AES-CBC Data Tampering (“Bit Flipping”)

CryptoHack “Flipping Cookie” Writeup

February 23, 2024

AES - A Refresher

AES is a well-known symmetric cipher, encrypting blocks of 16-bytes using keys of length 128, 192 or 256. It is trusted to be a secure cipher, protecting the data transmitted over an insecure channel from a 3rd party.

Modes of Operation - ECB

There are many ways to use AES, from just a classic block cipher all the way to using it as a stream cipher.

The most basic mode of AES is ECB (Electronic codebook), which, given a key, encrypts every 16 bytes to the same 16 byte block.

In summary, given a plaintext p, key k and e_k, the encryption function of AES-ECB for a key k, we know that: $c = e_k(p)$.

Since this is the result every time p is encrypted. This allows an attacker to easily identify patterns in the encrypted data.

Modes of Operation - CBC

CBC differs from ECB by providing **diffusion.** In CBC, the ciphertext is starts from $C_0 = IV$ where IV (or **Initialization Vector**) are “random” 16 bytes (it does not have to actually be random to provide diffusion, just not repeating).

Then, for every $P_1, P_2, \ldots, P_i$, we encrypt using $C_i = e_k(P_i \oplus C_{i-1})$.

This means that $d_k(C_i) = d_k\left(e_k(P_i \oplus C_{i-1})\right) = P_i \oplus C_{i-1}$, so to decrypt, we XOR with the previous ciphertext again.

Hence:

$$d_k(C_i) \oplus C_{i-1} = P_i \oplus C_{i-1} \oplus C_{i-1} = P_i$$

Flipping Cookie

You can get a cookie for my website, but it won't help you read the flag... I think.

```python
from Crypto.Cipher import AES
import os
from Crypto.Util.Padding import pad, unpad
from datetime import datetime, timedelta


KEY = ?
FLAG = ?


@chal.route('/flipping_cookie/check_admin/<cookie>/<iv>/')
def check_admin(cookie, iv):
    cookie = bytes.fromhex(cookie)
    iv = bytes.fromhex(iv)

    try:
        cipher = AES.new(KEY, AES.MODE_CBC, iv)
        decrypted = cipher.decrypt(cookie)
        unpadded = unpad(decrypted, 16)
    except ValueError as e:
        return {"error": str(e)}

    if b"admin=True" in unpadded.split(b";"):
        return {"flag": FLAG}
    else:
        return {"error": "Only admin can read the flag"}


@chal.route('/flipping_cookie/get_cookie/')
def get_cookie():
    expires_at = (datetime.today() + timedelta(days=1)).strftime("%s")
    cookie = f"admin=False;expiry={expires_at}".encode()

    iv = os.urandom(16)
    padded = pad(cookie, 16)
    cipher = AES.new(KEY, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(padded)
    ciphertext = iv.hex() + encrypted.hex()

    return {"cookie": ciphertext}
```

Let's asses the situation:

- The cipher used is AES-CBC.
- We have a cookie for a website, containing a date of expiry and user permissions.
- Randomized IV and unknown Key, correct Padding.
- We are allowed to decrypt whatever we want, but we can only encrypt the cookie.
- We also can't see the result of decryption.

Some XORing

Let's say we can decrypt, and we have a part of the ciphertext, similar to the conditions we have in the challenge (but a simpler encryption).

- Assume P is an 8-letter string, that starts with $"hello"$, and $P \oplus IV$ happens (we know the IV, since it's a public property).
- We want the decrypted ciphertext to contain the word $"admin"$. (which is impossible without tampering)
- We can provide C and IV to the decryptor, which will then do $C \oplus IV = P \oplus IV \oplus IV = P$ to obtain P again.

From the structure of AES-CBC, we know that changing a bit in the IV will alter the first decrypted plaintext block.

How can we use this fact? Let's see, we will create $IV_2 = IV \oplus "hello" \oplus "admin"$.

(In words, let's convert "hello" and "admin" to bytes and xor the first 5 bytes of IV with them).

Now the decryptor will do:

$P^{'} = C \oplus IV_2 = C \oplus IV \oplus "helllo" \oplus "admin" = "hello..." \oplus "hello" \oplus "admin" = "admin..."$

See what happened?

1. $C \oplus IV$ retrieved the original $P = "hello..."$
2. We then XOR'd the first 5 bytes of P with $"hello"$, creating 5 bytes filled with 0 (since $x \oplus x = 0$)
3. We then XOR'd the firtst 5 bytes of the result from (2) with $"admin"$, causing the first 5 bytes to be $"admin"$ (since $x \oplus 0 = x$)

So we forced $"admin"$ to appear on the plaintext string!

Solving The Flipping Cookie

Let’s get a random cookie using the API.

```
OUTPUT
{"cookie":"edd2553317f67b5b6a9843c3970a10752483c5de09129100db629163c54729a5951485f993a7a42fc548cac
569f52ad8"}
```

And split it to blocks, using Python.

```
>>> [b[i:i+32] for i in range(0, len(b), 32)]
['edd2553317f67b5b6a9843c3970a1075', '2483c5de09129100db629163c54729a5', '951485f993a7a42fc548cac569f52ad8']
```

We know:

- $IV = \boxed{'edd2553317f67b5b6a9843c3970a1075'}$
- $mal = \boxed{"admin = False" \oplus "admin = True;" =' 000000000000121319165e'}$

Let’s set our malicious IV to be $IV_2 = IV \oplus mal$, so $\boxed{IV_2 =' edd2553317f66948738e1dc3970a1075'}$.

We know that $"admin = False"$ is in the 2nd of the three blocks, since it’s in the start of the ciphertext and is smaller than 16 bytes, so it fits in a single block. Consider $P = "admin = False; expiry = 29485923852"$ as an example to what’s about to happen.

We know that $C_1 = e_k(P \oplus C_0) = e_k(P \oplus IV)$, so $d_k(C_1) \oplus IV = (P \oplus IV) \oplus IV = P$,

So if we change the decryption IV to be IV_2, we get:

$$d_k(C_1) \oplus IV_2 = (P \oplus IV) \oplus IV \oplus "admin = False" \oplus "admin = True;"$$

$$= P \oplus "admin = False" \oplus "admin = True;"$$

$$= "admin = False; expiry = 29485923852" \oplus "admin = False" \oplus "admin = True;"$$

$$\boxed{= "admin = True;; expiry = 29485923852"}$$

So, in order to pwn the challenge we need to $d(cookie, IV_2)$.

Let's submit IV_2 as the IV, and the following 2 blocks as the cookie:

And now we have the flipping cookie!