# Extra Topics

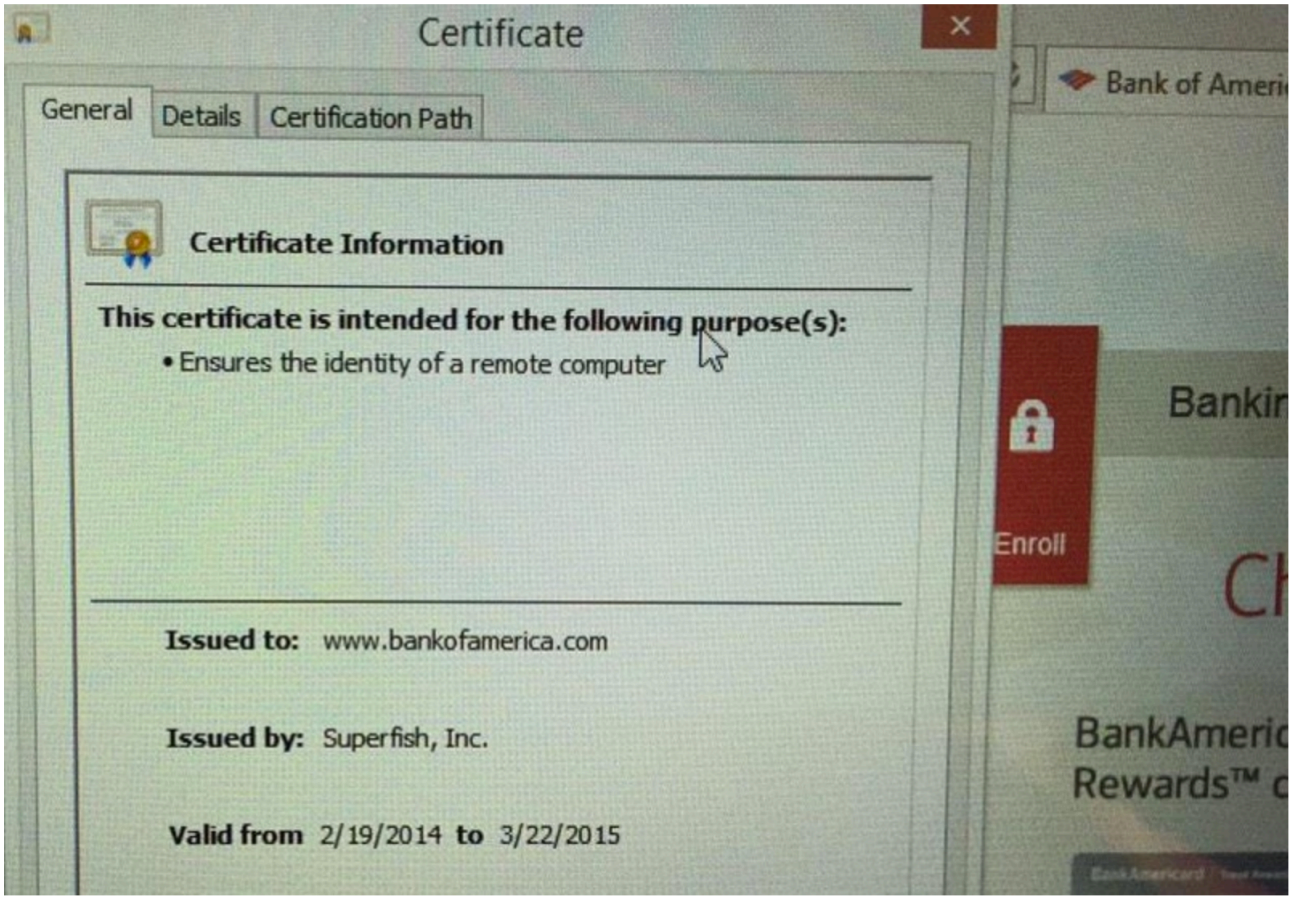
## SuperFish attack

**WHAT IS SUPERFISH?**
A program that effectively conducts a man-in-the-middle (MITM) attack. It sits between communication between any server, intercepts all communication to insert adverts for the user.

Superfish installs a self-signed root certificate that can intercept all HTTPS encrypted traffic (for every website you visit). Encrypted traffic is intercepted to inject advertisements.

