BEAST CTF User-guide

Yuval Lihod ,2024

Part 1: Background

BEAST stands for Browser Exploit Against SSL/TLS. Those are cryptographic protocol designed to encrypt communications and it used in the HTTPS protocol.

The BEAST (CVE-2011-3389) is an attack that allow a MITM attacker to uncover information from an encrypted SSL/TLS 1.0 (I will refer them simply as TLS from now and on) session by exploiting a known theoretical vulnerability in the way that TLS protocol generated Initialization Vectors (IV) in CBC mode.

Combined with some clever manipulation of block boundaries, the flaw allowed the MITM attacker to discover small amounts of information without performing any decryption.

The exploit worked regardless of the type or strength of the block cipher used.

Before diving into the attack, let's start with some background information about symmetric cryptography using block ciphers:

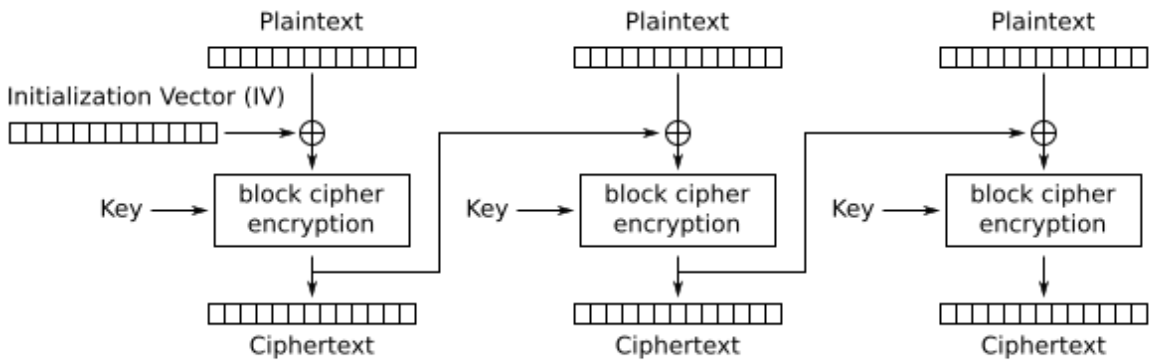
- **AES:** Block cipher - a deterministic algorithm that operates on fixed-length groups of bits, called blocks. Block size of AES is 16.

- **Initialization Vector(IV):** An input to a cryptographic primitive being used to provide the initial state. The IV is typically required to be random or pseudorandom, but sometimes an IV only needs to be unpredictable.

  The BEAST attack utilize the fact that it IV is predictable in TLS 1.0 and SSL versions.

- **Mode of operation:** An algorithm that designed to allow securely transform amounts of data larger than a block, using block cipher.

- **Cipher Block Chaining(CBC):** Mode of operation. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. To make each message unique, an IV must be used in the first block.

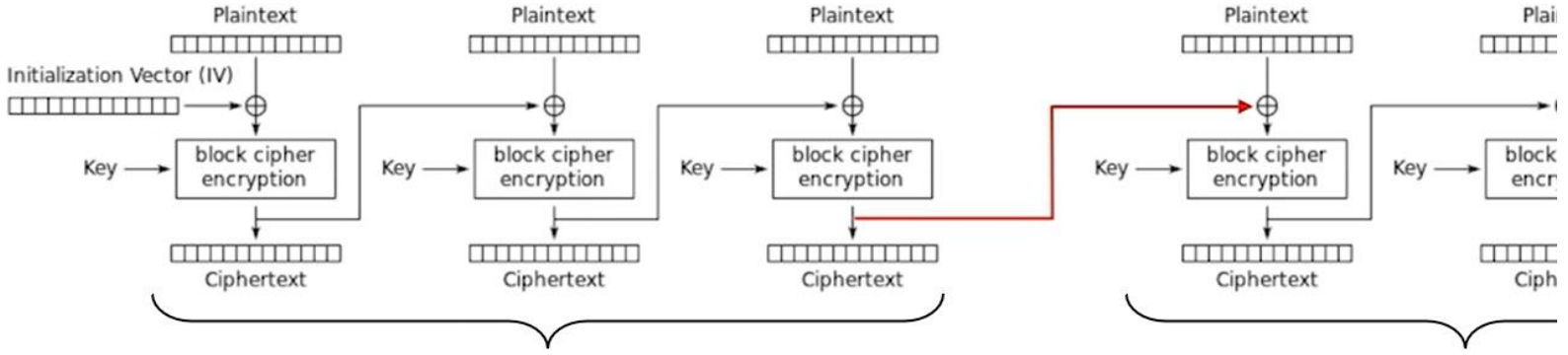


Cipher Block Chaining (CBC) mode encryption

The TLS(1.0) standard mandates the use of the CBC mode encryption with chained IVs;

