Return of the Return Of the Bleichenbacher's Oracle Threat

IN4253ET Applied Security Analysis - "Hacking Lab"
Supervisor: Dr. C. Doerr

Jonathan Raes	Menno Bezema	Sudharshan Kumar Swaminathan
4300343	4248252	5148340

May 6, 2020

1 Introduction

This report is aimed to accompany our work on an implementation of the Bleichenbacher's attack. The original Bleichenbacher's attack is an adaptive chosen ciphertext attack against certain protocols based on RSA using the PKCS#1 v1.5 padding scheme. The attack allows an RSA private-key operation to be performed if the attacker has access to an oracle that, for any chosen ciphertext, reveals one bit of information about the plaintext corresponding to the supplied ciphertext(Bleichenbacher, 1998). In 2018 a new paper was published that showed the vulnerability is still very prevalent (Böck, Somorovsky, & Young, 2018).

The goal of this report is to take a deep-dive into the ROBOT attack and produce a Proof of Concept demonstrating the exploitation of the ROBOT attack. To do this we first provide an *explanation of the attack*. Secondly, a section on the *scenarios* that follow from the attack. We then explain our *implementation of the attack* followed by the *demo*. After the demo we show some *corrections* to the original script which optimize the attack. Finally we *conclude on the implications of our findings.*

2 The Bleichenbacher's attack

In this chapter we will go into the Bleichenbacher's attack. First providing a short explanation of what the attack is. Followed by an explanation of how the ROBOT paper and script *scans for servers containing the vulnerability*. Then a section about how *the attack would be performed*. Finishing off with a short discussion on how to *mitigate* the attack.

2.1 Explanation of the Attack

The original Bleichenbacher's attack is an adaptive chosen-ciphertext attack on RSA PKCS#1 v1.5 as used in SSL (Bleichenbacher, 1998). The validity of RSA PKCS#1 v1.5 is based on the padding of the message. The padding is done as follows (Böck et al., 2018):

1. Take a message k and an RSA public key (N,e)
2. Generate a random padding string PS where $|PS| > 8$ ($|x|$ is defined as the length of x) and $|PS| = |N| - 3 - |k|$ and $0x00 \notin PS$.
3. Compute message block m as: $00||02||PS||00||k$.
4. Finally compute ciphertext $c = m^e * \mod N$.
5. Decryption is done by performing RSA decyption with the private key, checking the PKCS # v1.5 padding and extracting k.

The Bleichenbacher's attack is performed by sending a ClientKeyExchange message during the TLS handshake and modifying its contents for every connection. The server will then attempt to decrypt this and based on the validity of the PKCS#1 v1.5 padding of the resulting message, the server will provide a different response. The difference in responses provides an oracle to the attacker, who can utilize it to try and decrypt intercepted messages as well as sign arbitrary messages using the private key of the server. In section 2.3 we provide a short overview of the attack.[1]

2.2 Detection of the vulnerability

The flaw exploited by the Bleichenbacher's attack lies in the implementation of SSL/TLS and not in the protocol itself. The technologies vulnerable to the attack as mentioned in (Böck et al., 2018), implement the padding checks in such a way that gives rise to the aforementioned oracle. For example, in the vulnerable version of Erlang (v18 in this case), the implementation is such that when an encrypted message with correct padding format is decrypted at the server end, it responds TLS alert of 20 emphasising that it could not understand the decrypted message. This shows us that it did not find any mistake in the padding. Whereas if it encounters a wrong PKCS#1 v1.5 format (for example with no

[1]Because the previous papers provide a good explanation of the details of the attack we deemed it redundant to revisit the specifics of the attack here. If you are looking for that we would like to refer you to original papers (Bleichenbacher, 1998; Böck et al., 2018).

0x00 byte anywhere in the message) during decryption, it responds with a TLS alert of 51 indicating a decrypt error.

These differences in the responses for the messages sent with different padding formats is used to detect whether the server is vulnerable to Bleichenbacher's attack. The script as given on their Github repository (Meyer et al., 2014a), uses the following tests to check the presence of the vulnerability.

- **Correct message**.
  A message with the correct PKCS #1 v1.5 padding, meaning it starts with 0x0002 followed by random padding which is ended with 0x00, followed by the TLS version and a random premaster secret. This message is important because it helps to check server correctness. And it is later on used to verify if the attempt at forging a message is correct (Böck et al., 2018).

- **Wrong first two bytes, *0x4117***.
  This message should trigger an invalid server behavior (Bleichenbacher, 1998).

- ***0x00* on a wrong position**.
  This check is performed to see if the server correctly checks the length of the unpadded premaster secret. If this is not done correctly this hints at the possibility of a buffer overflow and inconsistent server behavior (Meyer et al., 2014b).

- **No *0x00* in the middle**.
  If this does not produce an error the PKCS #1 v1.5 implementation is unable to unpad the encrypted value, which can again lead to inconsistent server behavior (Böck et al., 2018).

- **Wrong TLS version number *0x0202***.
  This should trigger invalid behavior, which could lead to the server being an oracle (Klíma, Pokorný, & Rosa, 2003).