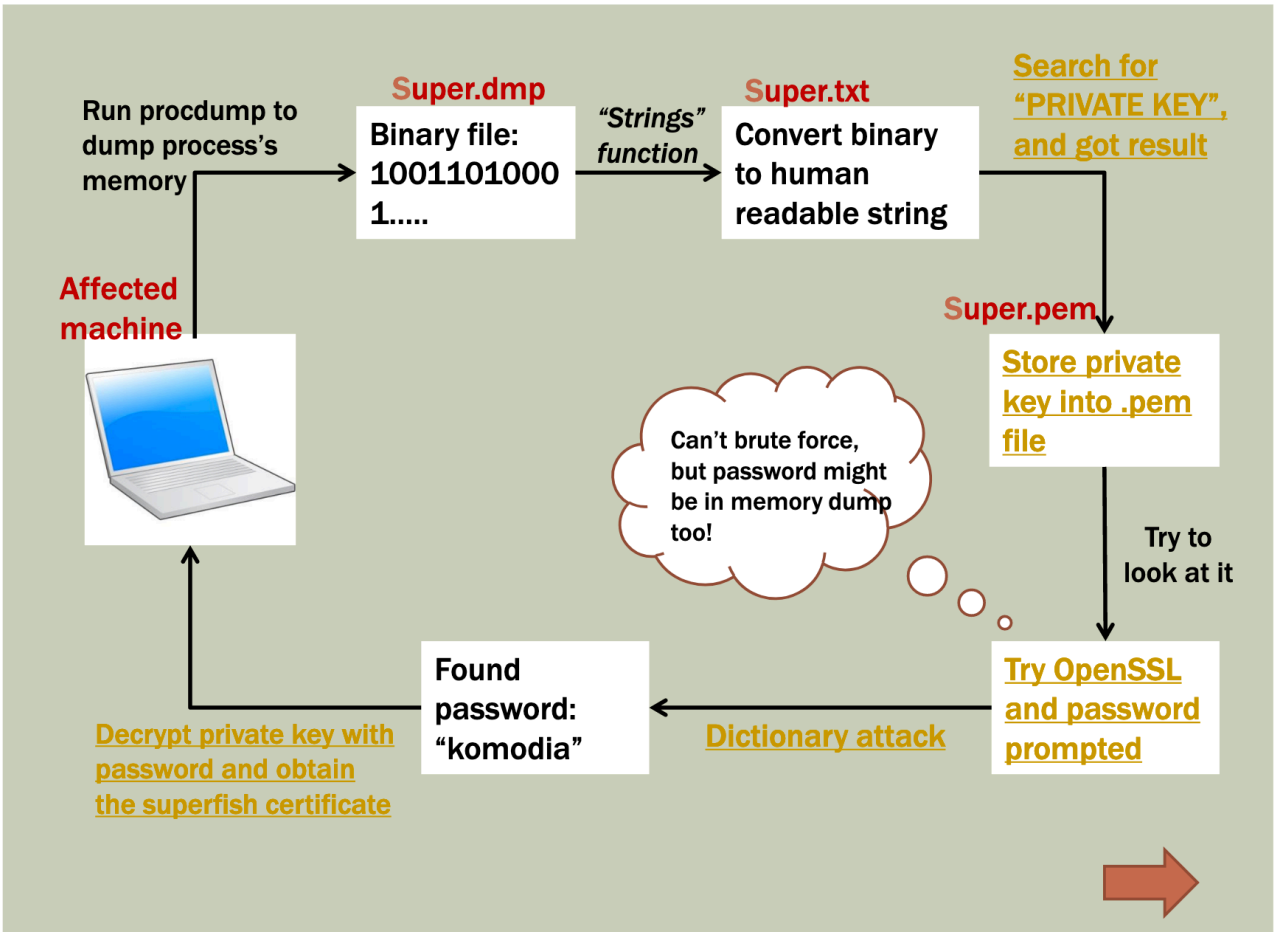
**WHAT IS THE SIGNIFICANCE?**
If you have SuperFish installed, you really can't trust secure connections to sites anymore.

For instance, take a look at the fake certificate affected Lenovo users see when they visit the Bank of America website. It is signed by Superfish, not Bank of America.




Users don't know that certificates have been messed with / replaced. Superfish seems to appear to use the same certificate for all installs.

**HOW CAN PRIVATE KEY BE OBTAINED?**
Likely to be stored within the machine itself. Use procdump / opjdump / debugger to view the memory accessed by SuperFish. Convert to String to get human readable format. Obtain private key by doing a search. The private key is likely to be encrypted. However, the password for encryption is also likely to be stored in the procdump of the file! You can dictionary search all the strings that you find in the procdump. Once decryption successful, can obtain the private key of Superfish!




**WHAT CAN WE DO WITH THE PRIVATE KEY?**
With the private key, we can issue a "fake" certificate to the web server that the victim is accessing.

