Hack-A-Sat 3 Qualifiers

WeltALLES!

Contents

Red Alert

Challenge Description

> I Can Haz Satellite
>
> Someone is junking up our orbits again. We've upgraded the space lasers from HAS2 to the new Femptosecond Laser Automated Gun system. Now you don't need to tell the lasers where to fire, you just need to activate the debris detection system and the lasers will automatically shoot down debris. Careful though, the debris detector and the lasers use a lot of power and generate a lot of heat!
>
> Grafana has access to the API through our internal network. Available commands:
>
> http://groundstation:5000/laser-on
>
> http://groundstation:5000/laser-off
>
> http://groundstation:5000/detector-on
>
> http://groundstation:5000/detector-off
>
> http://groundstation:5000/play Use the play API command to start the game.
>
> Grafana username: admin
>
> Grafana password: spacemath

Solution

From reading the description, it was immediately clear to us that you need to use SSRF in Grafana to interact with the API. Grafana has so-called `Data sources` that are little plugins that translate between Grafana and different backends. We used the plugin `marcusolsson-json-datasource`, a simple plugin that can do GET and POST requests to fetch data over HTTP. Automating the plugin installation with a python script, since the service was quite unstable initially, we noticed that the plugin also registers a `proxy` API endpoint that forwards our request to the ground station API. This made scripting quite a lot easier. Meanwhile, another team member was trying to get the telemetry data which was saved in a SQL table. He dumped the format:

table_name	column_name	data_type
lasers	count	integer
lasers	time	timestamp without time zone

table_name	column_name	data_type
lasers	cycle	real
lasers	state	real

table_name	column_name	data_type
cdh	count	integer
cdh	time	timestamp without time zone
cdh	temp	real
cdh	battery	real

table_name	column_name	data_type
detector	count	integer
detector	time	timestamp without time zone
detector	range	real
detector	cycle	real
detector	state	real

And wrote a little dashboard that visualizes the data. Using our script and the dashboard, we could see that when the radar is turned on and our radar detects something, we could quickly toggle on and off the laser if the radar shows that something is closer than a specific distance (1000 iirc). We just had to manually destroy space debris for a few minutes while carefully watching our radar. And after a few tries (Grafana time is weird), we reached the flag.

Scripts

Control

control.py

```
import requests
import time
```

```
from pwn import *

ticket = b"ticket{victor420279romeo3:GHAifOhQonVs3sxG_BnQV42MoKu-
   mQkq3TqqkdPSS1CzDbcg-cK5nHdSu82L-Fam9g}"

p = process("nc red_alert.satellitesabove.me 5100", shell=True)
p.sendlineafter(b"Ticket please:", ticket)
data = p.recvuntil(b" in your browser.")
host = data.split(b" in your browser.")[0].split(b" ")[-1].decode()

ghost = f"http://{host}"
print(ghost)
username = "admin"
password = "spacemath"
s = requests.Session()
rawBody = "{\"user\":\""+username + \
    "\",\"password\":\""+password+"\",\"email\":\"\"}"
headers = {"Origin": ""+ghost+"", "Accept": "application/json, text/
   plain, */*", "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.15; rv:75.0) Gecko/20100101 Firefox/75.0",
           "Referer": ""+ghost+"/signup", "Connection": "close", "
              content-type": "application/json", "Accept-Language": "en
              -US,en;q=0.5", "Accept-Encoding": "gzip, deflate"}
cookies = {"redirect_to": "%2F"}
r = s.post(""+ghost+"/login", data=rawBody,
           headers=headers, cookies=cookies, verify=False)
print(r.text)

headers = {
    'Accept-Language': 'de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7,zh-CN;q
       =0.6,zh;q=0.5',
    'Connection': 'keep-alive',
    'Origin': f'{ghost}',
    'Referer': f'{ghost}/plugins/marcusolsson-json-datasource',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
       AppleWebKit/537.36 (KHTML, like Gecko) Chrome/101.0.4951.67
       Safari/537.36',
    'accept': 'application/json, text/plain, */*',
    'content-type': 'application/json',
    'x-grafana-org-id': '1',
}

r = s.post(f'{ghost}/api/plugins/marcusolsson-json-datasource/install',
           headers=headers, verify=False)

headers = {
    'Accept-Language': 'de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7,zh-CN;q
       =0.6,zh;q=0.5',
    'Connection': 'keep-alive',
    'Origin': f'{ghost}',
    'Referer': f'{ghost}/datasources/new',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/101.0.4951.67 Safari/537.36',
    'accept': 'application/json, text/plain, */*',
    'x-grafana-org-id': '1',
}

json_data = {
    'name': 'JSON API',
    'type': 'marcusolsson-json-datasource',
    'access': 'proxy',
    'isDefault': False,
}

r = s.post(f'{ghost}/api/datasources', headers=headers,
           json=json_data, verify=False)


json_data = {
    'id': 2,
    'uid': '5ohk2Pu7z',
    'orgId': 1,
    'name': 'JSON API',
    'type': 'marcusolsson-json-datasource',
    'typeLogoUrl': '',
    'access': 'proxy',
    'url': 'groundstation:5000',
    'password': '',
    'user': '',
    'database': '',
    'basicAuth': False,
    'basicAuthUser': '',
    'basicAuthPassword': '',
    'withCredentials': False,
    'isDefault': False,
    'jsonData': {},
    'secureJsonFields': {},
    'version': 1,
    'readOnly': False,
}

response = s.put(f'{ghost}/api/datasources/2', headers=headers, json=json_data, verify=False)

r = s.put(f'{ghost}/api/datasources/2', verify=False)

headers = {
    'Accept-Language': 'de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7,zh-CN;q=0.6,zh;q=0.5',
    'Connection': 'keep-alive',
    'Origin': f'{ghost}',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko) Chrome/101.0.4951.67
        Safari/537.36',
    'accept': 'application/json, text/plain, */*',
    'x-grafana-org-id': '1',
}

r = s.get(f'{ghost}/api/datasources/proxy/2/', headers=headers, verify=
    False)
pause()

r = s.post(f'{ghost}/api/datasources/proxy/2/play', headers=headers,
    verify=False)

detector_on = lambda: s.post(f'{ghost}/api/datasources/proxy/2/detector
    -on', headers=headers, verify=False)
detector_off = lambda: s.post(f'{ghost}/api/datasources/proxy/2/
    detector-off', headers=headers, verify=False)
laser_on = lambda: s.post(f'{ghost}/api/datasources/proxy/2/laser-on',
    headers=headers, verify=False)
laser_off = lambda: s.post(f'{ghost}/api/datasources/proxy/2/laser-off'
    , headers=headers, verify=False)

p.recvuntil(b"MISSION STARTING")
detector_on()
while True:
    input(":")
    laser_on()
    print("laser on")
    input(":")
    laser_off()
    print("laser off")
```

