Nicola Mores, 31/03/2024

Challenge: AESWT PoC 2

Background

This challenge is about Cryptographic Failures [1] and CBC, an AES (Advanced Encryption Standard) block cipher mode, as an authentication mechanism. Being a block cipher, this symmetric cryptography algorithm works on blocks of fixed size, applying padding in order to expand the last block whenever needed. The main idea behind this cipher mode is XOR-ing every i^{th}-plaintext block with the $i\text{-}1^{th}$-ciphertext block, and then encrypt it with the key, obtaining a new ciphertext block. To apply this rule in every block we also introduce an Initialization Vector (IV), used during the computation of the 1^{st}-ciphertext block as replacement for the 0^{th}-ciphertext block that otherwise would be missing. While CBC mode is considered more secure than others such as ECB (due to its ability to obscure plaintext patterns), it may still be vulnerable to certain attacks, including Padding Oracle attacks and Malleability attacks. The latter is exploited in this challenge to obtain the flag. Malleability is a (usually) undesirable property of cryptosystem that allows attackers to modify the content of the decrypted plaintext, violating the integrity of the encrypted message.

Vulnerability

In this case the vulnerability arises from the malleability of CBC that can be exploited in order to modify the plaintext obtained from the server after the encryption, allowing us to bypass the sanitization, in order to create a token the blacklisted word “admin”.

Solution

In order to complete this challenge we need to deeply understand the encryption/decryption schema of CBC and also notice how the implementation of the access control system works in this challenge. Starting with the latter, we can notice how the system generates a new token, using the method `signup()`, creating a string made of the provided description, followed by the username. The two arguments are arranged in a key-value format (like a dictionary or a map), dividing each entry with a `&`. In the same way, the method `information_schema` uses this format in order to fetch the two main components (description and user) from the decrypted token. Talking about the CBC, we need to notice how it is possible to modify one of the ciphertext blocks in order to get a desired plaintext block after the decryption, using this formula: $C'_{i-1} = P_i \oplus C_{i-1} \oplus P'_i$, where i is the index of the block we want to modify, P stands for plaintext, C for ciphertext and the ‘ means modified block [2]. Applying this formula we can properly modify the i^{th}-block of plaintext, but we compromise the previous blocks, since every ciphertext block originates from the corresponding plaintext block and on the previous ciphertext block. Normally the junk plaintext generated should prevent a malleability attack, but in our case we can use the key-value format in order to store it away, so that we can effectively produce the two components as needed in order to get the flag. The following code [3] show how we can implement the attack, showing an example of the input and also how to modify the ciphertext:

```python
full_token = signup("?????........!!!!!!!!I am a boss", "........")     # 1st parameter is the username, 2nd is the description
# With this input we only need to properly change the 5th, 2nd and 1st block.
# The last 2 blocks should be unchanged, while the 3rd and 4th will get dirty without compromising the attack

# Setup Attack
initial_plaintext = ("desc=........&user=?????........!!!!!!!!I am a boss").encode().hex()
# desc=... - .....&us - er=????? - ........ - !!!!!!!! - I am a b - oss         P  Blocks
final_plaintext =   ("user=admin&********_______*******&desc=I am a boss").encode().hex()
# user=adm - in&***** - ***_____ -  ___***** - **&desc= - I am a b - oss         P' Blocks

token_parts = [full_token[i:i+16]for i in range(0, len(full_token), 16)]                                 # Divide in blokcs
initial_plaintext_parts = [initial_plaintext[i:i+16]for i in range(0, len(initial_plaintext), 16)]
initial_plaintext_parts.insert(0, token_parts[0])                                                       # Add IV
final_plaintext_parts = [final_plaintext[i:i+16]for i in range(0, len(final_plaintext), 16)]
final_plaintext_parts.insert(0, token_parts[0])                                                         # Add IV

# Modify Chipertext
final_chipertext_parts = [None] * (len(token_parts))
for i in range(len(token_parts)):
    final_chipertext_parts[i] = token_parts[i]

i = 4       # i = 4 will change the plaintext block containing !!!!!!!!
final_chipertext_parts[i] = xor(xor(initial_plaintext_parts[i+1], token_parts[i]), final_plaintext_parts[i+1])
i = 1       # i = 4 will change the plaintext block containing .....&us
final_chipertext_parts[i] = xor(xor(initial_plaintext_parts[i+1], token_parts[i]), final_plaintext_parts[i+1])
i = 0       # i = 4 will change the plaintext block containing desc=...
final_chipertext_parts[i] = xor(xor(initial_plaintext_parts[i+1], token_parts[i]), final_plaintext_parts[i+1])
```

With a similar code we can obtain a new token that once decoded will allow us to successfully obtain the flag for this challengeReferences

[1] OWASP Cryptographic Failures:
https://owasp.org/Top10/A02_2021-Cryptographic_Failures/
[2] Paper about CBC vulnerabilities. Notice the chapter 3.4 about Malleability attack:
https://www.usenix.org/system/files/sec20fall_ender_prepub.pdf
[3] Code Example:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment3/solution2.py