A server that is not vulnerable to the Bleichenbacher's attack should respond to all of the above messages with the same alert messages. However a vulnerable server responds with verbose decryption errors, resulting in either a weak or a strong oracle. The script characterizes an oracle as weak if it only identifies packets starting with *0x0002* and having correct padding. The oracle is considered weak because of the low probability of this happening, and thus the large amount of messages needed to successfully execute the attack. Alternatively, if the server leaks information on multiple messages it is considered a strong oracle and the attack will complete much quicker (Böck et al., 2018).

2.3 Exploitation of the vulnerability

These verbose decryption errors can be used as an oracle to *predict* the plaintext of a corresponding ciphertext. The space in which a message will lie is huge but finite because of the format of the padding, and the oracle can now help us shorten or guess the interval in which the decrypted message lies. Eventually reaching an interval with only one element.

The attack starts when an attacker sends a ciphertext c to the oracle. The oracle will send a different reply if the plaintext starts with *0x0002* than if it does not. Thus, the oracle can be described in the following way:

$$O(c) = \begin{cases} 1, & \text{if } m = c^d \mod N \text{ starts with } 0x0002. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Then, the homogeneity of RSA is used where multiplying a ciphertext with the inverse of a constant will decrypt to the message multiplied by the constant itself. Using this the attacker can perform plaintext multiplications using an integer s and RSA public key (N, e):

$$ciphertext' = (ciphertext * s^e) \mod N = (m * s)^e \mod N$$

Using a PKCS #1 v1.5 conforming message m. The attacker can start with small values of s and iteratively increase it, computing c' with the new s and query the oracle. This is repeated till the oracle confirms that the ciphertext is indeed PKCS #1 v1.5 conforming. On that response the attacker learns new bounds that narrow the search for a valid message. By repeating this process the attacker narrows down the search until he find out the final message $m = c^d \mod N$.

This final message is the result of $c^d \mod N$. When c is supplied as a PKCS padded plaintext message, the result is that message raised to the servers private key mod N, which is the RSA signing function. Thus, m is now a valid signature signed by the server's private RSA key.

When the initial supplied c was a valid PKCS#1 v1.5 padded *RSA encrypted* message, the oracle will perform decryption by raising it to the power of the server's private key. The final m will then be a PKCS padded plaintext. As stated earlier, the attack is explained in more detail by the original source (Bleichenbacher, 1998).

2.4 Mitigation

This leads to the possible mitigations of this vulnerability. The whole attack relies on servers sending different messages based on validity of the ciphertext. Such implementation errors have rather simple fixes, mostly involving consistent results regardless of the formatting of the input data. The best solution would therefore be to immediately perform the integrity check after the decryption and not aborting if the padding of the data is incorrect. In this case, if the padding is wrong, it will still result in a failed integrity check and eventually consistent results for the overall decryption operation thereby not giving way to oracles. For example, OpenSSL which is resilient to the padding variation of the Bleichenbacher's attack returns a value of `-1` if the decryption of the received message is not successful. This is regardless whether the individual operations (such as RSA decryption, padding removal and MAC check) of the decryption process were successful or not. It is also recommended to disable RSA encryption modes altogether (RSA cipher with DHE or ECDHE are still safe) as they are becoming too risky and will only become riskier with increasing technological advances (Böck et al., 2018).

3 Exploitation Scenarios

We use this chapter to look at the possible exploitation scenarios of the Bleichenbacher's attack. The first scenario we discuss is a *passive Man-in-the-middle attack*, wherein we exploit the vulnerability to decrypt the premaster secret and subsequently use the decrypted premaster secret to decrypt the intercepted application data. In the second scenario, we use the attack to *sign Ephemeral Diffie-Hellman parameters* defined by us (the attacker) on behalf of the victim server that would then enable us to be an active man-in-the-middle or impersonate the server itself.

3.1 Scenario 1: Decrypting captured SSL/TLS session

As stated in Section-2, we should be able to exploit the Bleichenbacher vulnerability to decrypt any arbitrary message with the server's private key (in the case of a signature, encrypt). In the case the cipher suite `TLS_RSA_WITH_AES_256_CBC_SHA` is being used, RSA is used for key exchange and the application data is encrypted with AES in CBC mode using the derived keys. An SSL/TLS handshake involves exchange of encrypted premaster secret by the client which is then used to derive the master key and the subsequent AES keys. We as an attacker act as a passive Man-in-the-middle who is able to capture SSL/TLS sessions between the client and the victim server and store them in *pcap* files. Then by following the steps given below, we execute the attack. [2]

