Assignment 3 Report - COL759

Ankit Solanki // 2016CS50401

August 11, 2020

1 Protocol: SSLv3

1.1 Introduction

- Preceded after two versions, Secure Sockets Layer (SSL) is a standard security tech for making an encrypted connection between a server and a client.
- Designed in 1995, it is now an insecure, obsolete, and rarely used protocol (because of the next stable version of TLS. But as it goes, many TLS versions remains backwards compatible with SSLv3 protocol; this is to ensure a smooth experience for the user when he switches with an old protocol. Usually, for authentication handshakes the latest version common to client and server is used, but since a client can use a very old protocol (SSLv3 here,) server has to be compatible with that.
- SSLv3 creates a session link between client and server; it utilizes keys used for encryption and decryption, generated with symmetric encryption scheme and public-key cryptographic protocols such as RSA, AES, DES.

1.2 Hash-based Message Authentication Code

- Message Authentication Code (MAC) is a minimal piece of information using which one can find if the received message is authentic, i.e., it has come from the stated sender and had not been tampered with.
- HMAC (Hash-based Message Authentication Code) is a particular type of such MAC, in which a secret cryptographic key is used alongside a cryptographic hash function. It can be used to verify the data authenticity (coming from the right client) and data integrity (same data as sent by the client.) SSLv3 uses HMAC to ensure these two important properties of the received data.
- SSLv3 uses the MAC-then-encrypt mode for authenticating. Which implies to know the content, the receiver has to decrypt and then check for authenticity and integrity.
- Calculation of HMAC types:

- Encrypt-then-MAC: First encrypt the data and then take MAC and append to cipher.
- MAC-and-encrypt: Take MAC and appended to the encrypted data.
- MAC-then-encrypt: Take MAC of data then append it to the data then encrypt.

1.3 Cipher Block Chaining Mode

- SSLv3 encryption system utilizes RC4 stream cipher or a block ciphering technique in a type of mode commonly known as cipher block chaining because ciphered block are chained to each other.
- In this mode of chaining, we take XOR of every block of plain-text with the previous encrypted block before encrypting it. Thus every encrypted block depends on all blocks processed up to that point, making it all inter-dependent. To make each encryption of plain-text unique what one usually practices is to have a random initialization vector for a first block every next time.

1.3.1 Padding in blocks:

- After dividing the plain-text into blocks of given block-length (here 16 bytes; depends on the encryption algorithm), the padding is done in a manner such as the length of the data will be a next integral multiple of block-length.
- In padding, all bytes are random except the last with represents the size of padded bytes.
- For example:
  Say block length = 8
  A - P - P - L - E - 0xae - 0xee - 0x02
  - Here ‘0xae - 0xee - 0x02‘ is the padded bytes and the last byte i.e., 0x02 is length of the padded bytes.

  If the message already has 8 bytes, say:
  Z - U - C - C - H - I - N - I then we pad 8 bytes to it
  Z - U - C - C - H - I - N - I - 0xae - 0xee- 0xab- 0xbe- 0xac- 0xce- 0xcc- 0x08
  signifying the 8 padded bytes by last byte as 0x08.

1.3.2 Encryption and Decryption Methodology:

We do encryption as:

```
Ci = Ek(Pi XOR Ci-1) and C0 = IV
```

And decryption as:

```
Pi = Dk(Ci XOR Ci-1) and C0 = IV
```

Here Ek and Dk are encryption and decryption functions of the chosen algorithm with symmetric key as k.