Suppose Alice wants to access IVLE. Bob notices that Alice is using a Lenovo laptop and is vulnerable to this attack. Suppose Mallory already has access to the private key, he can:

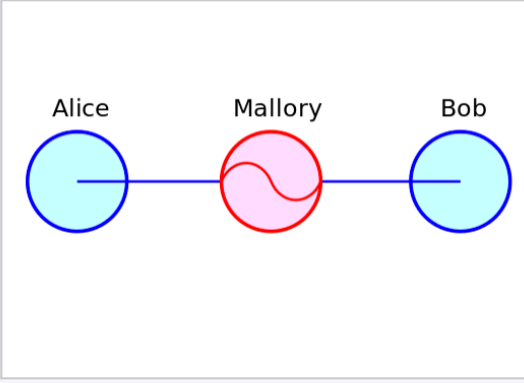
1. Issue a "fake certificate" to another server that looks like IVLE, issued by SuperFish (signed by private key). The associated domain name in this "fake certificate" is ivle.nus.edu.sg.
2. In this certificate, Mallory states the identity and a public key that he has generated. Mallory keeps the associated private key secret.
3. Mallory then tricks Alice into heading into his server. This can be done by sending a phishing email, doing a DNS spoofing attack etc.
4. Once Alice reaches Mallory's website, the browser obtains the certificate of Mallory's server.
5. Since Alice's computer recognises SuperFish as a CA, the browser trusts this certificate received during the authentication process. At this point, Alice thinks that she has secure connection to IVLE, but is connected to Mallory.
6. When Alice attempts to enter her login credentials, Mallory can obtain it from the server in plaintext.

Note: This attack may not work on secure Wireless networks (with WPA2 protection, since each connection is encrypted with a  secret key that Mallory would not know this secret key)

# THE MAN IN THE MIDDLE ATTACK, EXPLAINED

Suppose Alice wishes to communicate with Bob. Meanwhile, Mallory wishes to intercept the conversation to eavesdrop and optionally to deliver a false message to Bob.

First, Alice asks Bob for his public key. If Bob sends his public key to Alice, but Mallory is able to intercept it, an MITM attack can begin. Mallory sends a forged message to Alice that purports to come from Bob, but instead includes Mallory's public key.



An illustration of the man-in-the-middle attack

Alice, believing this public key to be Bob's, encrypts her message with Mallory's key and sends the enciphered message back to Bob. Mallory again intercepts, deciphers the message using her private key, possibly alters it if she wants, and re-enciphers it using the public key she intercepted from Bob when he originally tried to send it to Alice. When Bob receives the newly enciphered message, he believes it came from Alice.

1. Alice sends a message to Bob, which is intercepted by Mallory: Alice *"Hi Bob, it's Alice. Give me your key."* → Mallory Bob
2. Mallory relays this message to Bob; Bob cannot tell it is not really from Alice: Alice Mallory *"Hi Bob, it's Alice. Give me your key."* → Bob
3. Bob responds with his encryption key: Alice Mallory ← *[Bob's key]* Bob
4. Mallory replaces Bob's key with her own, and relays this to Alice, claiming that it is Bob's key: Alice ← *[Mallory's key]* Mallory Bob
5. Alice encrypts a message with what she believes to be Bob's key, thinking that only Bob can read it: Alice *"Meet me at the bus stop!" [encrypted with Mallory's key]* → Mallory Bob
6. However, because it was actually encrypted with Mallory's key, Mallory can decrypt it, read it, modify it (if desired), re-encrypt with Bob's key, and forward it to Bob: Alice Mallory *"Meet me at the van down by the river!" [encrypted with Bob's key]* → Bob
7. Bob thinks that this message is a secure communication from Alice.

This example[4][5] shows the need for Alice and Bob to have some way to ensure that they are truly each using each other's public keys, rather than the public key of an attacker. Otherwise, such attacks are generally possible, in principle, against any message sent using public-key technology. A variety of techniques can help defend against MITM attacks.

# HeartBleed attacks

**WHAT IS THE HEARTBLEED BUG**

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

**WHAT KINDS OF INFORMATION CAN BE LEAKED?**

**WHAT IS THE HEARTBEAT PROTOCOL?**
Heartbeat is an echo functionality where either side (client or server) requests that a number of bytes of data that it sends to the other side be echoed back.

**WHAT IS THE PURPOSE OF THE HEARTBEAT PROTOCOL?**
The idea appears to be that this can be used as a keep-alive feature, with the echo functionality presumably meant to allow verifying that both ends continue to correctly handle encryption and decryption.

It is normally used to check the health of a machine. It is important to verify that the transport that you have to that machine is as reliable as possible. (Or if the machine is done, to take the necessary action).