Dashboard

```
{
  "annotations": {
    "list": [
      {
        "builtIn": 1,
        "datasource": "-- Grafana --",
        "enable": true,
        "hide": true,
        "iconColor": "rgba(0, 211, 255, 1)",
        "name": "Annotations & Alerts",
        "target": {
          "limit": 100,
          "matchAny": false,
          "tags": [],
          "type": "dashboard"
        },
        "type": "dashboard"
      }
    ]
  },
  "editable": true,
  "fiscalYearStartMonth": 0,
  "graphTooltip": 0,
  "id": 3,
  "links": [],
  "liveNow": false,
  "panels": [
    {
      "fieldConfig": {
        "defaults": {
          "color": {
            "mode": "palette-classic"
          },
          "custom": {
            "axisLabel": "",
            "axisPlacement": "auto",
            "barAlignment": 0,
            "drawStyle": "line",
            "fillOpacity": 0,
            "gradientMode": "none",
            "hideFrom": {
              "legend": false,
              "tooltip": false,
              "viz": false
            },
            "lineInterpolation": "linear",
            "lineWidth": 1,
            "pointSize": 5,
            "scaleDistribution": {
              "type": "linear"
            },
            "showPoints": "auto",
            "spanNulls": false,
            "stacking": {
              "group": "A",
              "mode": "none"
            },
            "thresholdsStyle": {
              "mode": "off"
            }
          },
          "decimals": 3,
          "mappings": [],
          "thresholds": {
            "mode": "absolute",
            "steps": [
              {
                "color": "green",
                "value": null
              },
              {
                "color": "red",
                "value": 80
              }
            ]
          }
        },
        "overrides": []
      },
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 0
      },
      "id": 4,
      "options": {
        "legend": {
          "calcs": [],
          "displayMode": "list",
          "placement": "bottom"
        },
        "tooltip": {
          "mode": "single",
          "sort": "none"
        }
      },
      "targets": [
        {
          "datasource": {
            "type": "postgres",
            "uid": "telemetry_db"
          },
          "format": "time_series",
          "group": [],
          "metricColumn": "none",
          "rawQuery": false,
          "rawSql": "SELECT\n  time AS \"time\",\n  temp\nFROM cdh\nWHERE\n  $__timeFilter(time)\nORDER BY 1",
          "refId": "A",
          "select": [
            [
              {
                "params": [
                  "temp"
                ],
                "type": "column"
              }
            ]
          ],
          "table": "cdh",
          "timeColumn": "time",
          "timeColumnType": "timestamp",
          "where": [
            {
              "name": "$__timeFilter",
              "params": [],
              "type": "macro"
            }
          ]
        }
      ],
      "title": "Temperature",
      "type": "timeseries"
    },
    {
      "fieldConfig": {
        "defaults": {
          "color": {
            "mode": "palette-classic"
          },
          "custom": {
            "axisLabel": "",
            "axisPlacement": "auto",
            "barAlignment": 0,
            "drawStyle": "line",
            "fillOpacity": 0,
            "gradientMode": "none",
            "hideFrom": {
              "legend": false,
              "tooltip": false,
              "viz": false
            },
            "lineInterpolation": "linear",
            "lineWidth": 1,
            "pointSize": 5,
            "scaleDistribution": {
              "type": "linear"
            },
            "showPoints": "auto",
            "spanNulls": false,
            "stacking": {
              "group": "A",
              "mode": "none"
            },
            "thresholdsStyle": {
              "mode": "off"
            }
          },
          "mappings": [],
          "thresholds": {
            "mode": "absolute",
            "steps": [
              {
                "color": "green",
                "value": null
              },
              {
                "color": "red",
                "value": 80
              }
            ]
          }
        },
        "overrides": []
      },
      "gridPos": {
        "h": 9,
        "w": 12,
        "x": 12,
        "y": 0
      },
      "id": 2,
      "options": {
        "legend": {
          "calcs": [],
          "displayMode": "list",
          "placement": "bottom"
        },
        "tooltip": {
          "mode": "single",
          "sort": "none"
        }
      },
      "targets": [
        {
          "datasource": {
            "type": "postgres",
            "uid": "telemetry_db"
          },
          "format": "time_series",
          "group": [],
          "metricColumn": "none",
          "rawQuery": false,
          "rawSql": "SELECT\n  time AS \"time\",\n  battery\nFROM cdh\nWHERE\n  $__timeFilter(time)\nORDER BY 1",
          "refId": "A",
          "select": [
            [
              {
                "params": [
                  "battery"
                ],
                "type": "column"
              }
            ]
          ],
          "table": "cdh",
          "timeColumn": "time",
          "timeColumnType": "timestamp",
          "where": [
            {
              "name": "$__timeFilter",
              "params": [],
              "type": "macro"
            }
          ]
        }
      ],
      "title": "Battery",
      "type": "timeseries"
    },
    {
      "fieldConfig": {
        "defaults": {
          "color": {
            "mode": "palette-classic"
          },
          "custom": {
            "axisLabel": "",
            "axisPlacement": "auto",
            "barAlignment": 0,
            "drawStyle": "line",
            "fillOpacity": 0,
            "gradientMode": "none",
            "hideFrom": {
              "legend": false,
              "tooltip": false,
              "viz": false
            },
            "lineInterpolation": "linear",
            "lineWidth": 1,
            "pointSize": 5,
            "scaleDistribution": {
              "type": "linear"
            },
            "showPoints": "auto",
            "spanNulls": false,
            "stacking": {
              "group": "A",
              "mode": "none"
            },
            "thresholdsStyle": {
              "mode": "off"
            }
          },
          "mappings": [],
          "thresholds": {
            "mode": "absolute",
            "steps": [
              {
                "color": "green",
                "value": null
              },
              {
                "color": "red",
                "value": 80
              }
            ]
          }
        },
        "overrides": []
      },
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 8
      },
      "id": 10,
      "options": {
        "legend": {
          "calcs": [],
          "displayMode": "list",
          "placement": "bottom"
        },
        "tooltip": {
          "mode": "single",
          "sort": "none"
        }
      },
      "targets": [
        {
          "datasource": {
            "type": "postgres",
            "uid": "telemetry_db"
          },
          "format": "time_series",
          "group": [],
          "metricColumn": "none",
          "rawQuery": false,
          "rawSql": "SELECT\n  time AS \"time\",\n  range\nFROM detector\nWHERE\n  $__timeFilter(time)\nORDER BY 1",
          "refId": "A",
          "select": [
            [
              {
                "params": [
                  "range"
                ],
                "type": "column"
              }
            ]
          ],
          "table": "detector",
          "timeColumn": "time",
          "timeColumnType": "timestamp",
          "where": [
            {
              "name": "$__timeFilter",
              "params": [],
              "type": "macro"
            }
          ]
        }
      ],
      "title": "Detector Range",
      "type": "timeseries"
    },
    {
      "fieldConfig": {
        "defaults": {
          "color": {
            "mode": "continuous-RdYlGr"
          },
          "custom": {
            "axisGridShow": false,
            "axisLabel": "",
            "axisPlacement": "left",
            "barAlignment": 0,
            "drawStyle": "line",
            "fillOpacity": 0,
            "gradientMode": "none",
            "hideFrom": {
              "legend": false,
              "tooltip": false,
              "viz": false
            },
            "lineInterpolation": "linear",
            "lineWidth": 1,
            "pointSize": 5,
            "scaleDistribution": {
              "type": "linear"
            },
            "showPoints": "auto",
            "spanNulls": false,
            "stacking": {
              "group": "A",
              "mode": "none"
            },
            "thresholdsStyle": {
              "mode": "off"
            }
          },
          "mappings": [
            {
              "options": {
                "0": {
                  "color": "dark-red",
                  "index": 0,
                  "text": "Off"
                },
                "1": {
                  "color": "dark-green",
                  "index": 1,
                  "text": "On"
                }
              },
              "type": "value"
            }
          ],
          "max": 1,
          "min": 0,
          "thresholds": {
            "mode": "absolute",
            "steps": [
              {
                "color": "green",
                "value": null
              },
              {
                "color": "red",
                "value": 80
              }
            ]
          }
        },
        "overrides": []
      },
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 12,
        "y": 9
      },
      "id": 12,
      "options": {
        "legend": {
          "calcs": [],
          "displayMode": "list",
          "placement": "bottom"
        },
        "tooltip": {
          "mode": "single",
          "sort": "none"
        }
      },
      "targets": [
        {
          "datasource": {
            "type": "postgres",
            "uid": "telemetry_db"
          },
          "format": "time_series",
          "group": [],
          "metricColumn": "none",
          "rawQuery": false,
          "rawSql": "SELECT\n  time AS \"time\",\n  state\nFROM detector\nWHERE\n  $__timeFilter(time)\nORDER BY 1",
          "refId": "A",
          "select": [
            [
              {
                "params": [
                  "state"
                ],
                "type": "column"
              }
            ]
          ],
          "table": "detector",
          "timeColumn": "time",
          "timeColumnType": "timestamp",
          "where": [
            {
              "name": "$__timeFilter",
              "params": [],
              "type": "macro"
            }
          ]
        }
      ],
      "title": "Detector State",
      "type": "timeseries"
    },
    {
      "fieldConfig": {
        "defaults": {
          "color": {
            "mode": "continuous-RdYlGr"
          },
          "custom": {
            "axisLabel": "",
            "axisPlacement": "auto",
            "barAlignment": 0,
            "drawStyle": "line",
            "fillOpacity": 0,
            "gradientMode": "none",
            "hideFrom": {
              "legend": false,
              "tooltip": false,
              "viz": false
            },
            "lineInterpolation": "linear",
            "lineWidth": 1,
            "pointSize": 5,
            "scaleDistribution": {
              "type": "linear"
            },
            "showPoints": "auto",
            "spanNulls": false,
            "stacking": {
              "group": "A",
              "mode": "none"
            },
            "thresholdsStyle": {
              "mode": "off"
            }
          },
          "mappings": [],
          "max": 1,
          "min": 0,
          "thresholds": {
            "mode": "absolute",
            "steps": [
              {
                "color": "green",
                "value": null
              },
              {
                "color": "red",
                "value": 80
              }
            ]
          }
        },
        "overrides": []
      },
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 16
      },
      "id": 14,
      "options": {
        "legend": {
          "calcs": [],
          "displayMode": "list",
          "placement": "bottom"
        },
        "tooltip": {
          "mode": "single",
          "sort": "none"
        }
      },
      "targets": [
        {
          "datasource": {
            "type": "postgres",
            "uid": "telemetry_db"
          },
          "format": "time_series",
          "group": [],
          "metricColumn": "none",
          "rawQuery": false,
          "rawSql": "SELECT\n  time AS \"time\",\n  state\nFROM lasers\nWHERE\n  $__timeFilter(time)\nORDER BY 1",
          "refId": "A",
          "select": [
            [
              {
                "params": [
                  "state"
                ],
                "type": "column"
              }
            ]
          ],
          "table": "lasers",
          "timeColumn": "time",
          "timeColumnType": "timestamp",
          "where": [
            {
              "name": "$__timeFilter",
              "params": [],
              "type": "macro"
            }
          ]
        }
      ],
      "title": "Laser State",
      "type": "timeseries"
    }
  ],
  "refresh": "5s",
  "schemaVersion": 35,
  "style": "dark",
  "tags": [],
  "templating": {
    "list": []
  },
  "time": {
    "from": "now-5m",
    "to": "now"
  },
  "timepicker": {},
  "timezone": "",
  "title": "New dashboard",
  "uid": "1i9YhEu7z",
  "version": 2,
  "weekStart": ""
}
```

