POODLE Attack CTF Challenge

Yonatan Goldenberg
206786568

Abstract

This document aims to give a deeper understanding of how the POODLE attack works and to describe how the CTF challenge was made and what it aims to achieve. In the first part, we will go over the attack to better understand how it really works, and then we will move on to see the structure of the challenge. Please note that it is recommended to read this document after you've completed the challenge since the detailed explanation of the attack will give you the solution right away.

1 Introduction

In October of 2014 Google has disclosed an attack that targeted a vulnerability in the SSL 3.0 protocol. SSL 3.0 is an old version of a cryptographic protocol that was used to encrypt web traffic. SSL 3.0 was released in 1996 and later replaced by TLS with it's later versions. TLS was a safer protocol but despite TLS appearing since 1999 many servers still supported SSL 3.0 out of legacy support reasons for many years, and the POODLE used to exploit it. SSL 3.0 was officially declared deprecated in 2015 shortly after Google's report.

2 Related Work

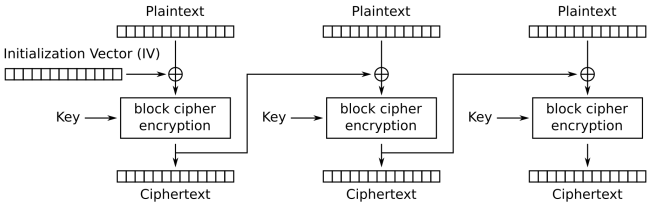
To better understand how the attack and challenge work, it is best to have prior knowledge in web communications, specifically what is HTTPS and SSL, what is a TLS handshake, and how raw requests look like during the communication. Additionally it is very important to know basic cryptography and to know how CBC encryptions and MAC work.

3 Attack description

The POODLE (Padded Oracle On Downgraded Legacy Encryption) is a type attack that is aimed to decrypt the encrypted traffic between a server and a user. It exploits a vulnerability in the SSL 3.0 where it didn't verify the padding added during CBC encryption, and thus the attacker could enter any type of information he wanted in that section and the server wont notice.

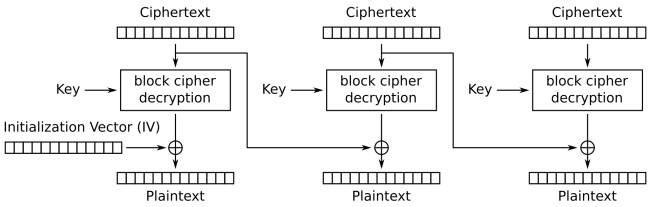
The attack itself had prior steps before it could make use of this vulnerability. The attacker had to first create a situation where he can sniff all the traffic between the user and the server (MITM). Next, he would have to inject a script to the user and force him to run it. The script will now send requests to the server on behalf of the user. The attacker will now aim to downgrade the encryption protocol down to SSL 3.0. He can do that by dropping the connection repeatedly during the TLS handshake stage. This will make the server think that the user cannot support the current protocol and he will downgrade back to SSL 3.0 in case the server still has support for legacy encryptions.

Now that all those steps were achieved the attacker can start attacking the encryption protocol itself. At this part it is important to give a refresher on how SSL 3.0 and CBC encryption work. When a request is ready to be sent through the web it will need to be encrypted. The SSL protocol will first generate a MAC from the request and append it to the end of it. Next he will add padding so the request fits the block size of the CBC encryption. When CBC encrypts messages it divides the message to block, usually of 16 or 8 bytes. It will then encrypt each block using a secret key (the encryption method itself is not important but it is usually AES) but before encrypting the block he will XOR him with the encrypted bytes of the previous block. This creates a chain where each block of bytes, after it is encrypted, is used to XOR with the next block before it is being encrypted. The first block of bytes does not have a prior block so he is being XORed with a random block of bytes called an initialization vector (iv).



Cipher Block Chaining (CBC) mode encryption

Decryption of this protocol works the same way in reverse. Each block is first decrypted and then XORed with the previous block of encrypted bytes. To decrypt a message you would have to own the encryption key and the iv.



Cipher Block Chaining (CBC) mode decryption

The padding that the SSL protocol adds to the message is there so the message would be dividable evenly to blocks for the CBC. The bytes of the padding are not relevant and are either random or a repetition of the last byte of the padding. The last byte of the padding is not random, it is meant to specify how much padding was added. So when the message is decrypted you can know where does the padding end and the MAC begins. After the addition of the MAC and the padding the entire message is then encrypted and sent off.

When the receiver of an encrypted message wishes to decrypt it he will first use the CBC key and iv he possesses to decrypt the entire string of bytes. He will then look at the last byte to see how much long is the padding and remove the specified amount. If the last byte is a number bigger than the block size a padding error will occur and the message will be dropped. After the padding is removed the next block is the MAC of the rest of the message. The