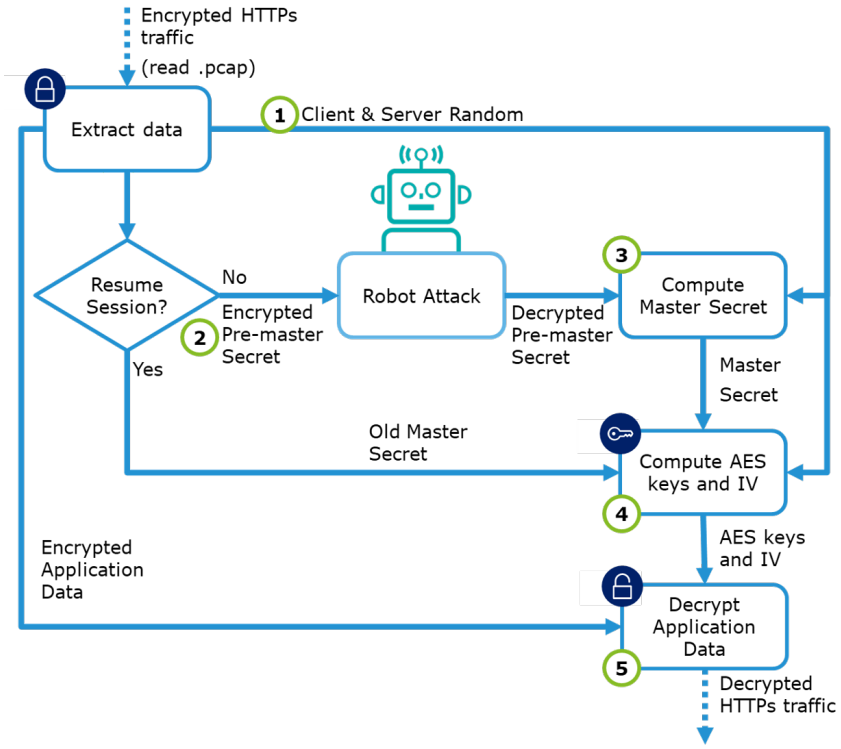

Figure 1: Schematic overview of the attack

[2]Refer to *Appendix A* for the corresponding Python code.

1. We gather the client random and server random from the Client Hello and Server Hello messages respectively. This will be used in computing the master secret and the AES keys.

2. We then obtain the encrypted premaster secret from the Client Key Exchange and use the Bleichenbacher attack to decrypt it. This involves sending queries to the oracle as the decryption is done without actually obtaining the private key. As the premaster secret is 48 bytes in this case, the last 48 bytes is extracted from the value obtained from the ROBOT attack script.

3. The plaintext premaster secret is then used along with the client random and server random to derive the master secret. This is done as per the derivation given in Section 8.1 of RFC 5246.

4. The master secret is then used to derive the keys as given in Section 6.3 of RFC 5246. As given here, the first 40 bytes of key material (since SHA1 is used) will be client write MAC key and server write MAC key which we do not need at this point. The rest of the derived bytes are used as follows:

   (a) `keys[40..72]` - Client write key, which is used to encrypt the transmission from the client to the server. Therefore, can be used to decrypt application data being sent by the client.

   (b) `keys[72..104]` - Server write key, which is used to encrypt the transmission from the server to the client. Therefore, can be used to decrypt application data being sent by the server.

   (c) `keys[104..120]` - 16 bytes of client write IV, the IV used as the seed when the client encrypts the first transmitted message. Therefore, this is used as the IV while decrypting this message transmitted by the client.

   (d) `keys[120..136]` - 16 bytes of server write IV, the IV used as the seed when the server encrypts the first transmitted message. Therefore, this is used as the IV while decrypting this message transmitted by the server.

   In case of session renegotiation while fetching multiple resources from the server, we use the older master secret from the previous session and the newly transmitted client random and server random to compute the new set of keys and IVs.

5. We use the keys and the IV derived in the above step to decrypt the captured application data, and thereby being able to decrypt all the data that is transmitted between the client and the server.

Since the cipher `TLS_RSA_WITH_AES_256_CBC_SHA` does not provide forward secrecy implicitly, it is possible to decrypt any SSL/TLS session between the client and the server if the attacker is able to capture the handshake of any one of the sessions. We implemented this attack on a server running Erlang v18.0 and the results of the same have been demonstrated in Section-4.2.

3.2 Scenario 2: Active Man-in-the-middle by signing DH-parameters

It is also possible for an attacker to actively intercept the client and the server, without the client knowing about it, thereby effectively performing an active Man-in-the-middle. Considering the case where the attacker has poisoned the DNS cache of the resolver the client connects to, and is therefore able to redirect the traffic to the victim website's domain to his own server.

For this attack we assume the server uses `ECDHE` with `RSA` signature. The attacker now creates a set of `DHParams` and signs them with the victim server's private key using the Bleichenbacher's attack. These `DHParams` along with the signature are then returned to the client in the Server Key Exchange message which is sent after the attacker sends the legitimate server's certificate. The client accepts the certificate and now tries to verify the `DHParams` using the public key of the server. Since the `DHParams` have actually been signed with the private key of the server, the verification is successful, and the connection is seen as legitimate.

The attacker succeeds in imitating the server and is able to send and receive messages on behalf of the actual domain. Since the `DHParams` have been created by the attacker, they will be able to derive the shared key from it and will be able to decrypt the messages sent by the client.

We do not make an implementation of the above attack as a part of this project due to time constraints.