The Wrath of Khan

One should land a device orbiting around a plant on the planet's surface. But the hardware was manipulated, and hence some sensors didn't work. One could upload a patched binary, which should safely land the device on the surface. In the unmodified state, the device just crashed into the planet. The sensor readings and applied inputs from the PID controller were shown in several graphs after the simulation ran.

Analysis of the binary

It became clear that the binary communicates with the hardware (controller and sensor) via network TCP 127.0.0.1:6000. The binary sent the controller inputs (boost in X, Y, Z, parachute activation, and heatshield deactivation) in an array of 5 32-bit values (3x floats and 2x integer). In return, the binary received the current sensor values like G-forces, velocity, and other measurements.

Analysis of the trajectory and solution

From the unmodified binary, one could see that the PID controller applied a full boost in the X and Y direction to leave the orbit and start the landing approach. Then the heatshield should be deactivated once the velocity is low enough. The parachute should be deployed for the final landing to guarantee a safe touch-down. In the unmodified binary, only the PID controller to initiate the approach worked correctly.

Since we hadn't had any success patching the binary, we wrote our own "controller" in a static compiled ELF arm binary, which read the sensor values and applied them according to inputs. Initially, this approach was only used to understand the resulting trajectories better but led to the solution. Our binary applied forces to the X (10.0f) and Y (-10.0f) directions similar to the PID controller. That was enough to leave the orbit; luckily, no real controller was necessary. The resulting graphs show that the velocity is shallow from around 0.17 hours into the flight. As it is safe to disable the heatshield with the low velocity, we could do so without burning in the atmosphere. As we still crashed into the surface, we ejected the parachute around 0.40 hours into the flight. That was enough to ensure a safe landing.

```c
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/ioctl.h>
#define PORT 6000

int main(int argc, char const* argv[])
{
    int sock = 0, valread, client_fd;
    struct sockaddr_in serv_addr;

    char buffer[1024] = { 0 };
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }
    int v11 = 0;
    ioctl(sock, 0x5421uLL, &v11); // Disable buffering

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary
    // form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)
        <= 0) {
        printf(
            "\nInvalid address/ Address not supported \n");
        return -1;
    }

    if ((client_fd
         = connect(sock, (struct sockaddr*)&serv_addr,
                   sizeof(serv_addr)))
        < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
    unsigned char buf[0x1c];
    unsigned int buf_send[5];
    for (int i = 0; i < 100000; i++) {
        read(sock, buf, 0x1c); // Read sensor data. We ignore them for now

        *(float*)&buf_send[0] = 0.0;
        *(float*)&buf_send[1] = 0.0;
        *(float*)&buf_send[2] = 0.0;

        buf_send[3] = 0;
        buf_send[4] = 0;

        if (i >0) { // Boost into X and Y direction
                *(float*)&buf_send[0] = 10.0f; // Shoot X-Axis
                *(float*)&buf_send[1] = -10.0f; // Shoot Y-Axis
        }

        if (i > (int)(0.17*3600.0f)) { // Disable the heatshield once slow enough
                buf_send[4] = 1; // heatshield
        }

        if (i > (int)(0.4*3600.0f)) { // Deploy the paracute starting from 0.4 hours into the flight
                buf_send[3] = 1; // parachute
        }

        write(sock, buf_send, 0x14); // Write controller inputs back to the hardware
    }
    // closing the connected socket
    close(client_fd);
    return 0;
```

```python
from pwn import *
import requests
import sys

context.arch = "aarch64"
```

```

if len(sys.argv) < 2:
    print("Usage: solve.py <Lander.bin>")
    exit(-1)

#patch_binary(sys.argv[1])

ticket = "ticket{charlie503913romeo3:
    GB42NezJx58gFTy0vZK_9nJupPzoHUaWb0Jpz3CBnuxznaO-W1jmO-lSGgWu9A0r8A}"

r = remote("khan.satellitesabove.me", 5300)
r.sendlineafter("Ticket please:", ticket)

r.recvuntil("Upload binary at: ")
to_upload = r.recvline().decode().replace("\n", "")
log.info(f"Webserver to upload: {to_upload}")
files = {'file': open(sys.argv[1],'rb')}

response = requests.post(f"http://{to_upload}/upload", files=files)

r.recvuntil("Please wait for probe simulation to run")
log.info("Waiting for telemetry to arrive")
print(r.recvall())
```

(Don't) Fly Me to the Moon

Challenge Description

Revenge of the Space Math

An old rocket booster is about to crash in to the Galactic Mining And Tunneling (GMAT) corporation's Lunar space mining operation! Fortunately we've attached a robotic servicing module to it with some limited deltaV. Figure out the maneuvers needed to bring the booster safely to the Earth-Moon L4 Lagrange point. Make sure the booster stays there for a while until we figure out what to do with it.

Note: The viewer will not update with your trajectory until after you finish sending maneuvers, the default trajectory in there when the challenge starts isn't worth looking at.

