Assignment 3 Report

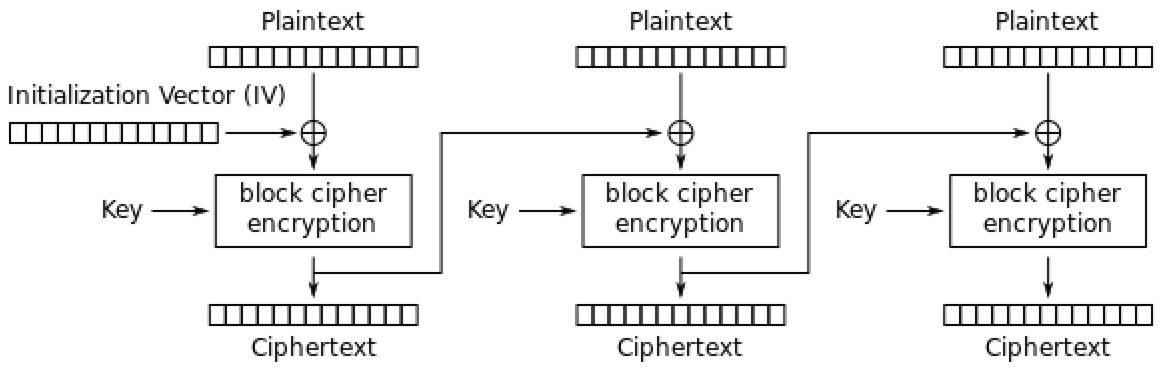
BEAST On CBC Ciphers in TLS 1.0

Protocol - TLS 1.0

TLSv1.0 is a successor to SSLv3.0 which is a cryptographic protocol to encrypt information and is used in the HTTPS protocol. TLS has a handshaking procedure before it starts transmitting data. The handshake is encrypted using public key encryption whereas the actual data is encrypted using symmetric encryption. The Handshaking procedure has mainly 4 steps between the client and server.

- The Client gives the server a list of ciphers / hash functions it can support and the server chooses one of them, usually the most secure one which it can support.
- The Server replies with its public key signed by a trusted CA
- The Client confirms the validity of the key.
- The Client uses Diffie-Helman key exchange protocol using the server's public key to exchange the symmetric key to be used for that particular session.

If any of the above steps fail the handshake is stopped and the connection is refused.

