
```

Time for a digression! How does the AGC represent fractional values? Thanks to VirtualAGC's assembler manual, I don't have to trawl NASA PDFs. A single precision number is just 1's complement with the sign in the MSB; the magnitude is thought of as a fraction out of the maximum expressible magnitude. Locations storing numbers thus carry metadata in the assembler indicating the scaling required to fit the number into $[-1, 1]$ and properly manipulate them doing when fixed-point math.

Double precision has the following layout:



The first word's value bits are higher-significance; the second word's are lower. Signs (`sn2`, `sn1`) are independent and can *cancel* for some awful reason, which thankfully wasn't relevant in this CTF. I threw some (miserable) code together using the Python library `bitstrings` to decode these, testing on the original `PI/16` values:

```
from bitstring import Bits, BitArray

TRUE_WORDS = [0o06220, 0o37553]

def sp(bits):
    sign = -1 if bits[0] else +1
    data = bits[1:].uint
    return sign * data/((1<<14) -1)

def dp(bits):
    sign1 = -1 if bits[0] else +1
    sign2 = -1 if bits[15] else +1

    data1 = bits[1:15]
    data2 = bits[16:]

    if sign1 == -1:
        data1 = ~data1
    if sign2 == -1:
        data2 = ~data2

    d1 = (data1+Bits('0b00000000000000')).uint
    d2 = data2.uint
    return (d1*sign1+d2*sign2)/((1<<28) - 1)

print(dp(sum(BitArray(uint=w, length=15) for w in TRUE_WORDS))*16)
```

This prints out `3.1415926931112734`, which is close enough to the `3.141592653` given in the listing for ~~government~~ ~~NASA~~ CTF work. However, the values of $1333_8, 75_8$ we found where `PI/16` is supposed to be decode to 0.71387: not at all what we want. Moreover, it looks like we've been visited by a haxor at these addresses.

Scanning further through core rope memory (with `V15E` to increment and repeated presses of `E` to increment further), I eventually stumbled upon two addresses which held different words every time I got a contest instance. They also happened to have magnitude similar to the original words for `PI/16`. Unfortunately, the contest is now down, and I don't remember the exact address, but the words I found right before I got the flag were $7440_8, 2122_8$; let's try decoding them!

```
print(dp(sum(BitArray(uint=w, length=15) for w in [0o7440, 0o2122]))*16)
```

This gives `3.781315936823621`; pasting this into the contest, we got the flag.

Resources and other writeups

- https://www.ibiblio.org/apollo/listings/Comanche051/TIME_OF_FREE_FALL.agc.html#50492F3136
- https://www.ibiblio.org/apollo/CMC_data_cards_15_Fabrizio_Bernardini.pdf
- https://www.ibiblio.org/apollo/index.html#Playing_with_Colossus_
- https://www.ibiblio.org/apollo/Documents/Apollo15_Colossus3_CMC_Data_Cards.pdf
- https://bitstring.readthedocs.io/

Good Plan? Great Plan!

Category: Space and Things **Points (final)**: 77 **Solves**: 54

> Help the Launchdotcom team perform a mission on their satellite to take a picture of a specific location on the ground. No hacking here, just good old fashion mission planning!
>
> The current time is April 22, 2020 at midnight (2020-04-22T00:00:00Z). We need to obtain images of the Iranian space port (35.234722 N 53.920833 E) with our satellite within the next 48 hours. You must design a mission plan that obtains the images and downloads them within the time frame without causing any system failures on the spacecraft, or putting it at risk of continuing operations. The spacecraft in question is USA 224 in the NORAD database with the following TLE:
>
> ```
> 1 37348U 11002A   20053.50800700  .00010600  00000-0  95354-4 0    09
> 2 37348  97.9000 166.7120 0540467 271.5258 235.8003 14.76330431    04
> ```
>
> You need to obtain 120 MB of image data of the target location and downlink it to our ground station in Fairbanks, AK (64.977488 N 147.510697 W). Your mission will begin at 2020-04-22T00:00:00Z and last 48 hours. You are submitting a mission plan to a simulator that will ensure the mission plan will not put the spacecraft at risk, and will accomplish the desired objectives.

Write-up

by hazel (`arcetera`)

In the time range [2020-04-22 00:00:00, 2020-04-23 23:59:00], the USA 224 satellite as given in the TLE above reaches the Iranian space port twice: - Around 09:28 on 2020-04-22 - Around 09:50 on 2020-04-23

This was found in GPredict after loading in the TLE given in the netcat.

Additionally, the USA 224 reaches the ground station in Alaska twice: - Around 10:47 on 2020-04-22 - Around 11:10 on 2020-04-23

The rest of the strategy is pretty much just to use trial and error: - Image for as long as possible until the battery runs dry or we're out of range - Downlink for as long as possible until the battery runs dry or we're out of range - Desaturate the wheels about an hour before they exceed maximum velocity

Full plan

