# Description

An arrogant prince is cursed to live as a terrifying beast until he finds true love.

# Flag

```
WMCTF{Dont_ever_tell_anybody_anything___If_you_do__you_start_missing_everybody}
```

# Writeup

This challenge is an instance of BEAST (Browser Exploit Against SSL/TLS) attack. This is a real world attack that was used to exploit against SSL 3.0/TLS 1.0 protocol and had prompted the movement to TLS 1.1.

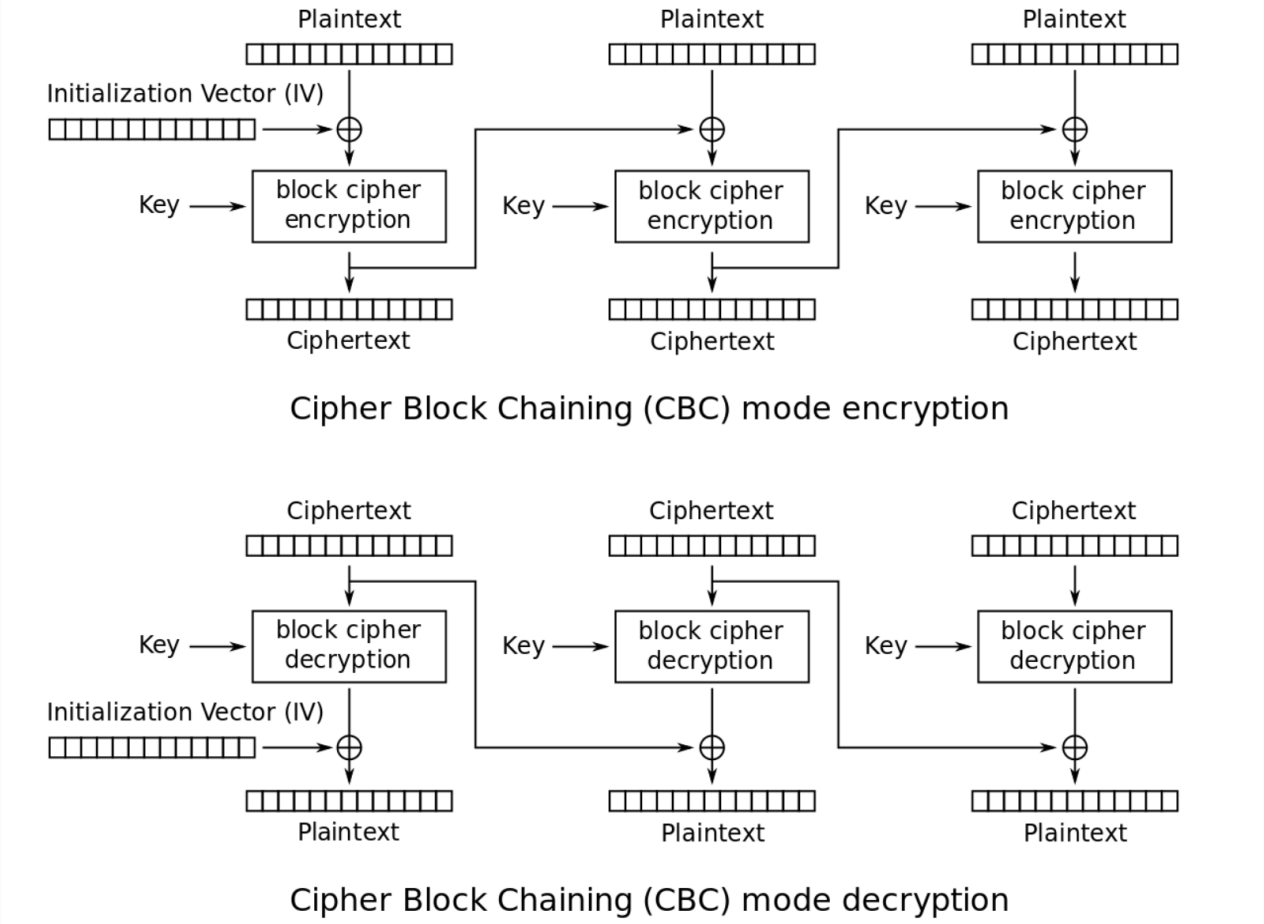
Regarding more information about the attack, we refer to the following resources:

- How the BEAST Attack Works | Netsparker
- Origin Paper: Here Come The ⊕ Ninjas

---

In this challenge, A simulation of the security game as described in the paper is implemented with some modification. To get the flag, we need to guess a secret 48 random bytes. The only useful function the server provided is encrypting the secret prepended by something the client gives by using AES-CBC.