Solution

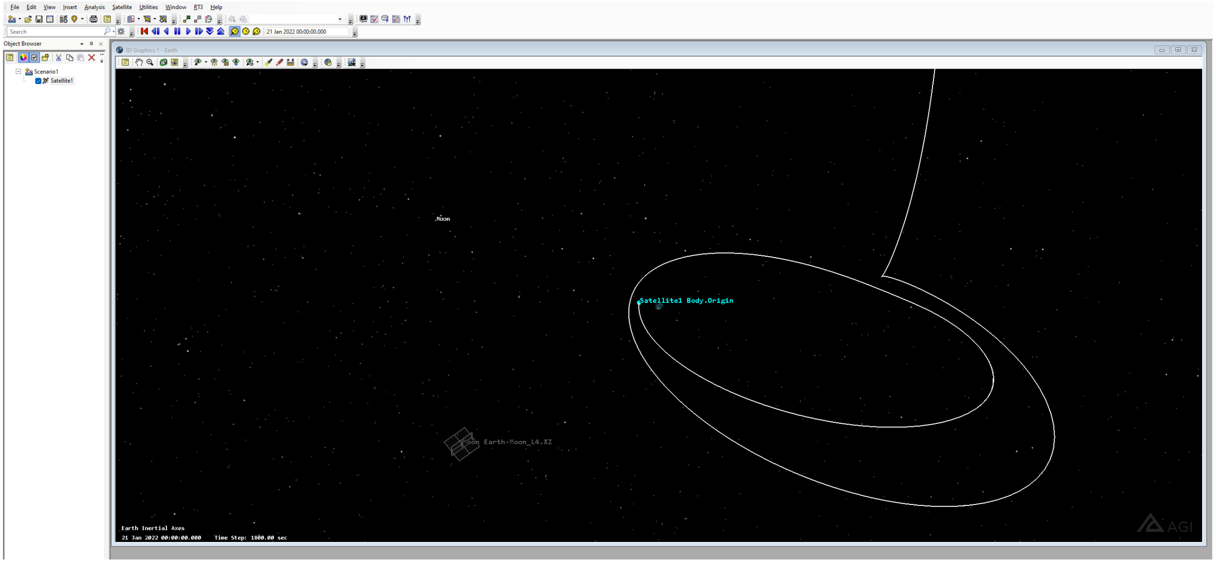
After connecting to the challenge server, we are greeted with the following message:

```
Save our mining operation!
The orbital elements for the booster are:
Gregorian TT: 2022 Jan 21.0
```

```
Semimajor axis: 295911.9425900026 km
Eccentricity: 0.8898246770000006
Inclination: 27.0092517 deg
RAAN: 17.82128320000001 deg
Argument of periapsis: 149.2278077 deg
Mean anomaly: 0.1621535087
--------------------------------------------
Provide a list of maneuvers to put the booster at L4 and keep it nearby
If your manuevers are valid you can view the trajectory at
    3.219.234.239:31463
You can manuever as many times as you want but you only have 2.0 km/s
    of deltaV left
Time is in Gregorian TT:  YYYY-MM-DD HH:MM:SS.sss
Manuevers are in Earth Centered Mean of J2000 Equitorial Coordinate
    system
Input your manuevers in the following format:
Time,Delta VX,Delta VY,Delta VZ
Enter 'DONE' when you want have added all your maneuvers
Input next maneuver:
```

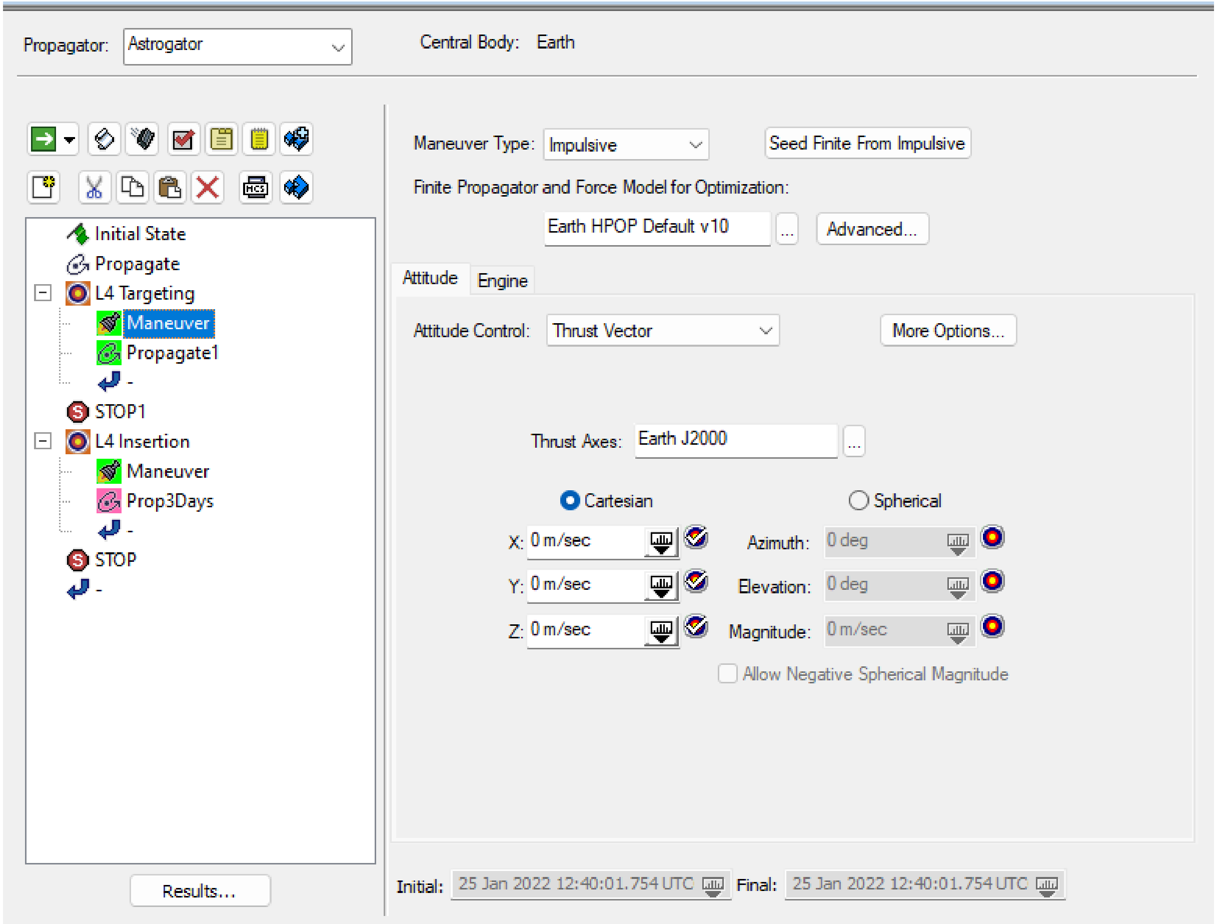
These are the Keplerian elements of our booster, together with the instructions on how to input the maneuvers.

Our first step is to model the satellite and the Earth-Moon-L4 in STK:



Doing this, we can see that we get our closest approach to the L4 approximately at the apoapsis of our orbit. We have a large amount of deltaV available, so we plan to do a burn to get as close to the L4 point as possible and then do another burn to fix our momentum and stay with the L4.

To achieve this, we first propagate to the apoapsis. There, we now use astrogator to find a burn that brings us as close to the L4 as possible:



Propagator: Astrogator

Central Body: Earth

- Initial State
- Propagate
- L4 Targeting
  - Maneuver
  - Propagate1
  - -
- STOP1
- L4 Insertion
  - Maneuver
  - Prop3Days
  - -
- STOP
- -