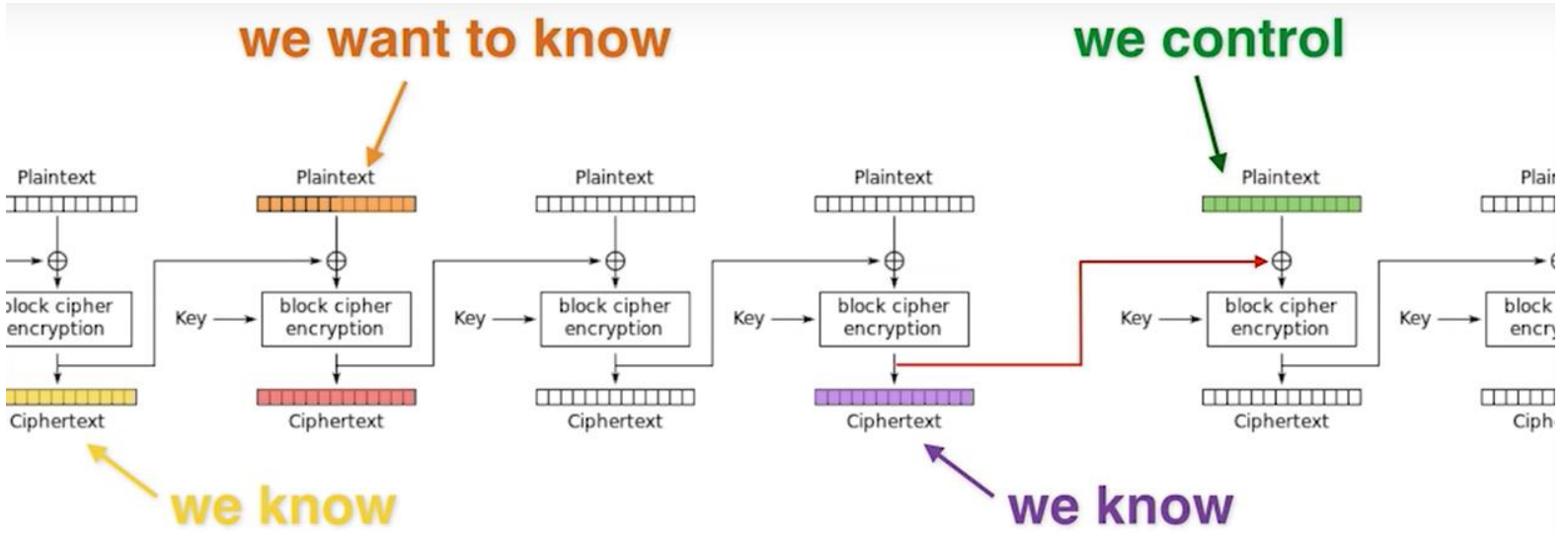
i.e., the IV used when encrypting a message should be the last block of the previous ciphertext.

For the first block of the first message random IV will be generated.



Now let's say we want to decrypt a specific block which contains the victim's cookie (orange), so that can impersonate that person. We eavesdrop to the communication between the victim and his bank server, so all we can see is the encrypted messages (yellow, red, purple). What can we do?

The last encrypted block (purple) will be the next IV for the next message, so if we can control the **first** following block to be encrypted, we can discover some information.



As we want to know the orange plaintext, we will set the green plaintext as follows:

If we took the right guess the input (plaintext) to the AES block cipher is the same, and since AES is deterministic it will yield the same output (ciphertext). Otherwise – we will change our guess.

To take another guess we need the victim to encrypt the same message again. Notice that we need 2 things: **1)** To make the victim send a new request.

2) It has to be the the same request.

For the second thing, as HTTP request have strict format headers it won't be a problem, and it won't be very difficult to guess which block (or blocks) is the cookie block.

The attacker will accomplish the first by taking advantage of vulnerabilities in HTTP Request APIs in Java Applet (or JavaScript) by injecting the victim's browser Java Applet while he accesses the attacker's malicious website. The same mechanism will allow him also to control the following message. I won't elaborate on this subject.

Great, all we need now it to guess 16 bytes (128 bits), there is only 2^{128} options for that.

Well, we can't really, it's too much. We need to design the request in such a way that we know most of the plaintext in that block (15 bytes out of 16 to be precise). How? The Java Applet also allow us to control the URL path of the request.