CBC mode is one of the most popular block cipher modes of operations. It can be shown as below:



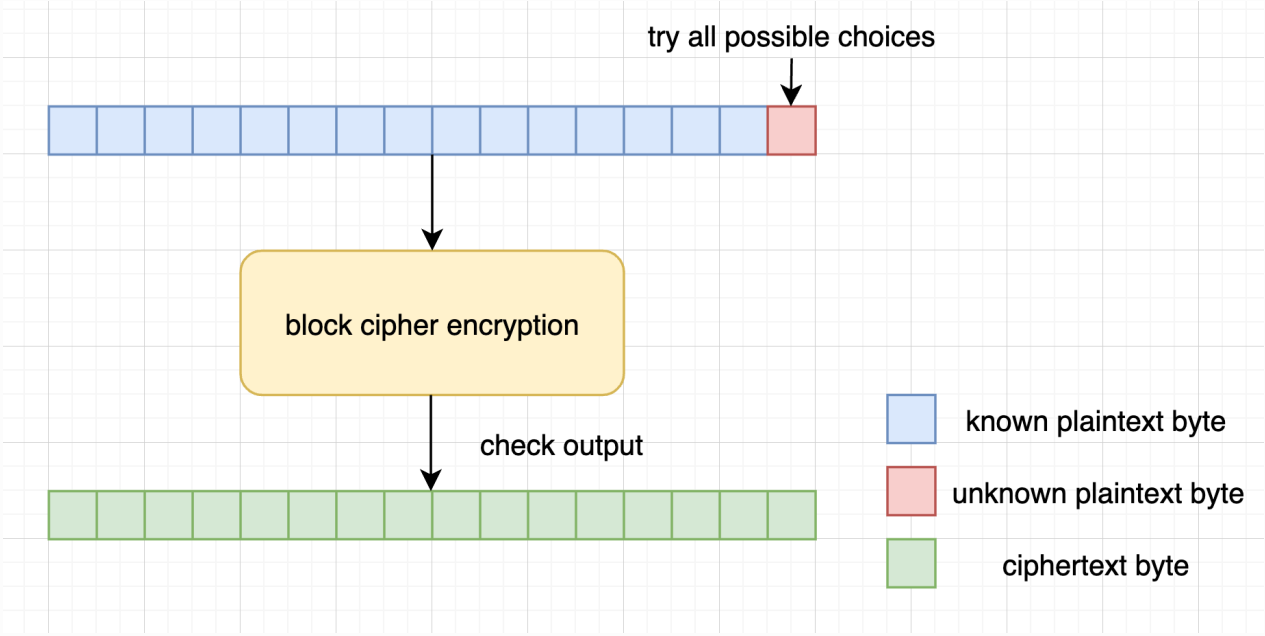

(picture from **wiki**)

The fatal vulnerability in this challenge is that, the initialization vector (IV) can be predicted during each encryption. This gives us space to manipulate the input of the *block cipher encryption*. Along with the (strange) encryption function the server provided, we can amout a chosen plaintext attack to recover the secret bytes.

---

We can consider the *block cipher encryption* as a pseudorandom permutation (PRP) that randomly permutes the input 16 bytes to another 16 bytes. The permutation depends only on the key and the same input results in the same output. So, if we are given a ciphertext (16 bytes), we can try all the possible input to the *block cipher encryption* until the output is the ciphertext. No doubt that this is infeasible because the size of plaintext space is $256^{16}$. However, if we have knownledge of the first 15 bytes of the input, it's easy to recover the last unkown byte by bruteforcing all the possible choices (only 256) of the last bytes, and check whether the output is the corresponding ciphertext.

> We mean *block cipher encryption* as AES encryption using ECB mode with one block.




(How to recover the last byte)