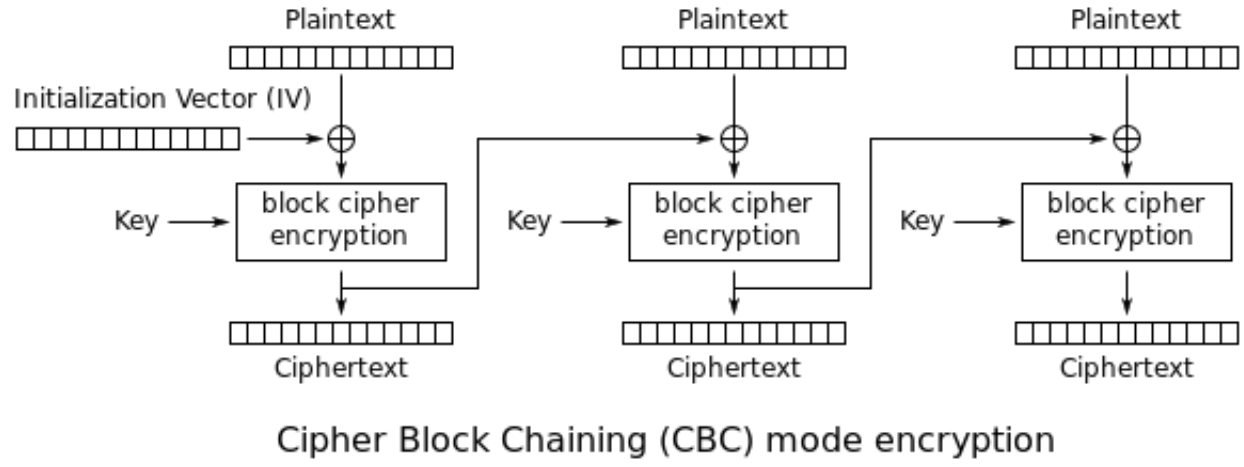

Figure 1: source

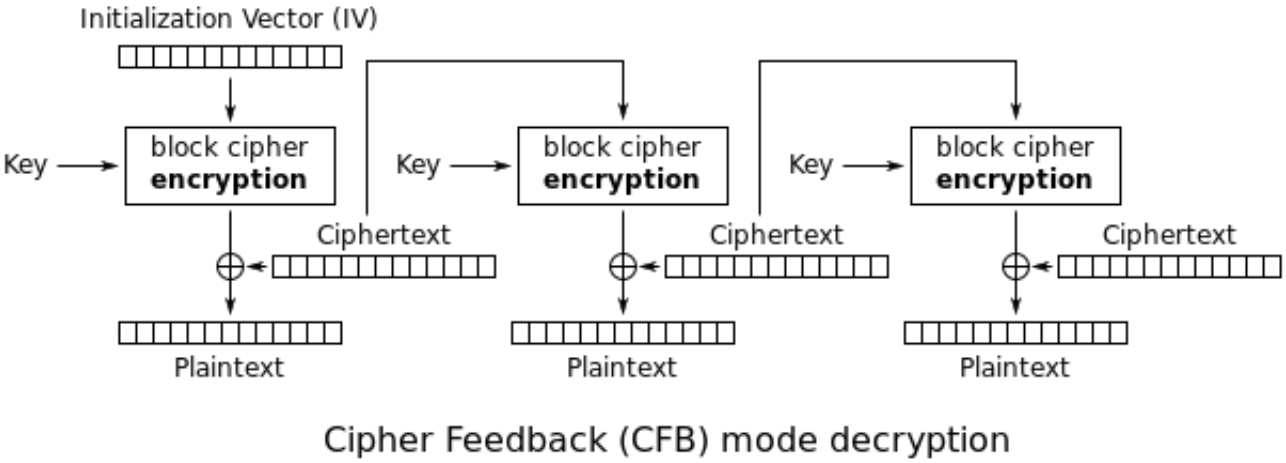

Figure 2: source

2 Poodle Attack

Poodle attack is abbreviated as Padding Oracle on Downgrade Legacy Encryption.

2.1 Introduction

- It is a type of man-in-the-middle (MINM) exploitation taking the advantage of security flaw in SSLv3 and clients using it. Any attacker needed to make on an average of 256

SSLv3 requests to the server to successfully exploit this vulnerability and reveal one byte of encrypted data that the attacker wants.

- Google Security Team discovered this vulnerability, and they disclosed it publicly not before October 14, 2014.

2.2 Vulnerability and Design Flaw

The attack can be exploited because of two serious design flaws in SSLv3:

- One of the flaws that go to the CBC encryption methodology which SSLv3 uses to encrypt: There is no way of knowing the actual length of padding, and is not covered in the authentication of the message. This creates a gap to exploit, i.e., a vulnerability that it's not possible for padding to be thoroughly checked and authentic during decryption. Mostly because of the HMAC technique mac-then-encrypt that SSLv3 utilizes.