Let's see an example. In this example we will use DES instead of AES for convenience only (DES has block size of 8 bytes versus 16 bytes of AES). The attack is the same for both block ciphers.

Consider a fairly standard HTTP request:

```
GET /index.html HTTP/1.1
Host: mysite.com
Cookie: Session=12345678
Accept-Encoding: text/html
Accept-Charset: utf-8
```

Parts of this request are required by the protocol to be in specific places, and many other parts are easily guessable.

Here, the only really variable part of the request are the page being requested and the value of the session cookie.

Now, let's say that, in CBC mode, the first 78 bytes of your plaintext request are:

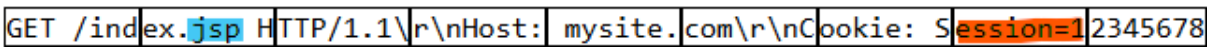
```
GET /index.html HTTP/1.1\r\nHost: www.somesite.com\r\nCookie: Session=12345678
```

DES encrypted in CBC mode, using a key of *password* and an IV of 0x0123456789abcdef, this becomes:

GET /ind	ex.html	HTTP/1.1	\r\nHost	: mysite	.com\r\n	Cookie:	Session=	12345678
a7d25abbd67b2dbf	e2ade7246ea5ed5	e99063ebe430b75b	746ae5eca36e2bc3	f6f62f99a076056f	6b14704973a779ae	7fa9300d4e490cba	b6040b9542a59ad5	f9bb888ac3763722b

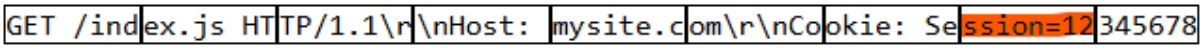
The hypothetical attacker would be interested in the last block, `f9bb888ac3763722b` — remember that the nature of HTTP makes it relatively straightforward for him to determine exactly what block or blocks contain the sensitive information, especially in the case of session cookies. Now he only can go over all the 2^{64} options for that block. Too much.

Remember he can also make modifications to the request as well. Lest say he did this:



Here, I've removed one character from the page being requested to index.jsp (turquoise), but I've also split the victim block across an 8-byte boundary. Now, if I want to find the first digit of the session ID, I only need a maximum of 256 guesses (1 byte) because I know that the plaintext is 'ession=?', where '?' is unknown byte. This can be done easily.

After finding the first byte of the session cookie, I can alter the request again:



I know that the plaintext is now 'ssion=1?'. I will continue in the same way until whole cookie is found.

References:

https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art027

https://www.youtube.com/watch?v=-_8-2pDFvmg

Part 2: CTF instructions

Overview

There are 2 CTF challenges: **Beginner** and **Advanced**. Each challenge created by removing some code parts of the E2E 'Attacker' implementation, where the Advanced challenge created by removing more parts than the Beginner. Thus, you should start with the Advanced challenge, and only if you can't solve it try the easier 'Beginner' challenge.

Instructions

```
#Complete missing parts below
iv = ?
cookie_cipher = ?
prev_cipher = ?
```

1. Complete the missing code parts denoted in ' ?'. Notice that each ' ?' might be more than 1 line of code.

2. Run the script. If it prints the client's cookie, you have exploited the vulnerability and retrieved the flag.

Hints

```
def request_func(step):
    """
      Construct the current file_path
    """
    #Complete missing part below
    #Hint: step = len(attacker_cookie)
    ?
```

Any hints, if exist, will appear in a remark adjacent to the relevant missing code part.

How to run

- **Prerequisites:** Docker must be installed on your system.
- Use 4 different terminals for the bank server, malicious server, client and attacker (this is you).
- Build and run the Bank server

1. Navigate to the bank directory:

```
cd bank
```

2. Build the Docker image for the bank server:

```
docker build -t server_image .
```

3. Run the bank server container:

```
docker run --rm -t --network=host server_image
```

- Build and run the Malicious server

1. Navigate to the malicious server directory:

```
cd malicious_web
```

2. Build the Docker image for the malicious server:

```
docker build -t mal_server_image .
```

3. Run the malicious server container:

```
docker run --rm -t --network=host mal_server_image
```

- Build and run the Attacker (your implementation)

1. Navigate to the relevant CTF challenge directory (beginner or advanced).

2. Build the Docker image for the attacker:

```
docker build -t attacker_ctf_image .
```

3. Run the CTF attacker container:

```
docker run --rm -t --network=host attacker_ctf_image
```

- Build and run the Client

1. Navigate to the client directory:

```
cd client
```

2. Build the Docker image for the client:

```
docker build -t client_image .
```

3. Run the client container:

```
docker run --rm -t --network=host client_image
```