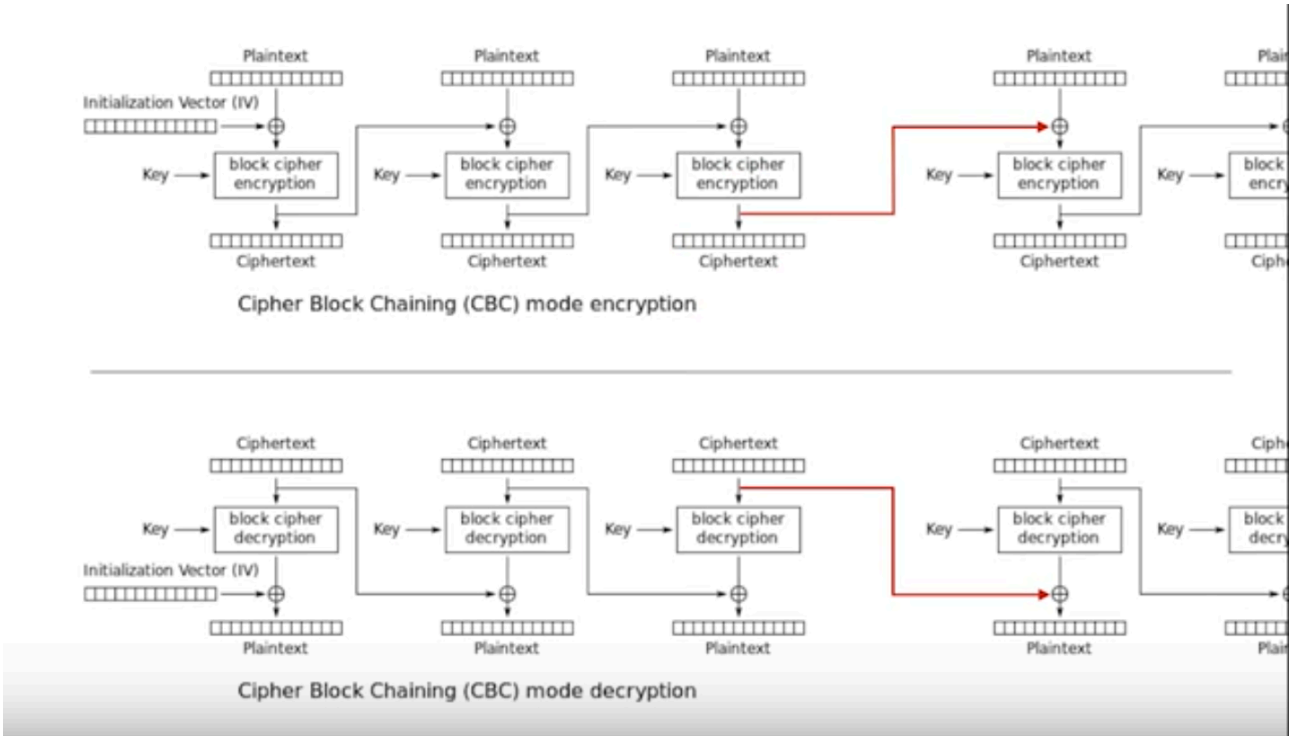
**HOW CAN SOMEONE ATTACK A SERVER WITH AN OUT-OF-DATE VERSION OF OPENSSL?**
1. Manipulate the heartbeat protocol
2. Send malicious packet. In the packet header, state that the packet length 64KB (max possible packet length). However, in the payload, only append a small number of bytes (e.g 10 bytes)
3. The receiver then receives the packet, and copies the specified number of bytes from the packet. Therefore, if the size is greater than the actual payload, the server then fills the response with data that is in contagious memory to the stored region of the payload received.
4. This data can contain sensitive information, like passwords, usernames, or even private keys.
5. With the leaked private keys, MITM attacks can be conducted.

**WHAT CAN WE DO WITH THE PRIVATE KEYS?**
1. Obtain the session key
    1. Suppose Alice and Bob want to authenticate one another. They each take turns to authenticate one another unilaterally, using the public keys infrastructure.
    2. At the end of authentication, there are 2 nonces produced, which are the keys used for encryption and MAC respectively.
    3. With the leaked private key, Mallory can decrypt the session keys established with the server, and obtain the session keys.
2. With the session keys, Mallory can decrypt all communication, and can also maliciously hijack existing communication by performing a MITM attack.

# Beast Attacks

**ATTACK ON CBC IN TLS**

**WHAT WAS THE CAUSE OF THE BUG?**

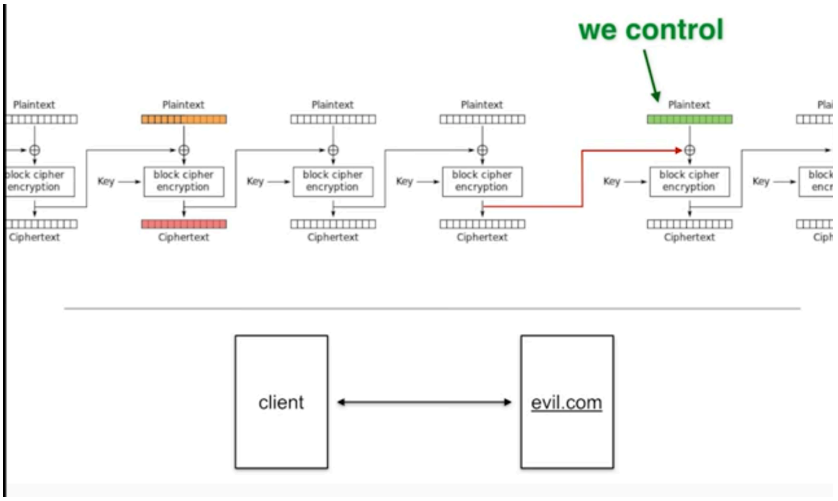



As we know, CBC uses the previous block of **ciphertext** to XOR with the **current block** of **plaintext** for encryption.