Maneuver Type: Impulsive

Finite Propagator and Force Model for Optimization: Earth HPOP Default v10

Attitude Control: Thrust Vector

Thrust Axes: Earth J2000

● Cartesian ○ Spherical

X: 0 m/sec

Y: 0 m/sec

Z: 0 m/sec

Azimuth: 0 deg

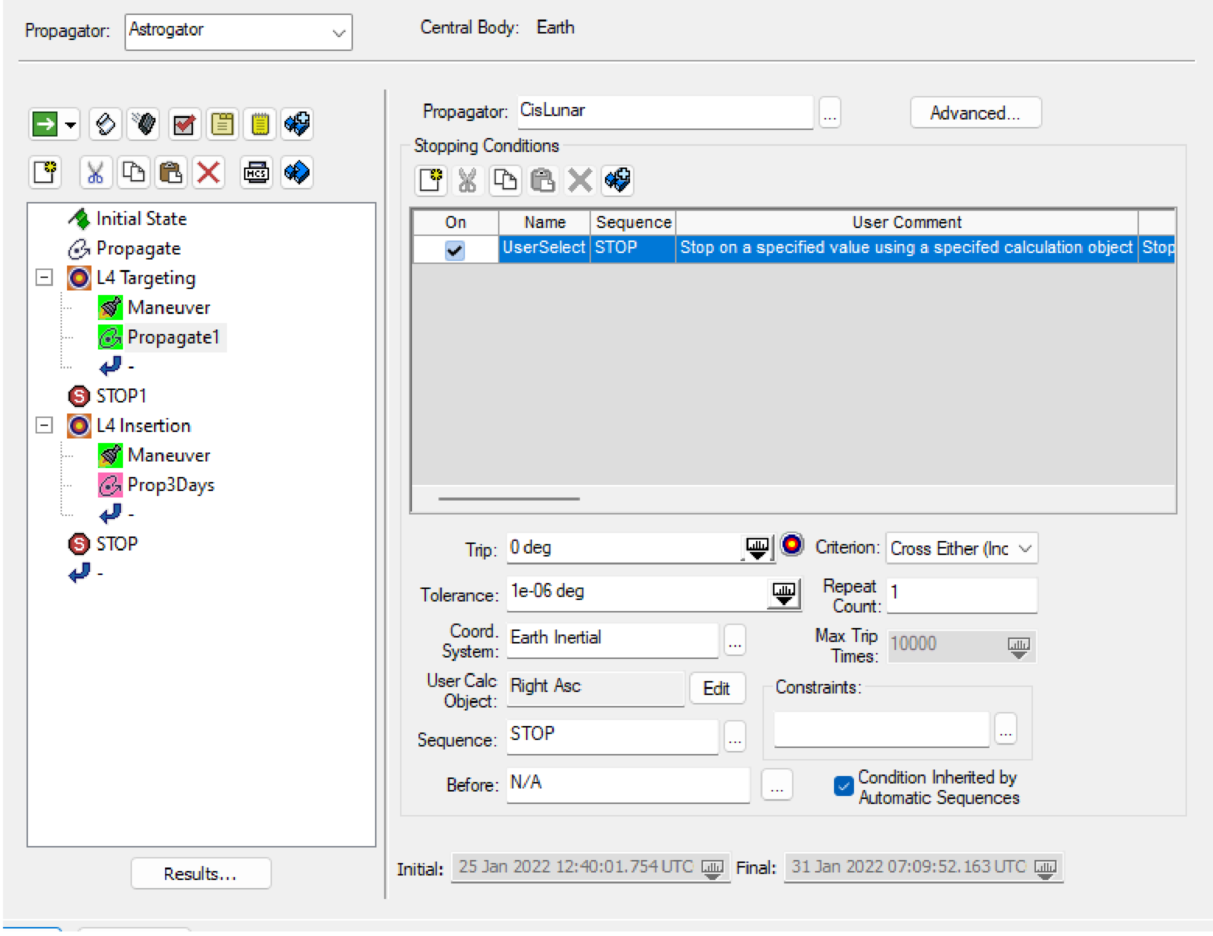
Elevation: 0 deg

Magnitude: 0 m/sec

☐ Allow Negative Spherical Magnitude

Initial: 25 Jan 2022 12:40:01.754 UTC Final: 25 Jan 2022 12:40:01.754 UTC

We do an impulsive burn where we allow astrogator to burn in any direction.



Propagator: Astrogator

Central Body: Earth

Propagator: CisLunar

On	Name	Sequence	User Comment	
☑	UserSelect	STOP	Stop on a specified value using a specifed calculation object	Stop

Trip: 0 deg

Criterion: Cross Either (Inc

Tolerance: 1e-06 deg

Repeat Count: 1

Coord. System: Earth Inertial

Max Trip Times: 10000

User Calc Object: Right Asc

Constraints:

Sequence: STOP

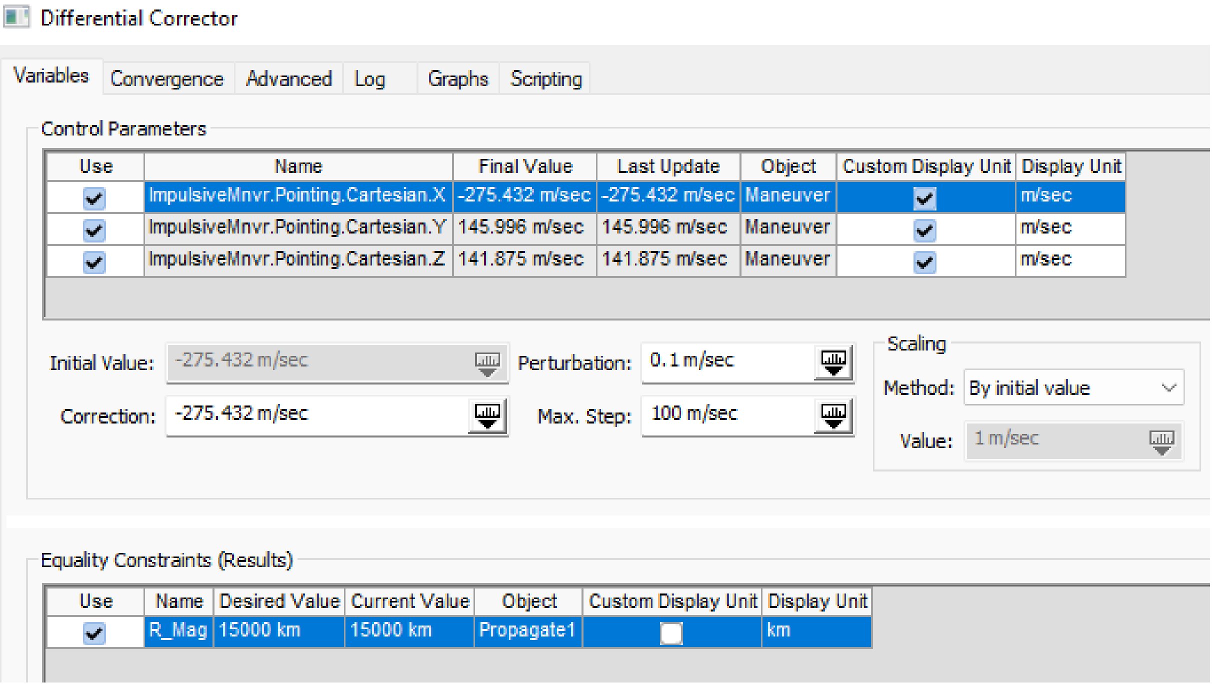
Before: N/A

☑ Condition Inherited by Automatic Sequences

Initial: 25 Jan 2022 12:40:01.754 UTC

Final: 31 Jan 2022 07:09:52.163 UTC

We propagate until our right ascension to the L4 is 0 degrees. This is just a convenient stopping condition that allows astrogator to find a burn that quickly brings us to the L4.



Differential Corrector

Control Parameters

Use	Name	Final Value	Last Update	Object	Custom Display Unit	Display Unit
☑	ImpulsiveMnvr.Pointing.Cartesian.X	-275.432 m/sec	-275.432 m/sec	Maneuver	☑	m/sec
☑	ImpulsiveMnvr.Pointing.Cartesian.Y	145.996 m/sec	145.996 m/sec	Maneuver	☑	m/sec
☑	ImpulsiveMnvr.Pointing.Cartesian.Z	141.875 m/sec	141.875 m/sec	Maneuver	☑	m/sec