- SSLv3 padding is badly designed. Using SSLv3 in block cipher chaining (CBC) mode of encryption, the client must add padding bytes to make the data (plaintext and its mac) to be an integral number of blocks in length. This leaves the gap that the padding can be at most one block in length. SSLv3 design says that only the last byte tells the length of the padding. It introduces another flaw.

2.3 Logic of Attack

Let's describe the logic of poodle attack. Assume there is a data that we want to send to the server.

- At the client-side:
  - Before sending this data, to maintain its authenticity and integrity, we first create its MAC and then append it to data (data + MAC(data)).
  - Now to make its size integral multiple of block size, we pad bytes [1, L] behind the data.
  - After this process, we encrypt (say AES) in CBC mode.
  - Now say there is a full padded full at the end (we can make it if not, we'll see in the next section.)

- At the server-side, after getting the request.
  - Decrypts the request with AES same key in CBC mode
  - Removes the apt padding, get the mac from request and authenticate it
  - If MAC(data) == mac then accept otherwise reject

- Accepting and rejecting like this at the server-side, we are essentially letting a listener know if the last byte is correct or not. Now if there's a full block of padding, let's say an attacker replaced that last non-data block (Cn) by an earlier encrypted data block (Ci) from the same request. Then this would be accepted if the last byte in Ci happens to be the same as encrypted form padded length byte and the server will reject it any else. This creates a point of free attack known as poodle attack.

- Since the last possible byte can be of 256 characters, the probability is 1/256 that the replaced last byte is exactly the same as the original. Thus, it means there will the request will be accepted by the server.

- Now anyone can use the following operation to find the last byte of the block Ci (which was replaced in place of Cn) as follows:

- ```
  Pn = Dk(Cn) XOR Cn-1
  Pn = Dk(Ci) XOR Cn-1
  xxxxxxx15 = Dk(Ci) XOR Cn-1
  Dk(Ci) = Cn-1 XOR xxxxxxx15
  Pi XOR Ci-1 = Cn-1 XOR xxxxxxx15
  Pi = Cn-1 XOR xxxxxxx15 XOR Ci-1
  ```

- Therefore, the last byte = Ci-1[15] XOR 15 XOR Cn-1[15]

- By doing this the attacker gets the last byte, similarly at this point attacked adds extra data to the request replacing next byte as the final byte, which then will be decrypted in a similar manner, revealing the next byte. Thus, the attacker can be doing it again and again until it has decrypted data.

2.4 Reduction and Safeguarding

- Attacks on SSLv3 are more common than thought, such as Beast, Lucky-13, Crime, and Poodle. Poodle attack requires the capability of an attacker to inject code into the victim's browser (a way of manipulating requests). But while other attacks had a workaround, the only way to stop Poodle is not to use SSLv3. And to disable SSLv3 support from browsers and web servers. This works because we have TLS coming in 1999, which is way better than SSLv3.

- Also, TLS FALLBACK SCSV mechanism exists in it that guarantees that SSL link will never fell back to a lower legacy version than the highest supported by the server or any version that is fault-free.