The problem with the SSL encryption was that to encrypt a **new block**, instead of randomly generating a **new IV**, the previous ciphertext gets used for encryption. Hence, it is possible to obtain the IV for decryption.

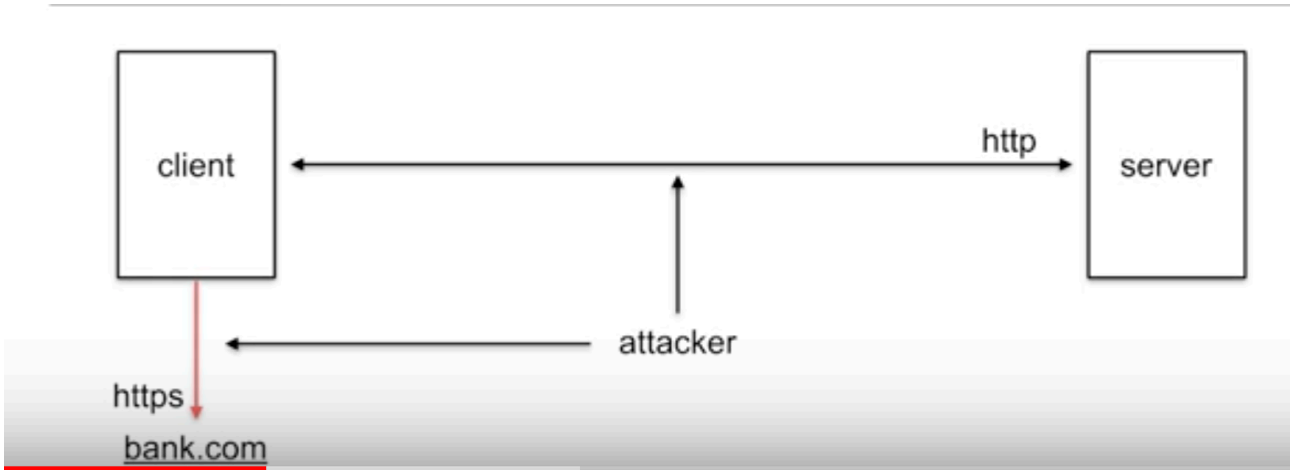
**HOW CAN WE CONDUCT A BEAST ATTACK?**

1. Suppose Alice wants to connect to a web server. (Mallory can inject javascript into this web server so that it gets executed when the client visits the website)
2. Mallory, a man-in-the-middle, observes the **encrypted ciphertext** and observes that there is something important. e.g session id, cookie, password etc
3. Control the plaintext. E.g this can be done by injecting some JavaScript etc. Suppose when Alice visits Mallory's website, JavaScript can be inserted to make some queries to securesite.com. With this, cookies can be observed and can try to decrypt these cookies when you are MITM between client and server.