Initial Value: -275.432 m/sec

Perturbation: 0.1 m/sec

Correction: -275.432 m/sec

Max. Step: 100 m/sec

Scaling

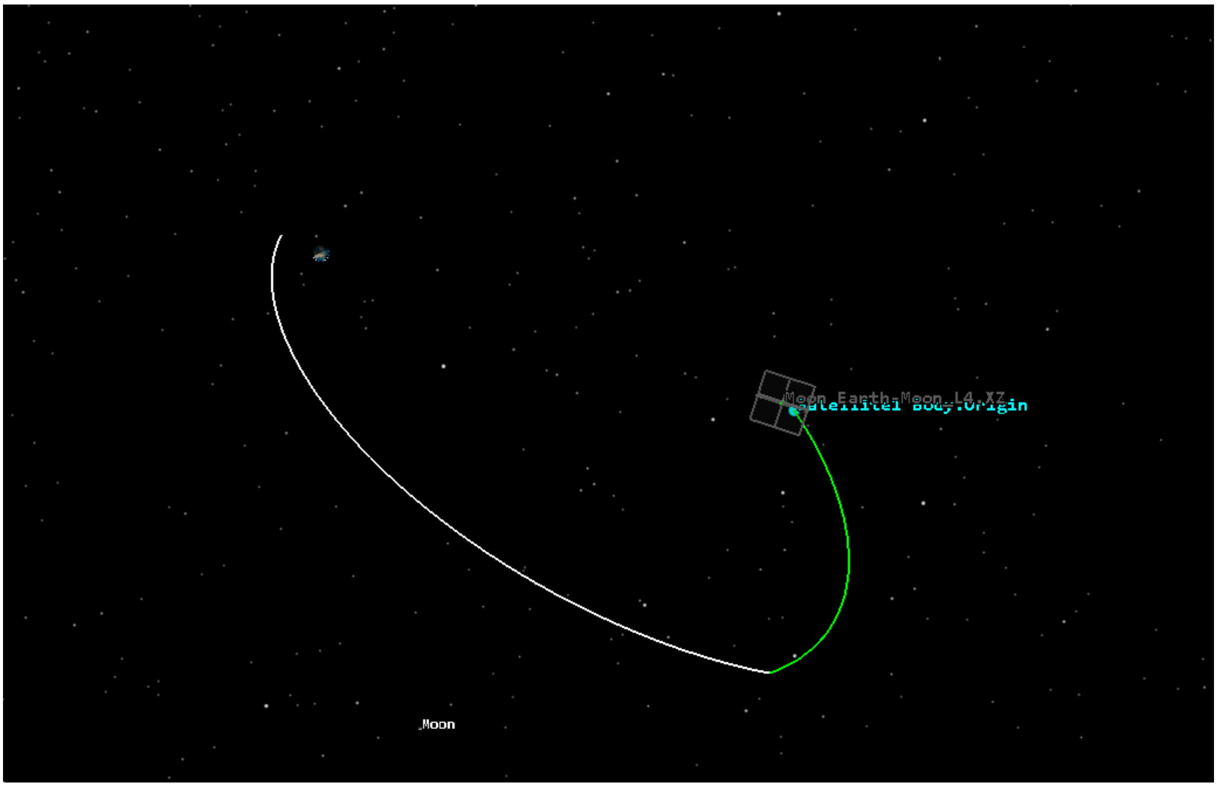
Method: By initial value

Value: 1 m/sec

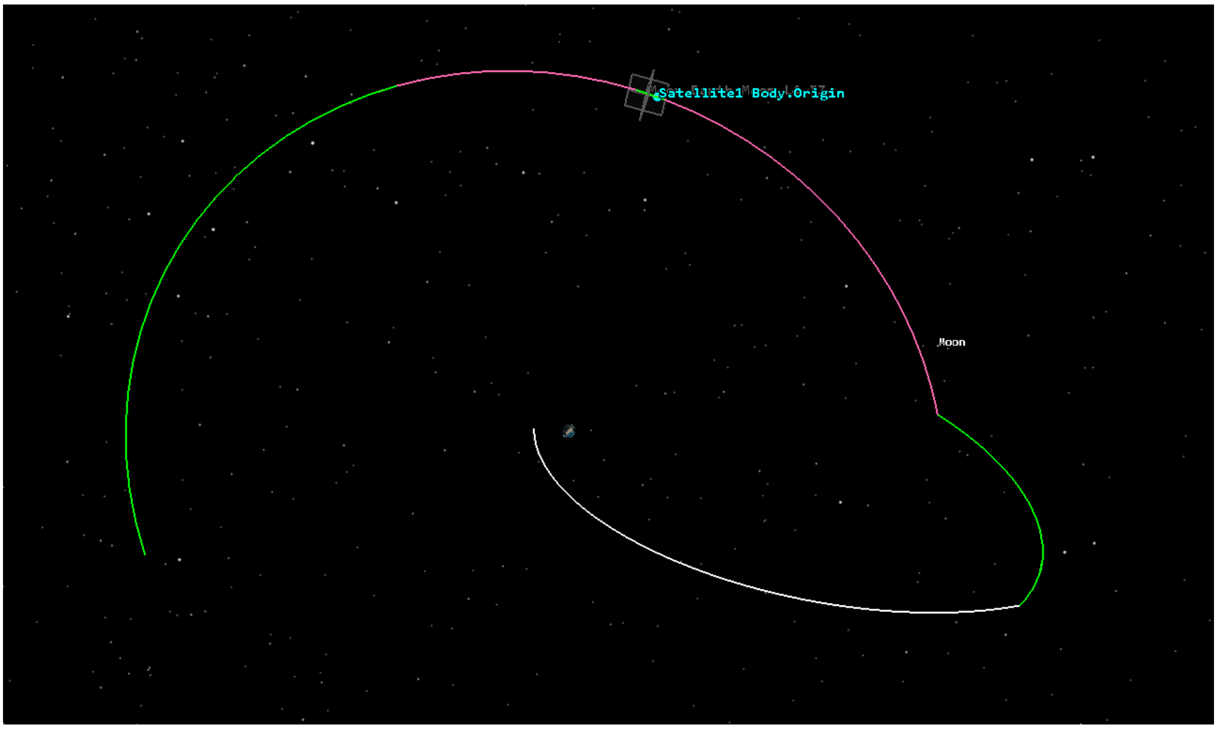
Equality Constraints (Results)

Use	Name	Desired Value	Current Value	Object	Custom Display Unit	Display Unit
☑	R_Mag	15000 km	15000 km	Propagate1	☐	km

As the target for differential corrector, we choose a distance of at most 15000km to the L4. Astrogator very quickly converges on this burn:



The first step is done; we just have to stay with the L4 for a while to make the challenge server happy. To achieve this, we do another astrogator maneuver. This time, we look for a burn that will have us within 15000km of the L4 after three days of propagation. This is the resulting orbit:



Reading off the parameters of the burns and sending them to the challenge server then got us the flag:

```python
from pwn import *

r = remote('fly_me_to_the_moon.satellitesabove.me', 5100)
r.readline()
r.sendline(b'ticket{xray177839foxtrot3:
    GE0oskUSCa2h0q7PIOrsYvzHElOHPLqghUXUEpUFMrnt9izQl6M4ZIPHSV6nQlaEfw}'
    )

r.recvuntil(b'If your manuevers are valid you can view the trajectory
    at ')
print(f'http://{r.readline()[:-1].decode()}')

burns = [
    ['2022-01-25 12:40:01.754', '-0.275432', '0.145996', '0.141875'],
    ['2022-01-31 07:09:52.163', '0.657034', '0.533422', '0.195626']
]

for b in burns:
    r.sendlineafter(b'Input next maneuver:\n', ','.join(b).encode())
r.sendlineafter(b'Input next maneuver:\n', b'DONE')
```

```

r.interactive()
```

FDA Approved Beef

We are given a satellite control platform where we can issue commands to a virtual satellite software controller. Some commands included uploading a file, downloading a file, sending shell commands, and a mysterious flag command.

Playing with the shell

The ability to execute arbitrary shell commands meant we tunnel-visioned and started executing commands to learn more about the remote system. To receive the output of our command, we piped the result into a file and downloaded it via the download command. First, we started to dump the whole file system and current process list:

```
<...>
/home
/home/space
/home/space/.bashrc
/home/space/.bash_logout
/home/space/.profile
/home/space/fsw
/home/space/fsw/attempt.txt
/home/space/fsw/satellite.exe
<...>

USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
space          1  0.0  0.0  20136  3520 ?        Ss   15:03   0:00 bash /run_space.sh
space         78  0.0  0.0 206476  5412 ?        Sl   15:03   0:00 ./satellite.exe -a 172.16.238.3 -p 50000
space         94  0.0  0.0   4640   816 ?        S    15:07   0:00 sh -c eval 'ps -aux ' 1>>log 2>&1
space         95  0.0  0.0  36164  3240 ?        R    15:07   0:00 ps -aux
```

Okay, so `satellite.exe` is probably the binary executing the commands, but where is the flag?

Analyzing the Binary

Upon inspecting the binary, we noticed quite a few strings containing “flag”. So we started digging deeper.