receiver will create a MAC of his own of the rest of the message using an agreed set of key and iv (should be different from the CBC key and iv for better security) and compare it to the MAC extracted from the decrypted message. If the MACs are the same the message is accepted, if there is a mismatch it means that the message was tampered with and the message will again be dropped.
Looking back from the attackers perspective, he can use the fact that the padding bytes are not checked and are ignored combined with the fact that the last byte is not random. The padding length can be between 1 and a whole block's length in case the original message is already dividable by the block size. And so the attacker will aim to create a message that has a full block of padding attached to it. He can easily do so by forcing the victim to send requests each with one byte longer body and sniff the encrypted packet sent. At some point the length of the packet will jump since the padding will get smaller with each request until the padding wont be needed and the protocol would just attach an additional full block of padding. Now the attacker has a full free block at the end of the request to work with knowing that the last byte in that block should be 16 (or 8 depending on the block size of encryption) for the message to pass the server check.
The attacker can now take any block from the encrypted request and place it instead of the padding block at the end. When that block is decrypted there is 1/256 chance of the last byte to be decrypted to 16 (complete random chance). And if by random chance this happens, the message will be accepted by the server since he will just skip to the beginning of the MAC, which is left unchanged, and verify the message. If the last byte is not right, the server will drop the connection and the attacker can monitor when that happens. By seeing when does the server accept the request or when does he drop the connection the attacker can understand if the last byte was of correct block size. This means that the attack can now calculate what the last byte of the original block really is.
We have an encrypted message of n blocks. For the next section we define P_i to be the original content of the i'th block in the message, meaning this is the original text of the block before it was XORed and encrypted. C_i is the encrypted form of the i'th block, these are the blocks that make up the encrypted message. And we will define $dec(C_i)$ as the function that decrypts the i'th block. This is only the decryption method, before the decrypted block will be XORed with the previous block.
The attacker took block number i and placed it in as the last block and the message was accepted by the server. This means we now have this equation (x are random bytes):

$$dec(C_i) \oplus C_{n-1} = x|x|x|x|x|x|x|x|x|x|x|x|x|x|16$$

$$dec(C_i) = C_{n-1} \oplus x|x|x|x|x|x|x|x|x|x|x|x|x|x|16$$

We know from the way that CBC works that:

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

So now we can write:

$$P_i \oplus C_{i-1} = C_{n-1} \oplus x|x|x|x|x|x|x|x|x|x|x|x|x|x|16$$

$$P_i = C_{i-1} \oplus C_{n-1} \oplus x|x|x|x|x|x|x|x|x|x|x|x|x|x|16$$

The last byte of the block can be retrieved by performing the XOR only on the last byte, since we know all the bytes that need to be XORed:

$$P_i[15] = C_{i-1}[15] \oplus C_{n-1}[15] \oplus 16$$

The attacker managed to find an original byte in the request before it was encrypted. By changing the URL and body lengths of the request the attacker can control the disposition of the message between the blocks and that way can control what byte will be at an end of a block. This way an attacker can reveal each block one byte at a time. Each byte will take the attack an average of 256 requests until the request is accepted but after enough requests the attacker can reveal most of the message and get sensitive information from it like cookies.

4 Design

The project is designed to test your understanding of the attack, specifically the cryptographic vulnerability part. That is why the parts of MITM, script injection and protocol downgrade are disregarded. The project is divided into two folders, one for the attacker and one for the server and victim, and is written entirely in Python. In order to recreate the SSL protocol a module file was created that uses pycryptodome library for AES-CBC encryption. This module file is use by both the front-end and back-end of the challenge to mimic how SSL 3.0 really worked.
The server only offers the minimum for the attack. He first generates a session cookie and two sets of 16 byte key and iv for the encryption and MAC. Then the victim makes a request to get the cookie simulating a login and thus the challenge begins. The sever will verify the session cookie in every request (besides the login that can be called only once) and will decline requests that don't have it. Additionally the server has a root route, where all the request are sent during the attack and a keys route, used by the victim for encrypting messages since after a failed decryption the server will create a new set of keys and ivs to simulate a TLS connection drop. The last route is called jokingly get-private-data and is meant to verify your victory in the end of the challenge.
The front side of the challenge simulates a victim using an API that communicates with the sever. The API offers a get-request function that takes a string for the path of the request and a string for its body and returns the encrypted bytes of the request before it is sent. This can be used to study how the encryption works and modify the requests. Next the API offers a send-request function which simply takes a set of bytes and sends it to the server through the victim script. That way the request is sent with the session cookie and server will try to decrypt the bytes. This function returns the server's response.
This challenge was designed to run using Docker for easy set up and to ensure a closed environment where the project would always work. That is why there is a **Dockerfile** in each folder and a **docker-compose.yml** in the main folder. When this challenge is run it creates a container for the server and a container for the attacker and victim together. Then both containers communicate through a docker network created for them to ensure isolation.

5 CTF instructions

To complete the challenge you will have to find the session cookie of the victim and verify that you can now communicate with the server as the victim. You will need to use the API provided by the

victim_api.py module to decrypt the cookie. You can only write code in the **attacker.py** file in the client folder. You cannot add or change any imports in that file, you can only write in the marked areas in the main function and add helper functions if needed.
It is recommended to run the project with **docker-compose -d –build**, as it will build the entire project and run automatically. This command will create images for the server and client and run containers based on them in a private networks that is also created. To stop the running containers run **docker compose down** in the root folder of the project. You can view the logs of each container with **docker logs client_container** or **docker logs server_container**, and if you want a more comfortable UI you can use Docker Desktop.
The challenge will complete once you can call the **check_solution** function, provided by the victim, and get the server to accept the request that is sent with your cookie.

6 Conclusion

If you completed this challenge, first of all congratulations. The jump from understanding to implementing the attack is not small. Second, this should give you some perspective on how creative and sharp one must be to break a security protocol this big.
Security is a highly important part in the advancement of technology in our age. Keeping communications secure is a never ending mission since new vulnerabilities constantly appear.
I hope you also enjoyed the challenge and took the opportunity to learn from it. This is a small example of a much deeper world of cryptography and web security, which are constantly expanding fields with great importance to our world today. With the growth of these fields more vulnerabilities like this will be sure to appear, and with them more insight to better our privacy.