4 Implemented Examples

This chapter starts of with a short summary of possible server implementations and then demonstrates our implementation of the attack on 2 demo servers, *BouncyCastle* and *Erlang*. From the vulnerable implementations as stated in Böck et al. (2018), we ruled out wolfSSL in combination with nginx early because it only had a weak oracle (correctly padded message would timeout, and any message otherwise would return errors), making the attack really slow. We then started OpenSSL, BouncyCastle and PurePython simultaneously. OpenSSL with apache was tried for a short period of time until we discovered that it might only be vulnerable to a timing-variant of the Bleichenbacher's attack which is only speculative and might not be exploitable[3]. PurePython was considered for a while but upon consultation with our professor we decided it would not be useful to build a vulnerable server to prove that it is vulnerable. BouncyCastle seemed more fruitful so we tried to implement the attack as given in Section-3.1 on BouncyCastle. Although, it seemed vulnerable at first (as was also indicated by the robot-detect script[4]), we believe that it was not exploitable in spite of the server returning slightly different results for differently padded data. We then tried the attack on Erlang v18.0 server and were successful. We demonstrate the steps taken for the setup and attack on both these servers in the following sections.

4.1 BouncyCastle

According to Böck et al. (2018), versions prior to 1.59, specifically before commit 6bad5c[5] should be vulnerable. We checked out the BouncyCastle source code one commit before this fix and built a simple TLS server using this BouncyCastle build. This server can be found in this Github repository[6]. It is a quite heavily modified version of a BouncyCastle TLS server implemented by Ruhr University in Bochum[7]. It needed to be modified more to make it use the JCE API in Java for cryptographic operations. The BouncyCastle source code was also slightly modified to make it print out the Master and pre-master secrets for every connection for debugging purposes.

BouncyCastle returns an alert 80 packet when the *0x00* byte terminating the padding is in a wrong location, but returns a `ChangeCipherSpec` message in all other cases. This is as described in Böck et al. (2018). However, even though the provided *robot-detect* script[4] thought so, we think the server implementation was not actually vulnerable to the Bleichenbacher's attack. The output of the scripts' run for detecting the vulnerability is shown below. The output has been truncated for clarity purposes.

[3] `https://mta.openssl.org/pipermail/openssl-dev/2017-December/009887.html`
[4] `https://github.com/robotattackorg/robot-detect`
[5] `https://github.com/bcgit/bc-java/commit/a00b684465b38d722ca9a3543b8af8568e6bad5c`
[6] `https://github.com/jonathanraes/BouncyCastleServer/`
[7] `https://github.com/RUB-NDS/BouncyCastleTLS`

```
>>> VULNERABLE! Oracle (strong) found on 127.0.0.1/127.0.0.1, TLSv1.2,
standard message flow: ... [truncated]

>>> Result of good request:                          Received something other
than an alert (b'\x14\x03\x03\x00\x01\x01')
>>> Result of bad request 1 (wrong first bytes):   Received something other
than an alert (b'\x14\x03\x03\x00\x01\x01')
>>> Result of bad request 2 (wrong 0x00 position): TLS alert 80 of length 7
>>> Result of bad request 3 (missing 0x00):        Received something other
than an alert (b'\x14\x03\x03\x00\x01\x01')
>>> Result of bad request 4 (bad TLS version):     Received something other
than an alert (b'\x14\x03\x03\x00\x01\x01')
```

As it can be seen from the above output, the server responds with TLS alert 80 (`internal_error`) for `bad_request 2` with *0x00* at the wrong position thereby giving us a different response when the padding is wrong as expected from a vulnerable setup. However, there is also no difference in output between other bad requests and the good request. This, we believe could have led to the unexploitable nature of the server. While querying the server with multiple wrongly formatted messages during the attack, it would not be possible to distinguish between a good message and a message with the wrong first bytes. The same applies for a message with a missing *0x00* and a message with bad TLS version.

The attack relies on identifying a PKCS conforming a message from the server (`good_request`), it therefore also strongly relies on distinguishing between the `good_request` and `bad_request`. Since the only message that could actually indicate that we have sent the server a wrong message is the one with a wrong *0x00* position, it is with significant probability that even the wrongly formatted messages (`bad_request 1, 3, 4` that are not PKCS conforming will be taken as valid messages by the attack script since the server responds with the same message as the `good_request`. This leads to false positives where PKCS non-conforming messages might also be taken to be PKCS conforming, and will therefore result in wrong intervals to be computed during the Bleichenbacher's attack and eventually in wrong results of the signature or the decrypted text. A snippet of the Bleichenbacher's function used in the attack script is given below which shows that only the result of good oracles is taken into consideration while progressing the attack (the comment with a ## has been inserted by us).

```
def BleichenbacherOracle(cc):
    ...
    ...
    if o == oracle_good: ## only PKCS conforming messages to be considered
        # Query the oracle again to make sure it is real...
        o = oracle(pms, messageflow=flow)
        if o == oracle_good:
            countvalid += 1
            return True
        else:
            print("Inconsistent result from oracle.")
            return False
    else:
        return False
```

We also noticed several inconsistent results sometimes at random points during the attack. When inspecting the issue we noticed that the attack script was selecting the full message flow in the handshake, while the paper described BouncyCastle as requiring the shortened message flow (where only the ClientKeyExchange is sent) in order to be vulnerable. When using the full message flow the BouncyCastle server would sometimes randomly break the connection resulting in a Broken Pipe error in the client, thereby disturbing the oracle results. Forcing the attack script to select the short message flow fixed the problems of inconsistent results, but unfortunately did nothing to fix the issue described above. It is due to the above mentioned reasons, we concluded that the server might not be exploitable in spite of being vulnerable and a decision was made to switch to an Erlang server for the proof of concept.