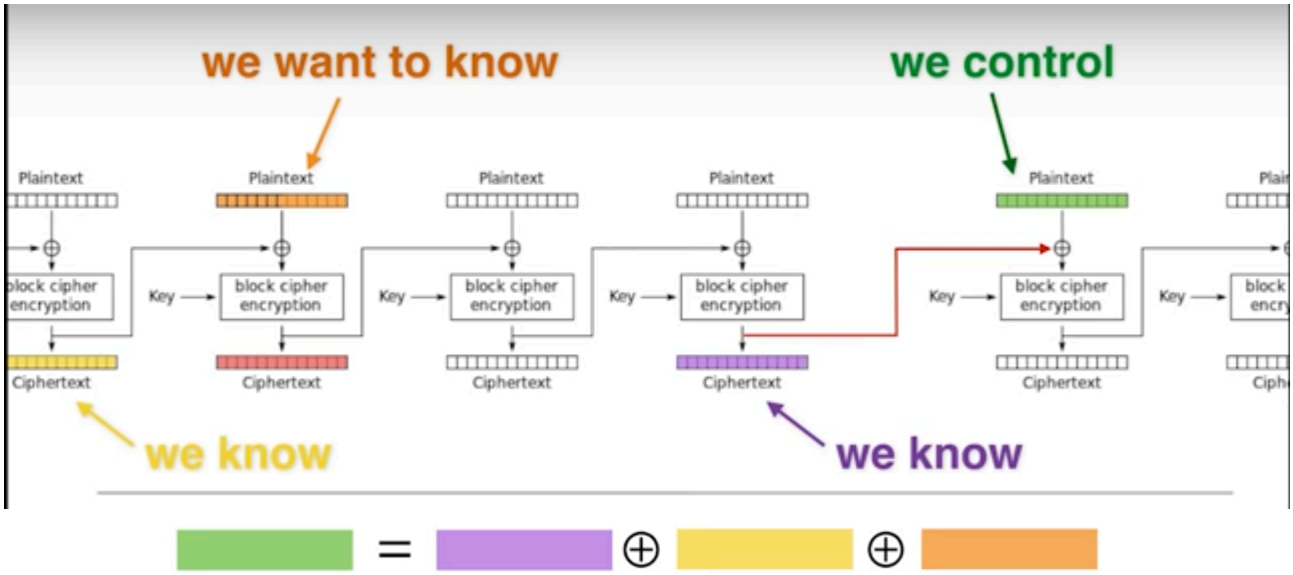



5. The easiest way for Mallory to inject JavaScript would be to wait for Alice to visit a website in **http**, and inject some **JavaScript** inside. This

JavaScript then makes some queries to another secure site, and you can control these queries (you know what the plaintext is).




6. Therefore, since the bug reuses the previous block of ciphertext as the IV, we have the plaintext and the previous ciphertext. We also know some of the ciphertext we have observed (e.g cookie, password etc) that were sent previously.

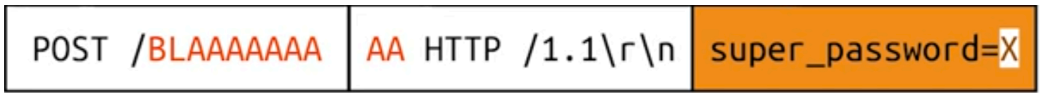



7. Mallory can then set the next plaintext as shown in the diagram above. When she does this, she gets back the exact data that was being encrypted previously. Note that the purple region will get cancelled out, because of it gets XOR'ed out.

8. Mallory can then observe the result of the decryption, to see if it matches the original ciphertext shown in yellow. This can tell Mallory if the chosen plaintext (orange) was correct.

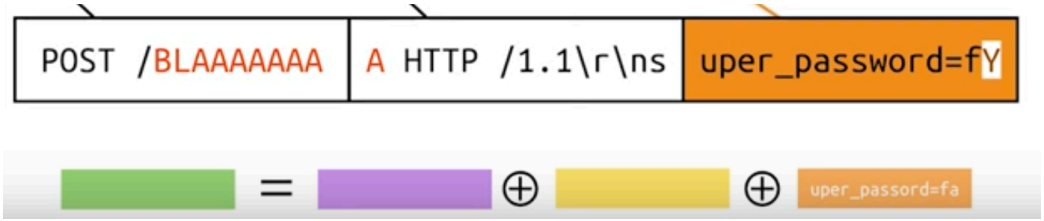
9. Brute force of the plaintext would be hard, as if AES-256 is being used there are $2^{256}$ possibilities, which is hard.

10. Mallory can make it easier by making most of the plaintext with null. This can be done by sending the request as follows:




11. Here, there is only a single byte of data in the plaintext. This can easily be guessed. So, how to decrypt the next byte?

12. The next byte can be decrypted by “pushing down” one byte, as follows:




13. This process can be repeated till the entire secret-password is obtained.

# Hash-length extension attacks

**WHAT IS IT?**
A vulnerability that arises when hashes are used to authenticate a user, instead of a HMAC. For example, let's say there is a secret key p that is shared between Alice and a server. When Alice sends a message p||m, the server verifies Alice by computing SHA-256(p||m) to see if it matches the server's record. If it does, it authenticates Alice.

**WHAT IS THE PROBLEM WITH THIS IMPLEMENTATION?**
Someone who does not know the key can be authenticated by **forging the MAC**.

**WHEN DOES THIS VULNERABILITY APPEAR?**
When hashes are used to authenticate a user.

**HOW TO AVOID THE HASH-LENGTH EXTENSION ATTACK?**
Use a HMAC for authentication, never simply rely on a hash for authentication.

**HOW DOES IT WORK?**
It works by forging the state of the registers that have been used for the previous SHA computation. By doing this, we can compute the hash as if the original message was the extended message.

**HOW TO EXPLOIT**
Suppose you already know the **length** of the secret key, that is used to append to the message prior to the hash being tabulated. Suppose you also know the **hash digest** of the message, and the **data**.

1. Since hash is computed in blocks (usually 256 or 512 bits), add the necessary padding to complete the first block (i.e key||message||padding).
2. Subsequently, concatenate the next message that needs to be extended. This message

# Padding Oracle Attack

**WHAT IS THE PADDING ORACLE?**

When it comes to CBC-mode encryption, encryption and decryption is done in blocks. Since each encrypted block will be XOR'ed with the next block, it is important to ensure that each block is of the same size.