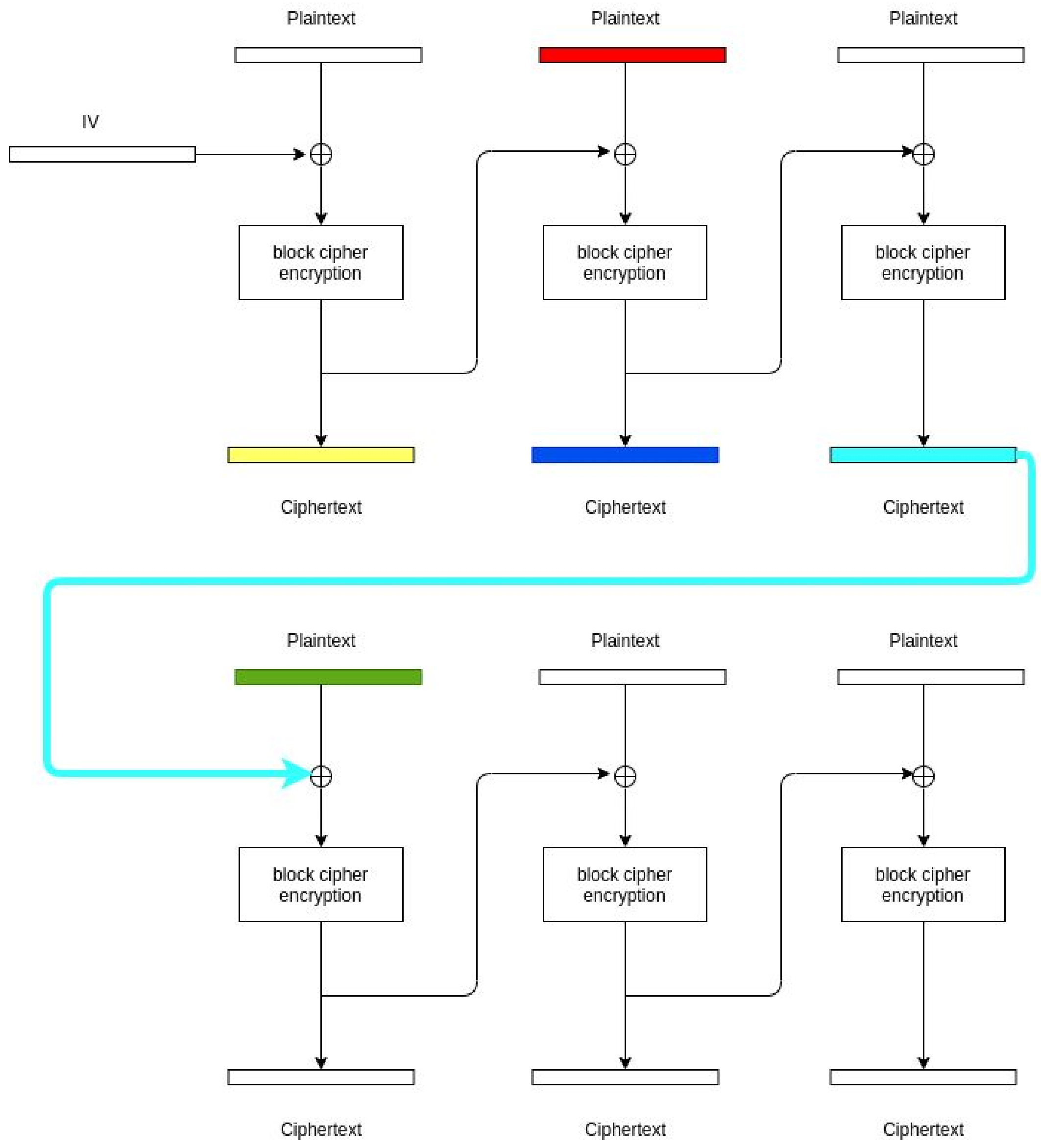


Cipher Block Chaining (CBC) mode encryption

Vulnerability Explanation

As you can see in the above diagram of the block cipher, when the plain text is divided into blocks and encrypted they are XOR'd with the previous block's ciphertext. For the first block since there is no cipher text a random vector known as Initialisation Vector(IV) is used.

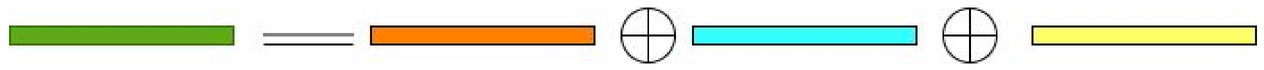
The main vulnerability lies not in the exact CBC cipher, but how the CBC cipher is implemented in TLS1.0. In TLS 1.0, the IV of the next message is not randomly generated and is taken from the previous message's ciphertext as you can see in the diagram below.



Since the IV of the new message is the last block of the previous ciphertext, we can guess parts of the previous message by controlling the first block of the plain text of the next message. For this we need to know the information sent to the secure website through the HTTP requests. Since HTTP requests have such strict format headers, it isn't very difficult to guess this. After we know some sensitive information is being stored in the red block(session cookies), the attacker needs to guess what is stored in it, so that we can impersonate a user with it.

Another important part of this attack is that the attacker needs to be able to control the plaintext to be able to change it. To accomplish this the attacker takes advantage of vulnerabilities in HTTP Request APIs in JavaScript or Java Applets to send malicious requests and change part of the requests. But we ignore this part in our simulation.

Now that we know we need to guess the red block. We do this



The orange block is the guess of the red block. Once the guess is correct and becomes the red block, the resulting ciphertext block becomes the blue block and we know our guess is correct.

But we cannot cycle through all possibilities because they are quite large. So we need to design the request in such a way that we know most of the plaintext in the red block and there is only a single letter of the sensitive info in the block.

```
GET /index.jsp HTTP/1.1\r\nHost: mybank.com\r\nCookie: Session=12345678
```