4.2 Erlang

Erlang has been vulnerable to three separate versions of the ROBOT attack. The first one got fixed in version 18.3.4.7, the second one in version 19.3.6.4 and the third one in version 20.1.7. In order to create the demo, version 18.3.4.6 was taken as a starting point, and a simple TLS server based on this Erlang version was made with the help a gist found on Github[8]. As with the BouncyCasle source code, we modified the Erlang code to make it print out the premaster and master secrets for each connection, for debugging purposes. The final Erlang server code van be found at `https://github.com/jonathanraes/erlang-tlsserver`. Building the docker container in that repository will use the modified Erlang source to build the server. Another Github repository [9] was used as a starting point to create this Dockerfile.

Unlike the BouncyCastle implementation, the Erlang server gave us clearer results to work on, wherein it was possible to clearly distinguish between `good_request` and `bad_request` to more significant extent. The sample output from a run of the script on the server is given below. The output has been truncated for clarity purposes.

```
>>> VULNERABLE! Oracle (strong) found on 172.17.0.2/172.17.0.2, TLSv1.2,
shortened message flow: ... [truncated]

>>> Result of good request:                         TLS alert 20 of length 7
>>> Result of bad request 1 (wrong first bytes):    TLS alert 51 of length 7
>>> Result of bad request 2 (wrong 0x00 position): TLS alert 20 of length 7
>>> Result of bad request 3 (missing 0x00):         TLS alert 51 of length 7
>>> Result of bad request 4 (bad TLS version):      TLS alert 20 of length 7
```

As it can be seen above, there is a much better distinction between PKCS conforming messages and PKCS non-conforming messages, i.e, `good_request` and `bad_request 1` and `bad_request 3`. These 2 types of PKCS non-conforming messages gives us a significantly better chance of landing up with the correct intervals containing the message during the Bleichenbacher's attack. The false positives that might be returned due to the similarity between `good_request` and `bad_request 2` are later adjusted with further shortening of intervals during the attack. As was expected from these better Oracle results from the server, we were able to exploit the server further to sign arbitrary messages and decrypt the any ciphertext as described in Section-3.

[8] `https://gist.github.com/lukebakken/5bff550515588d32d4d3c14d6ace51e7`

[9] `https://github.com/erlang/docker-erlang-otp/blob/e91894d9d9c3651382834b77978a05fa057338fb/18/Dockerfile`

4.3 Attack Demo

The aforementioned vulnerable Erlang server was used to create an attack demo. The demo consists of an attack script that opens a *pcap* file, or listens on a live interface and tries to find a TLS handshake. It then extracts the client and server randoms from the handshake as well as the encrypted premaster secret the client sends to the server. Then, it uses the Bleichenbacher's attack to decrypt the encrypted premaster secret. When the premaster secret is obtained, the set of AES keys and their corresponding IVs used for the communication are calculated (Section-3.1). These AES keys are then used to decrypt the TLS traffic following the handshake that also contains application data. The whole attack is shown schematically in Figure 3.1. In the following sections, we first show how we go about creating the test *pcap*, and then follow it up with the attack we perform on this captured *pcap*. The first step of the demo can be skipped as the repository containing the attack script also already contains 2 pre-made *pcap* files containing connections made to the server with both Firefox and curl.

4.3.1 Creating a packet capture for the attack

In order to make a connection to the server that can be attacked using Bleichenbacher's attack, some configuration changes need to be made. This is necessary because most modern applications have blocked the use of the RSA key agreement protocols because of their vulnerabilities. Our server therefore supports RSA and ECDH algorithms, and we need to force the clients to use RSA in order to demonstrate the attack. We show how to achieve this with 2 web clients, `curl` and Firefox.

- When using `curl`, the `--ciphers AES128-SHA:AES256-SHA256` option should be added when connecting to the server to select the right cipher suite, in addition to the flag `--insecure` to ignore errors about the self-signed certificate.
- When connecting from the browser Firefox, all key agreement protocols except for the RSA ones need to be disabled. Enter `about:config` in the *url* bar. In the configuration, options, search for the term `ssl3`. Flip all resulting options to false except for `security.ssl3.rsa_aes_256_sha` and `security.ssl3.rsa_aes_128_sha`. After this, a connection to the server can be opened with only RSA being the negotiated cipher suite. Refer to the screenshot shown below.