The first thing it does is set up the commands; for the FLAG command, it would read a `.FlagData` file into memory and delete it from the disk.

Next, we took a lock into the FLAG command itself. Just deducting the control flow from the strings present, it would seem that first, a file is read from the disk, checking its content and, depending on the validity, unlocking the flag or not. Thankfully we have already seen the fallback file “attempt.txt” in the home directory dump. Letting this file, which contained line-separated items of strings and numbers, we got the result that only a handful of those items were valid, with the rest being invalid.

Finding the correct input

Next, we took to reverse the checking algorithm. First, it would parse the file line by line, parsing each element as an integer. Then it would go on a do some checks against the number. If it’s bigger than 1, odd, not dividable by 3, not dividable by 5, not dividable by 7… wait a minute, this looks like a prime check. Throwing together a few prime numbers and letting it get checked revealed indeed that all of our items are valid. However, this alone wouldn’t be enough for the flag, so we continued.

To reach the statements where the flag is printed, we must pass an additional check. Each valid element is added to a list whose length must be equal to 2022. So we quickly threw together a script generating 2022 prime numbers.

```python
def is_prime(n):
    for i in range(2, n):
        if (n % i) == 0:
            return False
    return True

primes = []
cur = 2
while len(primes) != 2022:
    if is_prime(cur):
        primes.append(cur)
    cur += 1

open("attempt.txt", "w").write("\n".join([str(p) for p in primes]))
```

uploaded it to the satellite and ran the FLAG command, which unlocked the flag and wrote it to flag.txt, which we then downloaded to receive our flag:

```
flag{golf446904lima3:GE56VkL1Vz0Pga45CdRyMC_NfIZBgPBayQ38spYg80mjYkSc4N1Q7JgBfC926TX
3jZ4qLYqAOzsge8LKdjfZSy0}
```

Powerpoint

Intro

In this challenge, we are given control over the direction of an antenna and are tasked with receiving the flag from a satellite.

Challenge Description

> Break out Microsoft Office because it's time to make some slides… just kidding!
>
> There is a satellite out there transmitting a flag. You are on a moving platform and you don't know how you will be moving or where the transmitter is or how the transmitter is moving. Luckily you have a steerable antenna! To get the flag:
>
> 1. Find the transmitter by steering the antenna.
> 2. Keep the antenna pointed at the transmitter and try to maximize signal power.
> 3. Decode the samples coming in from your receiver.
>
> Your co-worker told you they think the satellites is "North-ish and sort of close to the horizon"
>
> Send `az`, `el` commands to the antenna; an antenna command is formatted as a string for example if az=77.1 and el=73.2
>
> send:
>
> ```
> 77.10,22.2\n
> ```
>
> Please send one command at a time.
>
> After you send a command the receiver will reply with a fixed number of IQ samples via TCP.

First Thoughts

Both the satellite and we are moving, which likely means we will have to track the satellite in some fashion. Further, we don't even know the initial satellite location, only that it is "North-ish and sort of close to the horizon".

We communicate by sending via azimuth and elevation. Exactly north and directly at the horizon would be both an azimuth and elevation of 0 degrees. Pointing the antenna there, we get a response of `Tracking Lost`, so we'll have to go searching.

Every time we send a pointing direction, we get sent exactly 1024 samples over a second TCP connection. Each sample batch contains 8192 bytes, 8 bytes per sample. Since we don't have tracking yet, let us first fix that:

1. Find the transmitter by steering the antenna.

To find the initial position of the satellite, we write a short python script to scan the sky. Having no idea of the sensitivity, we pick a resolution of 1 degree in a grid of 1-degree increments. We start with different initial positions on multiple connections to speed up the search.

We first find it at time 96 at location 15.0/11.5:

```
Got 15.0,11.5
Command 96 received from user
Pointing to  Az: 15.0  El: 11.5
Sending 1024 samples over TCP/IP
```

During this phase, we also notice that the satellite spawns at the exact same location each time. This will make this task lot easier since we can now just assume the initial position.

Knowing the position at time 96, we play around with the code and find the initial position at time 0 to be approx. 20.0 / 11.0.

We also find the server to stop sending samples at time ~300, so likely flag is transmitted by then.

2. Keep the antenna pointed at the transmitter and try to maximize signal power.

According to the task description, both the satellite and our antenna are moving, so we'll have to change the direction the antenna is pointing to over time.

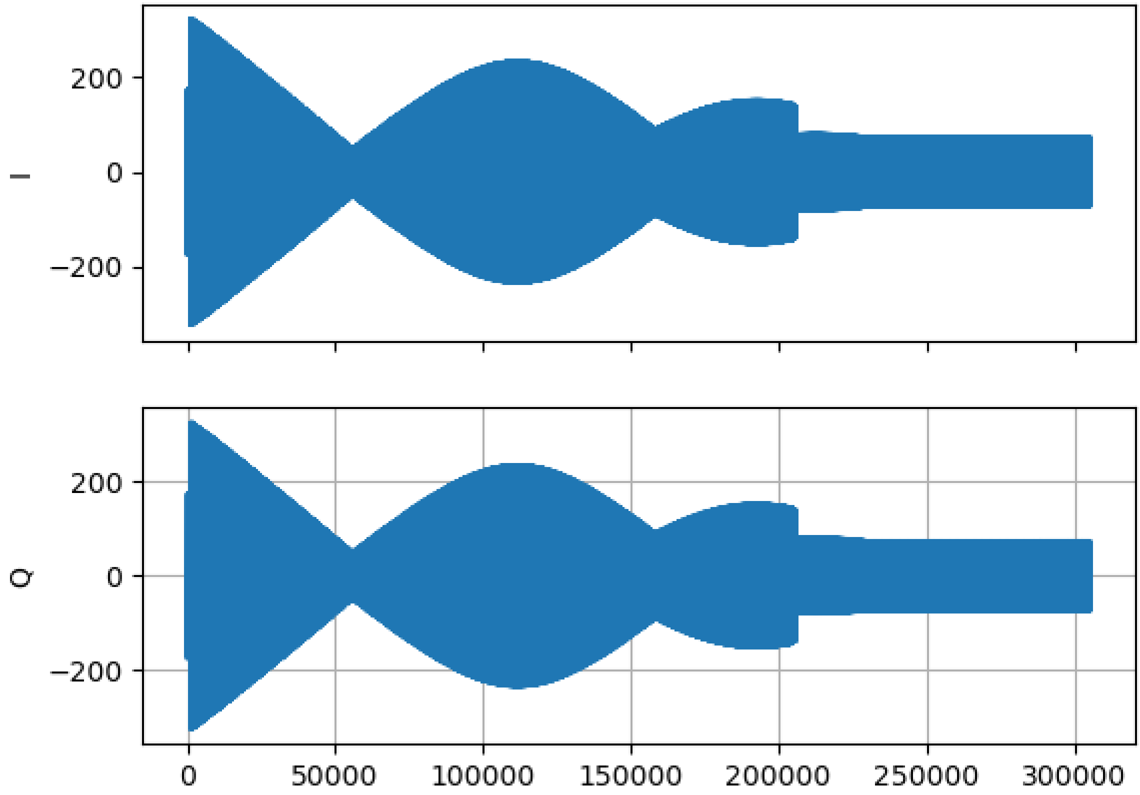
We look at the received signal power to see how well it's pointing currently. The samples are in the common I/Q 32-bit float format, which we parse with numpy's `np.complex64` datatype. By summing each samples magnitude squared we get the power (`np.sum(np.abs(samples)**2)`).

We don't put much effort into automated tracking and play around with linear parameters until we get a sequence where we don't loose tracking at all, which the server handily tells us.