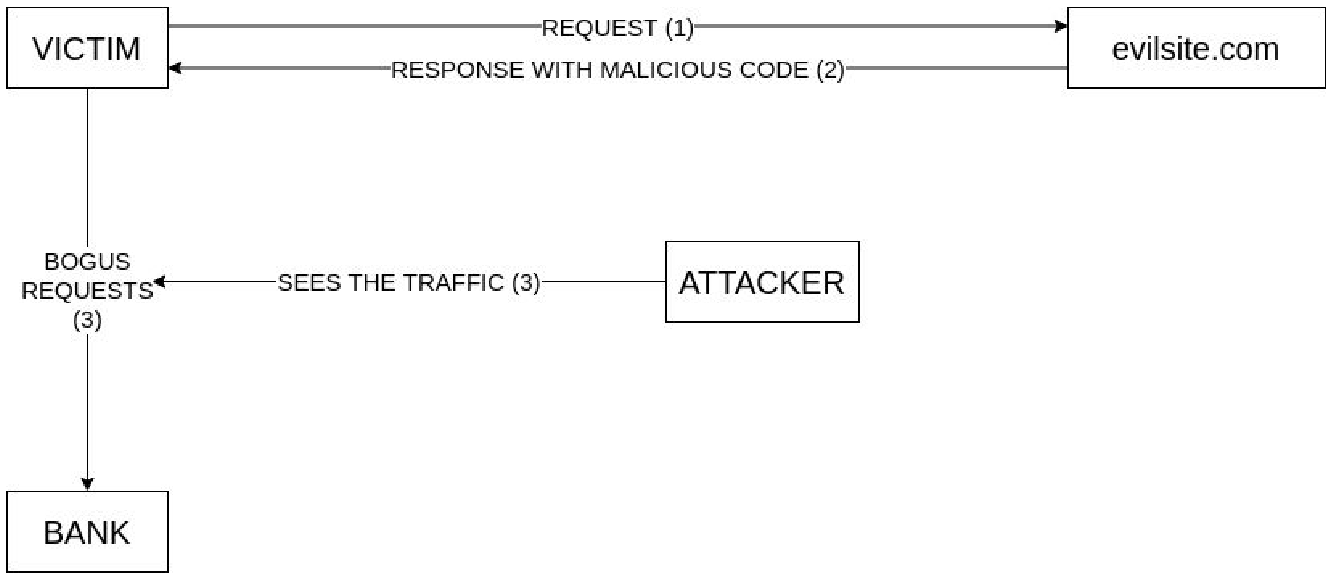
```
|GET /ind|ex.jsp H|TTP/1.1\|r\nHost:| mybank.|com\r\nC|ookie: S|ession=1|2345678|
```

The red text contains 1 character of the session ID. So to guess this the malicious script first sends this request to get the cyan and yellow and blue blocks. Once the malicious script knows these blocks. We already know most of the plaintext of the red block. Now the script creates green blocks for each guess and sends these malicious requests. Now the attacker who is the man in the middle goes through all these encrypted malicious requests and once he sees that a character's guess matches the blue block, he takes note of the character and repeats this by decreasing the requests' length by 1 each time till he guesses all characters.