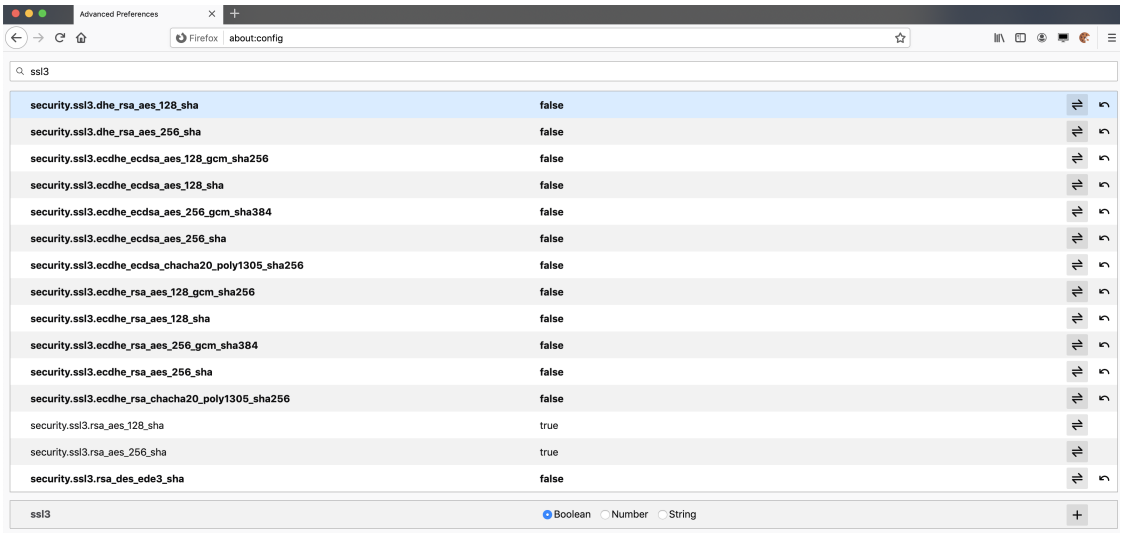

Figure 2: Cipher Suites to be disabled in Firefox

Note that we can only capture the traffic belonging to one client-server pair, i.e, traffic involving only one client. This is due to a limitation in the attack script that we discuss about in Section-4.3.3.

4.3.2 Performing the attack

Now we perform the attack on the *pcap* file as captured above for the purpose of demonstration. The demo can be replicated by following the below given steps:

1. Clone the Erlang server repository and the attack script repository as given below.

   ```
   > git clone https://github.com/jonathanraes/erlang-tlsserver
   > git clone https://github.com/sudharshankr/robot-detect
   ```

2. Build the Erlang server and start the container with the help of the Dockerfile present in the directory `erlang-tlsserver`.

   ```
   > docker build -t erlang-server .
   > docker run --publish 4000:4000 --name erlang-server erlang-server
   ```

   The above server will run on port 4000 and the port 4000 on the host is forwarded to this server.

3. Change into the directory `robot-detect/docker`. Install the requirements using the following command.

   ```
   > pip install -r requirements.txt
   ```

4. Change into the directory `robot-detect`. Run the attack by executing `attack.py` file along with the captured *pcap* file as the command-line argument as shown below, or leaving out the argument to listen on the localhost interface.

```
> python attack.py capture.pcapng
or
> python attack.py
```

   For the interactive mode, run the python file and make a request using curl or a correctly configured browser as described in the previous section.