---

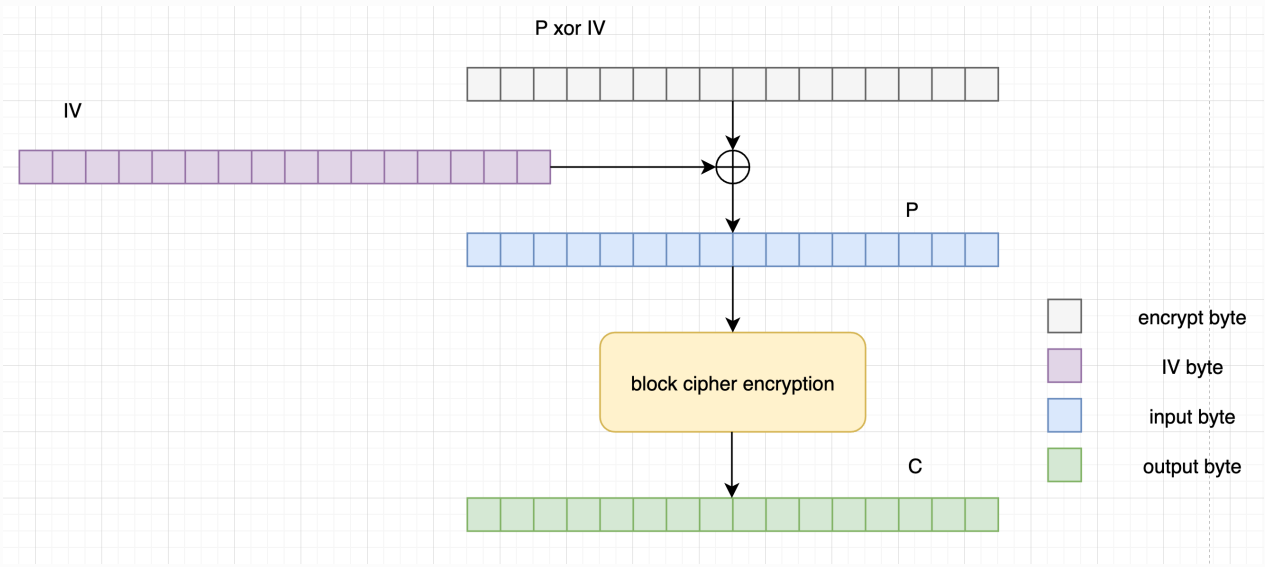
Back to this instance, the chosen plaintext attack consists of two phases:

1. Get the ciphertext of 15 known plaintext bytes and 1 unknown byte that we want to recover.
2. Try all the possible choices of the last unknown byte until the output is exactly the ciphertext, thus recovering the last byte.

But, this arises two questions as well:

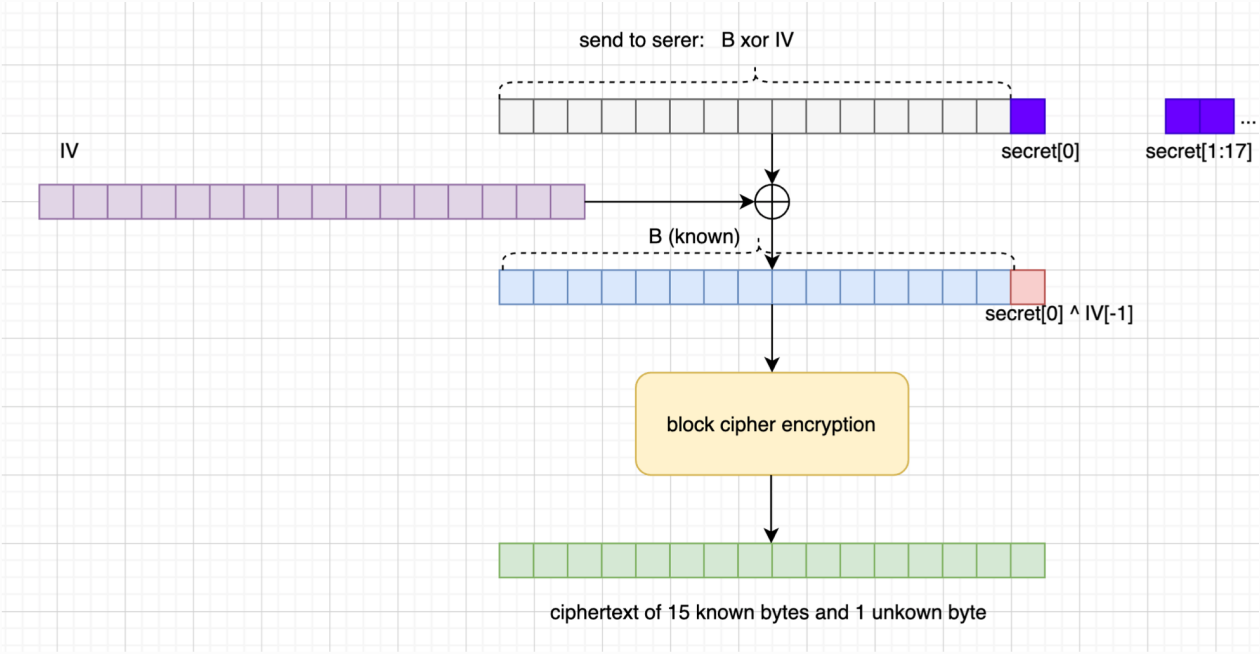
1. How can we control the input of the *block cipher encryption*?
2. How come we are aware of the 15 plaintext bytes?