- TLS also prevents an attacker from downgrading the connection to legacy SSLv3 instead of a more secure one, i.e., TLS1.0
  ```

3 Paper implementation of an attacker

To implement/simulate such attack, the attacker must have the following capabilities (we'll see how an attacker can get this):

- Control of the connection link between the client and the server.
- To be able to inject any code into client's browser (e.g., js-code)

3.1 Force Usage of SSLv3 Protocol

- For to exploit the SSLv3, the client and server must be using that protocol. However, usually they use more secure TLS protocol, but a wise attacker can force them to downgrade their protocol.
- Say an attacker controls the connection between the client and the server. Now he interrupts the handshake link between the client and the server, which results in both (client and server) thinking that they might be trying with higher versioned protocol, they'll downgrade their protocol, for compatibility reasons, this is a built-in fallback mechanism that lets them retry a failed negotiation by using an older version. The version can downgrade from TLSv1.2, TLSv1.1, TLSv1.0 and finally to SSLv3, which the attacker can exploit.
- To do this successfully and then to use the Poodle vulnerability, an attacker should likely to be either on the same network or somehow be able to successfully execute malicious JavaScript code and scripts in victim's browser.

3.2 Man in the Middle Attack Step-by-Step Progression:

In the previous section, we saw how the progression moves, now we will go deeper into it. Say the attacker know that the last encrypted block when accepted and it will give 15 since the last block is padding (if not we'll see how to make it so).

- Assume the attacker know that the important data (say cookie here) is in C3 (3rd) block of encrypted data. Since HTTP request headers are fixed, this can be easily assumed.
- The attacker then changes the requested data in a way such that the final block is only padding block. He can do this by adding anything in the data (say 'X') in the headers.
- ```
  GET / HTTPS/1.1 SECRET COOKIE
  GET /XXXXXXXXX HTTPS/1.1 SECRET COOKIE DATA (8 bytes will be padded)
  ```
- Now the attacker replaces the last block of the encrypted with the block he needs, here say C3.
  ```

- This request will either be accepted or not accepted based on whether if the padding length is correct after decryption at the server side; the value of last byte is 15. The server checks this first after decryption.
- If by luck, this padding length lies between [0, 15] (between the possible padding length, it won't be rejected right away), the server will remove padding and then check the MAC, which will fail in most cases.
- The attacker will make the same request again and again, which due to the randomized IV of AES CBC mode encryption, will have new encrypted data every time. On average after 256 requests, the server will accept the request, which means we know for certain that the last byte shows 15 length. Using this fact, and as discussed above, last byte value of 3rd block of the actual data can be found out as P3[15] = Cn-1[15] XOR 15 XOR C2[15]
- The attacker now found one byte, he shifts the data by adding 'X' in request path, and then does it again to find next byte. Once done with this block, he can move again for the next block.
- Now the attacker can repeat the process to decrypt the total data.

4 My Code

4.1 Explanation (Written as Function-documentation)

NOTE: Only **important** functions are reported here.

4.1.1 Protocol SSLv3

```
class SSLv3:
    '''Simulating SSLv3 Protocol'''

    def encrypt_data(self, data):
        '''
        @param: data <- unencrypted data
        - calculate the MAC of input data using md5 hash
        - append this mac to data as: data = data + mac
        - add necessary padding to the data
        - return data encrypted by AES in CBC mode
        '''

        return self.encrypt_cipher_block(self.padding(data + hashlib.new('
    md5', data).digest()))

    def decrypt_data(self, data):
        '''
        @param: data <- encrypted data
        - decrypt data using AES in CBC mode
```

```
        - reject if last byte doesn't belong to [0, 15] i.e., wrong padding
        - else remove padding and check mac of data
            - if mac not correct, reject request
            - else accept it (return True)
        '''

        data = self.decrypt_cipher_block(data)
        if not (data[-1] <= 15 and data[-1] >= 0):
            return False

        data = data[ : -(data[-1]+1)]
        plain, mac = data[ : -16], data[-16 : ]

        return False if hashlib.new('md5', plain).digest() != mac else True
```

4.1.2 Poodle

```
class Poodle:
    '''Simulate the attacker using poodle vulnerability'''

    def block_decryption(self, add, b_index):
        '''
        function to decrypt one block (say X) at a time:
        - target 1st byte of the block need to be decrypted:
            - intercept the encrypted data coming from client
            - modify it making it such as the last full block is padded block
            - replace the last block with X block and send to server
            - if rejected, repeat the process each time new last byte is encrypted
                - else: accepted -> we have successfuly decrypted the byte
            - move to next byte by adding one byte in request
        - return the decrypted block
        '''

        print("[#] Block: {}".format(b_index))

        plain = []
        _range = None
        if b_index == 1:
            _range = self.b_length - len(add)
        else:
            _range  = self.b_length

        for b in range(_range):
            count = 0
            while True:
                count += 1
                cip = self.client(add + b'x'* b, b)
                new_cip = cip[:-self.b_length] + cip[b_index*self.b_length : b_index*self.b_length+self.b_length]
                if self.server(new_cip):
```

```
                    break

            new_blocks = self.ssl.fragment_blocks(new_cip)
            val = new_blocks[b_index-1][-1] ^ 15 ^ new_blocks[len(
    new_blocks)-2][-1]
            print("\t[*] #{} byte | attempts - {} decode: {}".format(b,
    count, bytes([val])))

            plain.append(bytes([val]))
        plain.reverse()

        print("[*] Decrypted Block: {}\n".format(str(b''.join(plain))))
        return b''.join(plain)
```