This will start the attack by extracting the encrypted premaster secret from the given *pcap* file or live interface, and then launching the Bleichenbacher's attack on the server to decrypt it. This is then used to decrypt the application data following the TLS handshake. A sample output of the attack is shown below. (the *pcap* file used here for the purpose of demo is present in the `robot-detect` repository cloned from Github in the above steps). As can be seen from the output, the script was successfully able to decrypt the HTTP request made by the client and the HTTP response given by the server which also contains the HTML page. The attack shown below took about 17,000 queries to complete, please note that the attack can take anywhere up to several hundreds of thousands of requests.

```
    >>> Found ClientHello
    >>> client random:  0x7951fc076dc68561716bda47de5a3d5d8e087ef77bb38c6e51db759
5a6d1cc1b

    >>> Found ServerHello
    >>> server random:  0x5e997c459bad430150ae9dae1d34829fef6ee6dbd45e2fa87e4b15f
9b09e4791

    >>> Found Client key exchange
    >>> Encrypted premaster secret: b'717f0fc6d4e07ef22185ccee05bdc15142aea7ca833
a5223230d2efd79748c65e05d5e6b11e5b8143104866fdc785091e43b40a4363f094065d732ac1c9b
031443a59fc9e2c60ef81904dc6a992c8f08bda39ff0151dffd351e254bbe946ab4e5a03aa864bc3f
c85ac2dcebd50f754f293c1d2bc38a7dbc3a93ff51947dee902b4bcd7e30a81549e93d6fe642399a7
d05bf97c6a0f6733b7efe1a70ce5493800b3da3c9b87f2958857db672914ca9f40d4cdc45d8533a5e
672e91b18d8a2c926e153e1a45a9bef4c5b5819d0bb0fc67570d52189c669b4c6916122809bbe1034
45a1e64c8966dd0b081141e8cbdc7792ec6a9f785da920794c87a236586d8937'

    >>> VULNERABLE! Oracle (strong) found on 127.0.0.1/127.0.0.1, TLSv1.2, standar
d message flow: TLS alert 20 of length 7/No data received from server (TLS alert 2
0 of length 7 / No data received from server / TLS alert 20 of length 7)
    >>> Using the following ciphertext:  0x717f0fc6d4e07ef22185ccee05bdc15142aea7c
a833a5223230d2efd79748c65e05d5e6b11e5b8143104866fdc785091e43b40a4363f094065d732ac
1c9b031443a59fc9e2c60ef81904dc6a992c8f08bda39ff0151dffd351e254bbe946ab4e5a03aa864
bc3fc85ac2dcebd50f754f293c1d2bc38a7dbc3a93ff51947dee902b4bcd7e30a81549e93d6fe6423
99a7d05bf97c6a0f6733b7efe1a70ce5493800b3da3c9b87f2958857db672914ca9f40d4cdc45d853
3a5e672e91b18d8a2c926e153e1a45a9bef4c5b5819d0bb0fc67570d52189c669b4c6916122809bbe
103445a1e64c8966dd0b081141e8cbdc7792ec6a9f785da920794c87a236586d8937
    >>> Searching for the first valid ciphertext starting 1
    >>>  -> Found s0:  1

    >>> 16834 oracle queries, Interval size: 9 bit....
```

```
    >>> Starting exhaustive search on remaining interval
    >>> min:  0x221bf44d212533e1adccd22e2af314dae898e3a849d31cb74dccb9d2d38a04fd2
efaca5ae22727ff69fa11d96e5a5389c4d3c1fe1b9c20776e3659401798812b37b7cf7d1ef0e7fa16
92491520750560566ec93d1e948db6107708db4332746d13effcbd498d6a191851e1207cb50a8e877
96d387a3d3c45c587ee22dbc03089c83f01310b98c083751ad29f96a91f55b8e24be20ac6b133aaf3
6b4355ee8bddbc04ddcf2f25292865e166fe549e8c9288e5c2ef42fb8b386ce53b8416979f4f9fa9b
6a396176144dc51477f7040003039a235705f3276ebac6ffed2171d2452723058681d3c774b23de7d
eddde1f9260bcf3c6623180d6254ac1111a3836
    >>> max:  0x221bf44d212533e1adccd22e2af314dae898e3a849d31cb74dccb9d2d38a04fd2
efaca5ae22727ff69fa11d96e5a5389c4d3c1fe1b9c20776e3659401798812b37b7cf7d1ef0e7fa16
92491520750560566ec93d1e948db6107708db4332746d13effcbd498d6a191851e1207cb50a8e877
96d387a3d3c45c587ee22dbc03089c83f01310b98c083751ad29f96a91f55b8e24be20ac6b133aaf3
6b4355ee8bddbc04ddcf2f25292865e166fe549e8c9288e5c2ef42fb8b386ce53b8416979f4f9fa9b
6a396176144dc51477f7040003039a235705f3276ebac6ffed2171d2452723058681d3c774b23de7d
eddde1f9260bcf3c6623180d6254ac1111a39c2
    >>> C:       0x717f0fc6d4e07ef22185ccee05bdc15142aea7ca833a5223230d2efd79748c
65e05d5e6b11e5b8143104866fdc785091e43b40a4363f094065d732ac1c9b031443a59fc9e2c60ef
81904dc6a992c8f08bda39ff0151dffd351e254bbe946ab4e5a03aa864bc3fc85ac2dcebd50f754f2
93c1d2bc38a7dbc3a93ff51947dee902b4bcd7e30a81549e93d6fe642399a7d05bf97c6a0f6733b7e
fe1a70ce5493800b3da3c9b87f2958857db672914ca9f40d4cdc45d8533a5e672e91b18d8a2c926e1
53e1a45a9bef4c5b5819d0bb0fc67570d52189c669b4c6916122809bbe103445a1e64c8966dd0b081
141e8cbdc7792ec6a9f785da920794c87a236586d8937
    >>> result:  0x221bf44d212533e1adccd22e2af314dae898e3a849d31cb74dccb9d2d38a04
fd2efaca5ae22727ff69fa11d96e5a5389c4d3c1fe1b9c20776e3659401798812b37b7cf7d1ef0e7f
a1692491520750560566ec93d1e948db6107708db4332746d13effcbd498d6a191851e1207cb50a8e
87796d387a3d3c45c587ee22dbc03089c83f01310b98c083751ad29f96a91f55b8e24be20ac6b133a
af36b4355ee8bddbc04ddcf2f25292865e166fe549e8c9288e5c2ef42fb8b386ce53b8416979f4f9f
a9b6a396176144dc51477f7040003039a235705f3276ebac6ffed2171d2452723058681d3c774b23d
e7deddde1f9260bcf3c6623180d6254ac1111a3908
    >>> Time elapsed: 128.10201406478882 seconds (= 2.1350335677464805 minutes)
    >>> Modulus size: 2048 bit. About 0.06254981155507267 seconds per bit.
    >>> 16834 oracle queries performed, 2009 valid ciphertexts.

    >>> Decrypted premaster secret:  0x221bf44d212533e1adccd22e2af314dae898e3a849
d31cb74dccb9d2d38a04fd2efaca5ae22727ff69fa11d96e5a5389c4d3c1fe1b9c20776e365940179
8812b37b7cf7d1ef0e7fa1692491520750560566ec93d1e948db6107708db4332746d13effcbd498d
6a191851e1207cb50a8e87796d387a3d3c45c587ee22dbc03089c83f01310b98c083751ad29f96a91
f55b8e24be20ac6b133aaf36b4355ee8bddbc04ddcf2f25292865e166fe549e8c9288e5c2ef42fb8b
386ce53b8416979f4f9fa9b6a396176144dc51477f7040003039a235705f3276ebac6ffed2171d245
2723058681d3c774b23de7deddde1f9260bcf3c6623180d6254ac1111a3908

    >>> Found application data
    >>> Encrypted application data: 0x44667fbf6a89819eaf5b8d5fd0dfd97663e02932dc9
4ac2074a8ca2faf8c5bd645746024282bf463659e0a136c65927ebf509f020882b1469b5a9667d9b8
3b152b8701aa4e5364fca2df870e15c45521198d895482f9781c10d281d2b473051e770fe4d994174
2413741cb7b7f8b534424df4eaec224d33a929aa9f4c06e8508
    >>> Decrypted data:
    >>>  GET / HTTP/1.1
    >>> Host: localhost:4000
    >>> User-Agent: curl/7.69.1
    >>> Accept: */*
```

```
    >>> Found application data
    >>> Encrypted application data: 0x8a3dda8343c2bc838781a615b41808c1ff03a7270f6
c2be532ba9e1dfa4f81b148134fa106c1f3f2afd4729672e1e4bbe094afeb26f83a6d3c3bf3912f73
8e1548158cc7a61bba548486368ccbd6440454a42621fdbaf731a6aedfa63cb2828d7d2976783abae
de7849445c3c6e3fb5250dba32fbaeb09cdb94b4c1b9c9f43786432e3a34ae91b1cb9cff691916784
5ad6b0a6b4c7260da0f2cdabdd7002f85059bfe66807c5f1327fa9b108ec732c2a0257c2bfd2daaf9
2b90c94e8cde93fdfa369c62c6dfaa1677316c5ee35eb494a679377e98086bb2d84fb5844efdcf426
2cd9dda3b26a731ec23cf8684bc2952521fcb556619458a1e145b3e30a2e997f8e267907585b7e63d
17788636c4699dfe00157d15e5fc2645824b141eaa574ced3c71acacfe930bdf587b261084fab0696
9fb22257e783d3691125feeed65775
    >>> Decrypted data:
    >>>  HTTP/1.1 200 OK
    >>> Date: Sun, 18 Oct 2009 08:56:53 GMT
    >>> Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
    >>> ETag: 10000000565a5-2c-3e94b66c2e680
    >>> Accept-Ranges: bytes
    >>> Content-Length: 44
    >>> Connection: close
    >>> Content-Type: text/html

    >>> <html><body><h1>It works!</h1></body></html>
```