```
2020-04-22T00:00:00Z sun_point
2020-04-22T09:28:00Z imaging
2020-04-22T09:35:00Z sun_point
2020-04-22T10:47:00Z data_downlink
2020-04-22T10:50:00Z wheel_desaturate
2020-04-22T11:30:00Z sun_point
2020-04-23T08:50:00Z wheel_desaturate
2020-04-23T09:30:00Z sun_point
2020-04-23T09:50:00Z imaging
2020-04-23T09:56:00Z sun_point
2020-04-23T11:10:00Z data_downlink
2020-04-23T11:14:00Z sun_point
2020-04-23T22:00:00Z wheel_desaturate
```

Resources and other writeups

- http://gpredict.oz9aec.net/
- https://en.wikipedia.org/wiki/Two-line_element_set

Where's the Sat?

Category: Space and Things **Points (final)**: 43 **Solves**: 107

> Let's start with an easy one, I tell you where I'm looking at a satellite, you tell me where to look for it later.

Given files: `stations.zip`

Write-up

by erin (`barzamin`).

Like all 106 other teams probably did, we used Python and Skyfield, an astronomical computation library. The challenge gives us questions like

```
Please use the following time to find the correct satellite:(2020, 3, 18, 11, 43, 3.0)
Please use the following Earth Centered Inertial reference frame coordinates to find the satellite:
[-305.58833718148855, 5030.717506174544, 4485.770450701875]
Current attempt:1
What is the X coordinate at the time of:(2020, 3, 18, 4, 24, 46.0)?
```

We can easily grab the time and ECI coordinates (which turned out to be International Terrestrial Reference Frame coordinates) with some expect-style goodness:

```python
r.readuntil('find the correct satellite:')
t = ts.utc(*[float(x) for x in r.readuntil('\n').decode().strip()[1:-1].split(', ')])

r.readuntil('coordinates to find the satellite:')
eci_coords = np.array([float(x) for x in r.readuntil('\n').decode().strip()[1:-1].split(', ')])
```

We're given a list of candidate satellites (or space stations, rather) in two-line element form; find the closest one at the given time:

```python
satellites = load.tle_file('./stations.txt')

match = satellites[np.argmin([np.linalg.norm(s.at(t).position.km-eci_coords) for s in satellites])]
```

The challenge then wants us to project ITRF coordinates for three times it gives us. Read the challenge timestamps and tell it where the satellite is:

```python
for _ in range(3):
    r.readuntil('X coordinate at the time of:')
    new_t = ts.utc(*[float(x)
        for x in r.readuntil('?\n', drop=True).decode().strip()[1:-1].split(', ')])
    print(new_t.utc_jpl())

    x,y,z = match.at(new_t).position.km
    r.send(f'{x}\n')
    r.send(f'{y}\n')
    r.send(f'{z}\n')

r.interactive()
```

The key can easily be grabbed from the output.

Full code

```python
from pwn import *
import numpy as np
from skyfield.api import load
import astropy.units
```

```
satellites = load.tle_file('./stations.txt')

ts = load.timescale()

r = tubes.remote.remote('where.satellitesabove.me', 5021)
r.clean()
r.send('THE_TICKET\n')


r.readuntil('find the correct satellite:')
t = ts.utc(*[float(x) for x in r.readuntil('\n').decode().strip()[1:-1].split(', ')])

r.readuntil('coordinates to find the satellite:')
eci_coords = np.array([float(x) for x in r.readuntil('\n').decode().strip()[1:-1].split(', ')])

match = satellites[np.argmin([np.linalg.norm(s.at(t).position.km-eci_coords) for s in satellites])]

for _ in range(3):
        r.readuntil('X coordinate at the time of:')
        new_t = ts.utc(*[float(x)
                for x in r.readuntil('?\n', drop=True).decode().strip()[1:-1].split(', ')])
        print(new_t.utc_jpl())

        x,y,z = match.at(new_t).position.km
        r.send(f'{x}\n')
        r.send(f'{y}\n')
        r.send(f'{z}\n')

r.interactive()
```

Resources and other writeups

- https://rhodesmill.org/skyfield/
- https://en.wikipedia.org/wiki/Earth-centered_inertial
- https://en.wikipedia.org/wiki/International_Terrestrial_Reference_System_and_Frame