4.1.3 Attack

```
class Attack:
    '''Handle the Poodle vulnerability attack'''

    def perform_attack(self, enc_data):
        ''' @param: enc_data
        - start from 2nd block till 2nd-last block (exclusive) where the
    data is present (design choice)
        - attack block by block and return collected decrypted data
        '''

        enc_data, add = self.pad_full_block(enc_data)
        dec_text = b''
        for i in range(1, (len(enc_data) // self.poodle.b_length)-2):
            dec_text += self.poodle.block_decryption(add, i)
        return dec_text
```

4.2 How to run the code

For to input some data, run it like this

```
python3 poodle.py "type data here"
```

For to use the hard-coded data: "Cryptography Assignment 3 by Ankit", run it like this

```
python3 poodle.py
```

4.3 Example Simulation

```
solanki@gabito:~/Spring2020/759/poodle-attack$ python3 poodle.py
[?] Secret data:  b'Cryptography Assignment 3 by Ankit'
[#] Block: 1
  [*] #0 byte | attempts - 74 decode: b'r'
  [*] #1 byte | attempts - 269 decode: b'C'
[*] Decrypted Block: b'Cr'

[#] Block: 2
  [*] #0 byte | attempts - 224 decode: b'g'
  [*] #1 byte | attempts - 90 decode: b'i'
```

```
  [*] #2 byte | attempts - 149 decode: b's'
  [*] #3 byte | attempts - 205 decode: b's'
  [*] #4 byte | attempts - 34 decode: b'A'
  [*] #5 byte | attempts - 426 decode: b' '
  [*] #6 byte | attempts - 140 decode: b'y'
  [*] #7 byte | attempts - 162 decode: b'h'
  [*] #8 byte | attempts - 60 decode: b'p'
  [*] #9 byte | attempts - 272 decode: b'a'
  [*] #10 byte | attempts - 506 decode: b'r'
  [*] #11 byte | attempts - 299 decode: b'g'
  [*] #12 byte | attempts - 394 decode: b'o'
  [*] #13 byte | attempts - 524 decode: b't'
  [*] #14 byte | attempts - 839 decode: b'p'
  [*] #15 byte | attempts - 24 decode: b'y'
[*] Decrypted Block: b'yptography Assig'

[#] Block: 3
  [*] #0 byte | attempts - 5 decode: b't'
  [*] #1 byte | attempts - 330 decode: b'i'
  [*] #2 byte | attempts - 102 decode: b'k'
  [*] #3 byte | attempts - 46 decode: b'n'
  [*] #4 byte | attempts - 256 decode: b'A'
  [*] #5 byte | attempts - 38 decode: b' '
  [*] #6 byte | attempts - 146 decode: b'y'
  [*] #7 byte | attempts - 97 decode: b'b'
  [*] #8 byte | attempts - 80 decode: b' '
  [*] #9 byte | attempts - 141 decode: b'3'
  [*] #10 byte | attempts - 533 decode: b' '
  [*] #11 byte | attempts - 788 decode: b't'
  [*] #12 byte | attempts - 129 decode: b'n'
  [*] #13 byte | attempts - 163 decode: b'e'
  [*] #14 byte | attempts - 611 decode: b'm'
  [*] #15 byte | attempts - 63 decode: b'n'
[*] Decrypted Block: b'nment 3 by Ankit'

[$] Decrypted data:  b'Cryptography Assignment 3 by Ankit'
```