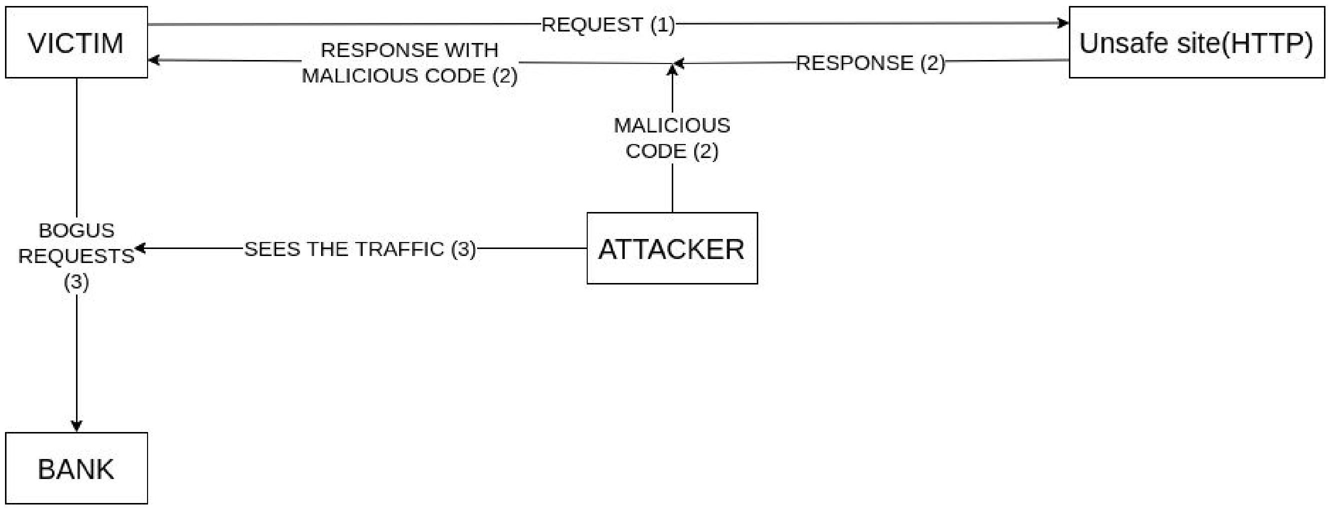
Attack Implementation

The attack starts with the installation of malicious JavaScript code or Java Code which takes advantage of vulnerabilities in HTML5 WebSocket API or Java URLConnection API respectively to modify the plaintext.

For this there are 2 methods:
In the first method the client visits a malicious website which puts malicious javascript code in the response and it executes on the client computer and this malicious code sends all the bogus requests and you listen to the encrypted traffic and deduce the sensitive data. The numbers in the text represent the order in which the steps occur.

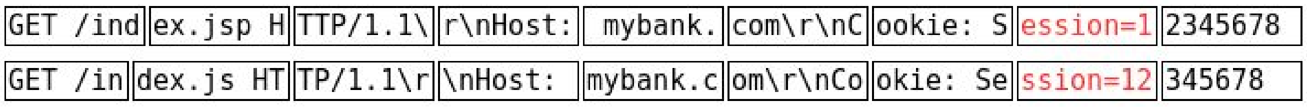


In the second method you are the man in the middle and wait for the victim to visit a HTTP website. Since the traffic is unencrypted, you can easily inject your malicious code in the response of the HTTP website. After the malicious code is injected the rest of the steps are the same as above.



So now we see what the malicious code sends exactly so that the attacker deduces the cookies. First the request is formulated in such a way that we know that a single byte of the information is in that block. After formulating that request the request is sent. Now the attacker keeps track of the blue block in the above part. Now the malicious code copies the block of the sensitive information and changes the last character and calculates the XOR of the 3 blocks like in the previous section. He changes the last character 10 times and sends 10 bogus requests. The attacker can now deduce the first character.

Now the malicious code reduces 1 byte in the request so that another byte of the sensitive information falls in the red block as shown below



GET /ind	ex.jsp H	TTP/1.1\	r\nHost:	mybank.	com\r\nC	ookie: S	ession=1	2345678
GET /in	dex.js HT	TP/1.1\r	\nHost:	mybank.c	om\r\nCo	okie: Se	ssion=12	345678

Now the same process is repeated. Remember that the code copies the red block directly, changes the **last** character and calculates the XOR. With this the attacker can keep deducing a single character until he deduces the entire cookie. The malicious code should know the restrictions on the cookie, i.e. its length and the character set (only digits, alphanumeric, ascii 0-256).

Code Explanation and Relaxations

In our simulation we have 4 classes. Victim, Attacker, Bank and CryptoHelper. The first 3 are pretty self explanatory. The CryptoHelper class contains the main cryptographic functions of CBC encrypt and decrypt, padding and unpadding functions, XOR function for byte blocks. The Bank class only uses the decrypt function whereas the encrypt and XOR functions are used by the Victim to simulate the malicious code. The cookie in this case is 8 byte alphanumeric. So each byte would have 62 guesses(26 + 26 + 10).

As for the socket arrangement there will be 2 server sockets one in Attacker and one in the Bank and 2 client sockets, i.e. one in the Attacker and one in the Victim. The attacker will directly relay the information it receives from the Victim to the Bank as it is the man in the middle and can't meddle with encrypted data.

In the Victim class there is a main loop which sends each request after removing a byte and for each request it simulates the malicious code by sending all guesses after the XOR function with the guess. This part is done by the *modify* function. There are a couple of helper functions known as *request* which gives the request after removing the required no of bytes and *block_print* which prints the modified plaintext in blocks of 16(with padding).

In the Attacker class there is only a main loop function which receives all encoded requests and takes note of all the necessary blocks and compares it with the modified requests' blocks to guess each byte. All the requests are naturally forwarded to the Bank socket.

In the Bank class the requests are decoded and the requests which have been modified by the XOR function are deemed invalid, and the ones which have been byte shifted are deemed not found because the path is a bogus path which doesn't exist.