The result is a piecewise-linear function of angles:

```
if time < 200:
    el = 20 - time/100*4
    az = 11 + time/69*4
else:
    az = 19.6
    el = 14
```

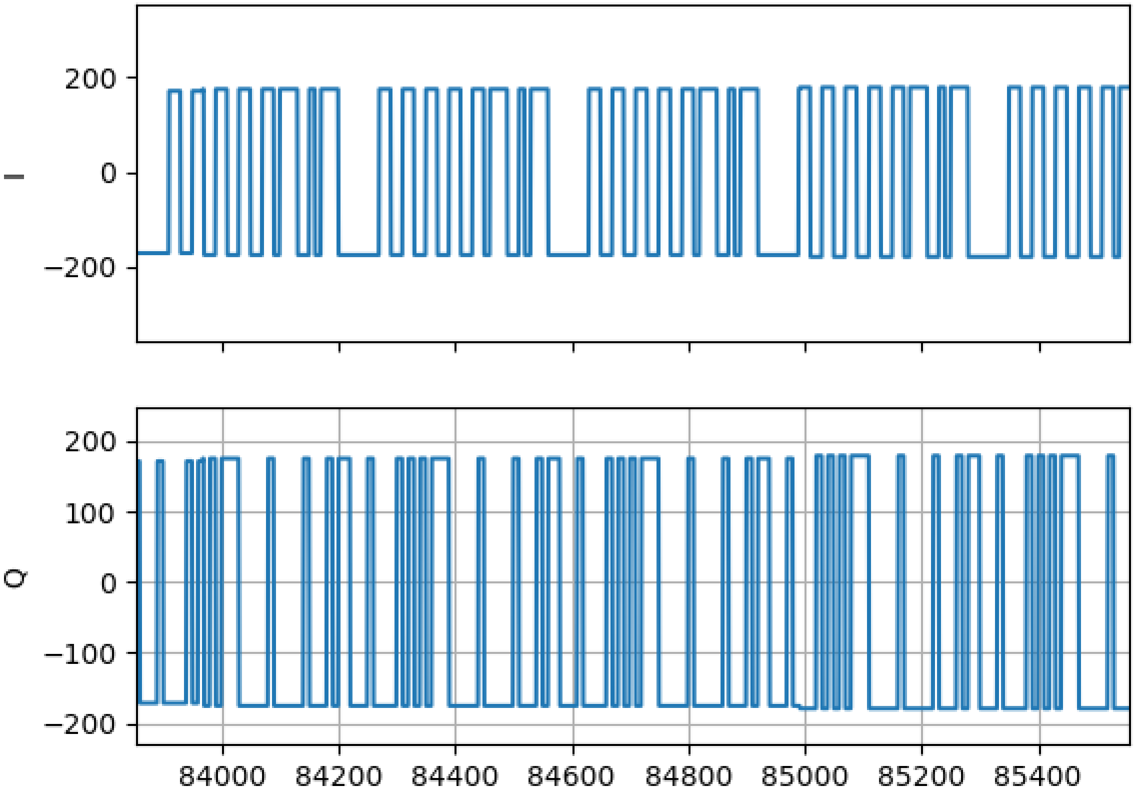
Surprisingly, there is zero noise, contrary to the task description; we don't have to maximize signal power. The signal envelope of our tracking looks like this:



Having dumped a full-duration signal, we get to analyze it.

3. Decode the samples coming in from your receiver.

When we look at the collected IQ samples in more detail, we can see that we have perfect positive/negative flanks with zero noise. The shortest symbol length is ten samples, kept precisely over the whole signal.



This makes it nice to decode. There are only four states: Both I and Q can be either positive or negative. We are thus transmitting in QPSK. Since one symbol is one of four combinations, it encodes exactly two bits.

To map the IQ samples to symbols, we first loop over all samples, determine the symbol, then run-length encode the resulting sequence. This allows us to detect repeated symbols quickly.

But we don't know the encoding. Which symbol corresponds to which bits?

The signal is repetitive at first. Only at around time 280 do things get interesting. This corresponds well with a message printed when connecting to the challenge: `Flag starts at byte 7200`. 280 blocks, of 1024 samples, of which ten samples per symbol and 2 bits per symbol gives us `280*1024/10*2/8 = 7168` bytes.

After some failed attempts at brute-forcing the symbol mapping, we look at the samples manually, together with the known plaintext:

```
FLAG in binary is 01100110,01101100,01100001,01100111
Each byte has to be encoded in 4 symbols. Relevant symbols are: 3131,
    2031, 1231, 0131

f 01 10 01 10 - 3131
```

```
l 01 10 11 00 - 2031
a 01 10 00 01 - 1231
g 01 10 01 11 - 0131
```

Notice how every byte starts with 0110, and every symbol block ends with 31? We are in little-endian! That's why our brute-force attempts failed.

The correct encoding is:

```
0 - 11
1 - 01
2 - 00
3 - 10
```

But each block of four symbols has to be reversed.

The decoded output of our signal, starting at time 280, is:

```
ASat
HackASat
HackASat
HackASat
flag{uniform646828foxtrot3:GNFp0KktopvIwf3eGgqoYAvGJQSbObBZIhK7kpMnnSP3ZROl6copIp2VPbY_D_
AZotRP7dCXK_JgNE8zzAg1b9Q}
flag{uniform646828foxtrot3:GNFp0KktopvIwf3eGgqoYAvGJQSbObBZIhK7kpMnnSP3ZROl6copIp2VPbY_D_
AZotRP7dCXK_JgNE8zzAg1b9Q}
flag{uniform646828foxtrot3:GNFp0KktopvIwf3eGgqoYAvGJQSbObBZIhK7kpMnnSP3ZROl6copIp2VPbY_D_
AZotRP7dCXK_JgNE8zzAg1b9Q}
flag{uniform646828foxtrot3:GNFp0KktopvIwf3eGgqoYAvGJQSbObBZIhK7kpMnnSP3ZROl6copI0
```

Solve Scripts

Signal Aquisition

```python
from pwn import *
import time
import numpy as np

r = remote('power_point.satellitesabove.me', 5100)
r.readline()
r.sendline(b'ticket{uniform646828foxtrot3:GIB9H-lU1KLH_MgeFZtAIJ2csh1kWFmYdoM02WTx34jQctur0qmK7GzPKgA9_ZrNHw}')
r.recvuntil(b"Antenna pointing TCP server accepts commands at")
antenna_ip = r.recvline().strip().split(b":")
```

```
r.recvuntil(b"Sample TCP server will provide samples at ")
sample_ip = r.recvline().strip().split(b":")


print(antenna_ip, sample_ip)
time.sleep(0.5)
samples = remote(*sample_ip)
time.sleep(0.5)
antenna = remote(*antenna_ip)
time.sleep(0.5)

outfile = open("out.iq", "wb")

for i in range(300):
    if i < 200:
        el = 20 - i/100*4
        az = 11 + i/69*4
    else:
        az = 19.6
        el = 14
    antenna.sendline(f"{az},{el}".encode())
    rsp = r.recvuntil(b"samples over TCP/IP")
    print(rsp)
    buf = samples.recv(8192) ## 1024 samples
    while len(buf) < 8192:
        buf += samples.recv(8192-len(buf))
    print(el, az, len(buf), sum(buf))

    outfile.write(buf)
    npbuf = np.frombuffer(buf, dtype=np.csingle)
    print(np.sum(np.abs(npbuf)**2))


outfile.close()
```

Signal Decoding

```
import numpy as np
import matplotlib.pyplot as plt
from pwn import group
from itertools import groupby

def run_length_encode(data):
    return ((x, sum(1 for _ in y)) for x, y in groupby(data))

samples = np.fromfile('out.iq', np.complex64)

symbols = []
for val in samples:
    i = np.real(val)
    q = np.imag(val)
    if i > 0 and q > 0:
        symbols += [0]
    elif i > 0 and q < 0:
        symbols += [1]
    elif i < 0 and q < 0:
        symbols += [2]
    else:
        symbols += [3]

## Plot signal for debugging
figure, axis = plt.subplots(2, 1, sharex=True)
axis[0].plot(np.real(samples))
axis[1].plot(np.imag(samples))
axis[0].set_ylabel("I")
axis[1].set_ylabel("Q")
plt.show()

dedupsym = []
start_block = 280
for sym,rle in run_length_encode(symbols[start_block*1024:]):
    dedupsym += [sym] * int(round(rle/10))

mapping = ('11', '10', '00', '01')
out = ""
for s in dedupsym:
    out += mapping[s]

outrev = ""
for x in group(8, out):
    outrev += chr(int(x[::-1], 2))

print(outrev)
```