To ensure this, all blocks will have some form of padding (in bytes, until it reaches a size which is an integer multiple of the required blocks size). *Note that even if the block is full without any padding, a full block-length worth of padding is added to ensure that the receiver will be able to separate the padding from the actual message accurately. Note that the trailing bytes of each block also contain the number of bytes of padding. For example, if there are 6 bytes of padding, the final 6 bytes should state 06 each.*

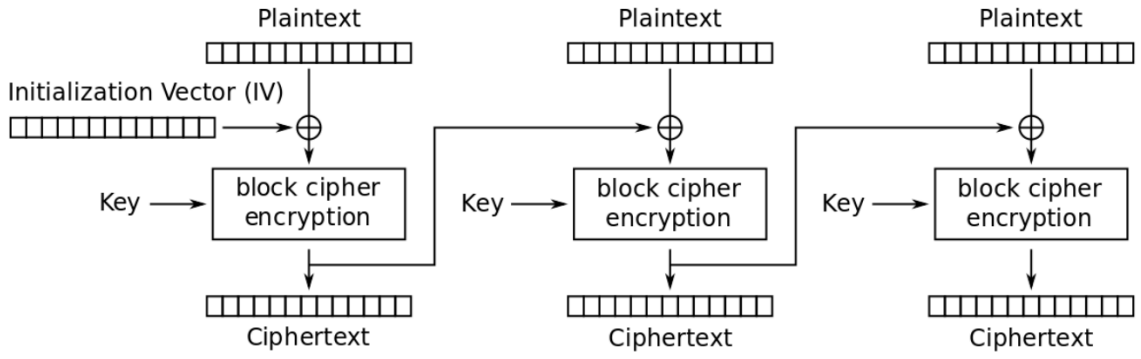
Therefore, when the receiver receives an encrypted message to decrypt, it needs to find a way to respond to the sender on whether or not the decryption was successful. *This is important, especially if it is a web server, because packets can get corrupted during transmission over to the server. Hence, the server can subsequently send an error message of some sort, indicating that decryption was not successful.*

This behaviour of the server can then be manipulated, as we can now black-box this mechanism into what we call a *padding oracle*, which basically tells us whether the specified padding in an encrypted message is correct or not.

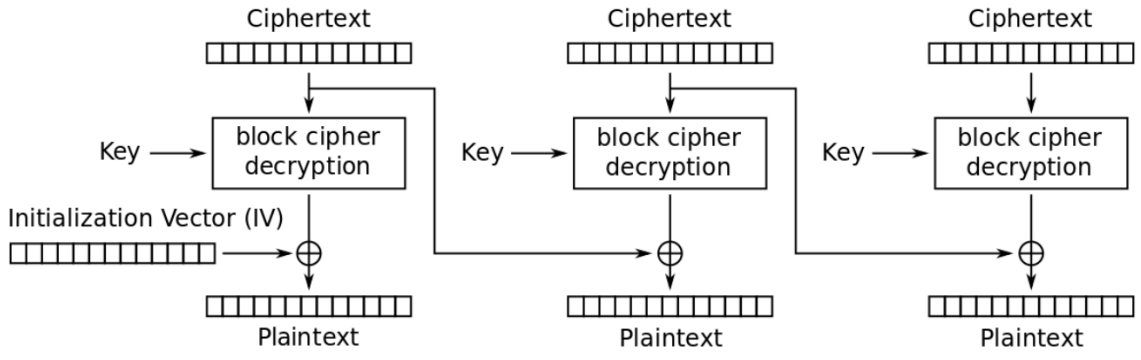
**HOW DOES THIS HELP US?**

Recall that under the CBC mode of **decryption**, the previous block of ciphertext is used to XOR with the current block of encrypted text, the previous block of ciphertext is XOR'ed against the current block of plaintext.

**THE KEY OBSERVATION**




Cipher Block Chaining (CBC) mode encryption



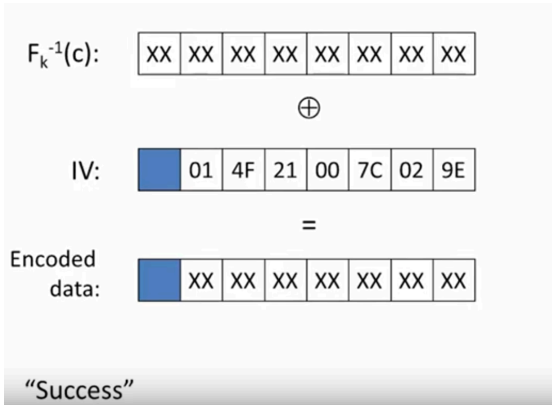

Cipher Block Chaining (CBC) mode decryption

The attacker has access to the IV that is XOR'ed with each

**THE OBSERVATION**

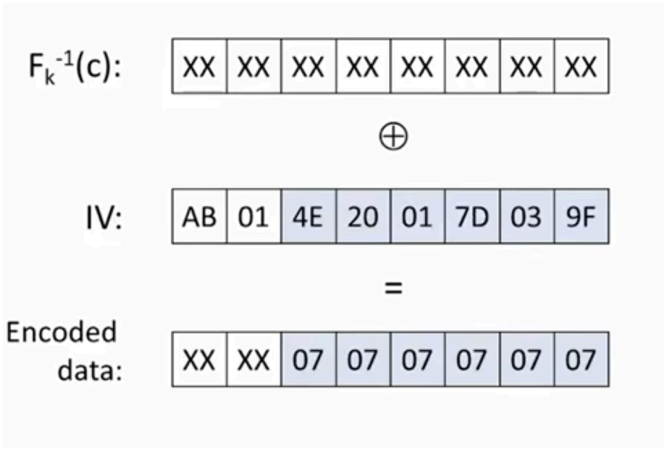
Note that in this attack, the attacker has access the previous block of ciphertext (IV) that is used for decryption. The attacker can then take advantage of this in 2 ways:

1. Figure out how many bytes of padding is done for each block of ciphertext.
    1. This can be done by manually moving byte by byte by changing the IV from the first byte down. The byte in which we get an error message is the beginning of the first byte of padding.

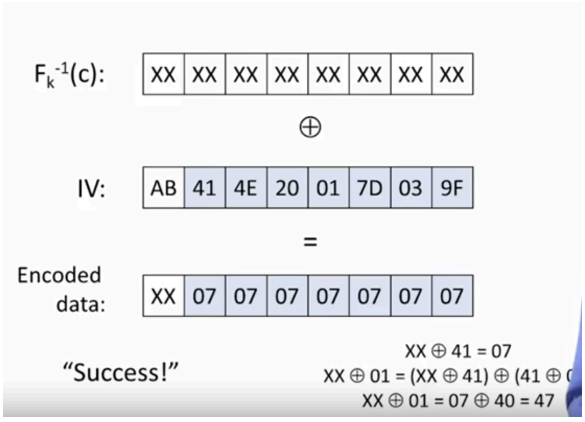



For example, if the 3rd byte of the changed IV and causes an error, the length is 2 bytes.

2. Manually change the IV byte by byte from the back, such that you increment the size of padding by one. This can be done with a simple xor operation - e.g 9E xor 07 xor 06 to change encoding to 07.




3. Once you reach the block with the actual data, manually try all possibilities of the byte until the decryption is successful. You can then find out the original byte, by inverting the xor operation as follows:




e.g if the IV is 41 and the resulting decryption is successful, we know that XX xor 41 = 07

Then xor with the actual byte of the intercepted ciphertext (in this example, it was one). We now know the ciphertext of the 2nd byte transmitted by the sender.

This process can be repeated to find the first byte of the IV, and then can be used to decrypt the encoded data.

# SSL/TLS Renegotiation attacks

**WHAT IS A BRIEF SUMMARY OF THIS ATTACK?**

1. Client is connected to a server. They are communicating and there is a need to establish a new connection in the midst of the ongoing connection.
    1. This could happen for a couple of reasons:
        1. E.g on online marketplace web app, and the website uses HTTPS to secure connection. After initial handshake has happened, you go on to browse the page for a while and you add some items to cart and want to check out. The site may re-negotiate a new SSL connection to ensure that you have a secure connection before you key in sensitive data like your credit-card info.
        2. E.g a new encryption key is needed in the midst of a secure connection that has already been established / new hash that needs to be generated.
2. Attacker first connects to the TLS server.
3. Attacker hijacks the connection by proxying client traffic
4. Client attempts to renegotiate connection with the server. From this point, client and server communicate directly.
5. The client is actually communicating with the attacker in the clear, but the second handshake is encrypted and goes over the attacker's channel. Note that the attacker cannot actually view the data that is sent in plaintext (because it is encrypted, but can manipulate it to get what he wants)
6. The client also does not know that he is renegotiating.
7. However, the server thinks that the initial traffic sent by the attacker is also from the client

*Impact on Existing Applications*

TLS itself is just a security protocol, so the impact of this attack depends on the application protocol running over TLS. The most important of these protocols is of course HTTP over TLS (HTTPS). Most Web applications do initial authentication via a username/password pair and then persist that authentication state with HTTP cookies (a secret token that is sent with any request). An attacker might exploit this issue by sending a partial HTTP request of his own that requested some resource. This then gets prefixed to the client's real request.

E.g., the attacker would send:

```
GET /pizza?toppings=pepperoni;address=attackersaddress HTTP/1.1
X-Ignore-This:
```

And leave the last line empty without a carriage return line feed. Then when the client makes his own request

```
GET /pizza?toppings=sausage;address=victimssaddress HTTP/1.1
Cookie: victimscookie
```

the two requests get glued together into:

```
GET /pizza?toppings=pepperoni;address=attackersaddress HTTP/1.1
X-Ignore-This: GET /pizza?toppings=sausage;address=victimssaddress HTTP/1.1
Cookie: victimscookie
```

**THE FIX**

Disable renegotiation (note that there are some instances where renegotiation can be beneficial, e.g certificate based authentication when client requests for protected resource)
Send 2 messages at once, such that server only takes the second request etc.