For the first one, we can find that the IV of each encryption is predictable. Strictly, IV should only stand for the first xor vector before *block cipher encryption*. Here, however, we mean IV by all the xor vectors. The first IV is provided by the server and later ones are the ciphertexts of previous block. To control the input of the *block cipher encryption*, we can send the server a payload that has been xored with the predicted IV. For example, if we want the input of *block cipher encryption* to be $P$, and the predicted IV is $iv$, we can send the payload as $P \oplus iv$.




(How to control the input of the *block cipher encryption*)

As to the second one, since the server receives anything the client sends, we can control the block boundary by prepending well-prepared amounts of bytes. For example, if we send 15 bytes $B$ to the server, the server will encrypt $B \,||\, \text{secert}$. This will be grouped into $[B \,||\, \text{secret}_0], [\text{secret}_{1-16}], [\text{secret}_{17-32}], [\text{secret}_{33-47} \,||\, \text{pad}]$, and then xor with the corresponding IV to do the *block cipher encryption*. Therefore, the first block indeed satisfies the condition.




(How to get the ciphertext of 15 known plaintext bytes)

After we have successfully recovered the first byte of the secret, we can consider the recovered as known bytes and send 14 bytes to the server to try recovering the second secret byte. By repeating this attack 48 times, we can recover all the 48 bytes of the secret and then go to get the 🚩. On average, it requires about $128 \times 48 = 6144$ quries. We set the alarm time as 1200 seconds (20 minutes). It's sufficient to complete the exploit. 🤔

---

The source code of the exploit is provided as below (written in Python 3.8):

```python
# !/usr/bin/env python3
import re, string
from hashlib import sha256
from itertools import product

from pwn import *

def xor(a, b):
    return bytes(x^y for x, y in zip(a, b))

def hex2bytes(data):
    return bytes.fromhex(data.decode())


r = remote("127.0.0.1", 10000)
# context.log_level = 'debug'

# PoW
```

```python
r.recvlines(13) # banner
rec = r.recvline().decode()
suffix = re.findall(r'XXXX\+([^\)]+)', rec)[0]
digest = re.findall(r'== ([^\n]+)', rec)[0]
print(f"suffix: {suffix} \ndigest: {digest}")
print('Calculating hash...')
for i in product(string.ascii_letters + string.digits, repeat=4):
    prefix = ''.join(i)
    guess = prefix + suffix
    if sha256(guess.encode()).hexdigest() == digest:
        print(guess)
        break
r.sendafter(b'Give me XXXX: ', prefix.encode())

# Attack
rec = r.recvline().decode()
IV_hex = re.findall(r'([0-9a-f]{32})', rec)[0]
IV = bytes.fromhex(IV_hex)

def getLastBit(known, cipher, IV):
    """
    known: first 15 bytes that we know

    cipher: Ek(known + ?) where ? is one byte that we want to get

    returns (?, IV)
    """
    for i in range(256):
        r.sendlineafter(b"> ", b"1")
        payload = xor(IV, known + bytes([i]))  # Ek(known + i) where len(known) = 15, len(i) = 1
        r.sendlineafter(b"(in hex): ", payload.hex().encode())

        rec = r.recvline(keepends=False)
        IV = hex2bytes(rec[-16*2:])
        if hex2bytes(rec[:16*2]) == cipher:
            return bytes([i]), IV
    return None

# Recover byte by byte.
recovered = b""
for k in range(4):
    # k=0: secret[0:15]   l={15, 14, ..., 1}
    # k=1: secret[15:31]  l={16, 15, ..., 1}
    # k=2: secret[31:47]  l={16, 15, ..., 1}
    # k=3: secret[47:48]  l={16}
    start = 15 if k == 0 else 16
    end   = 15 if k == 3 else 0
    for l in range(start, end, -1):
        r.sendlineafter(b"> ", b"1")
        r.sendlineafter(b"(in hex): ", IV[:l].hex().encode())
        rec = hex2bytes(r.recvline(keepends=False))
```

```
        if k == 0:
            known = b"\x00"*l + xor(IV[l:-1], recovered)
            last_byte = IV[-1:]
            cipher, IV = rec[:16], rec[-16:]
        else:
            known_IV, cipher, IV = rec[16*(k-1):16*k], rec[16*k:16*(k+1)],
rec[-16:]
            known = xor(known_IV, recovered[-15:])
            last_byte = known_IV[-1:]

        byte, IV = getLastBit(known, cipher, IV)
        recovered += xor(byte, last_byte)
        print(recovered.hex(), len(recovered))

# Get flag.
r.sendlineafter(b"> ", b"2")
r.sendlineafter(b"(in hex): ", recovered.hex().encode())
print(r.recvline(keepends=False))
# b'TQL!!! Here is your flag:
WMCTF{Dont_ever_tell_anybody_anything___If_you_do__you_start_missing_everybody}'

r.close()
```