4.3.3 Limitation of the attack script

Currently the attack script is capable of handling communication between 2 parties, i.e., one client and server. If the captured *pcap* file or the intercepted communication on localhost has more than one client communicating with the server, the attack script will be able to decrypt the traffic belonging to only one pair of client and server, the one that it encounters first, and will not be able to decrypt any other traffic corresponding to the communication involving other clients. We were not able to code out a solution to this problem due to the time constraints. We overcome this problem by filtering out the packets belonging to each client with the help of a tool such as Wireshark. This can help us to create separate *pcap* files corresponding to each client which can then be fed to our attack script to perform the decryption.

Additionally, the attack script currently only supports the RSA_AES cipher suites `RSA_AES_128_SHA` and `RSA_AES_256_SHA`, the DES cipher `RSA_DES_EDE3_SHA` is currently not supported.

5 Corrections to the *robot-detect* script

We made 2 corrections to the script in order to get the rounding-off of the integer values right. These corrections involve calculating the upper bound on the value of s being computed in *Step 2a* and *2c* of the Bleichenbacher's attack.

1. **Correction in Step 2a.** In this step, we need to calculate the smallest possible $s_1 \geq n/(3B)$. The script calculates this as below.

   ```
   >>>     s = N // (3 * B)
   ```

   The above code calculates $s_1 = n/(3B)$ and would actually floor the value instead of ceiling it. So, we change it to:

   ```
   >>>     s = -(-N // (3 * B))
   ```

   This would correctly *ceil* the value to give us the upper bound on the required equation. We would then be able to start with a better value of s.

2. **Correction in Step 2c.** This step requires the attacker to calculate $s_i \geq \dfrac{2B + r_i n}{b}$ which as in the previous step requires an upper bound. The code does the same as below:

   ```
   >>>     s = -(-(2 * B + r * N // b))
   ```

   As we can immediately see, this would not calculate the upper bound as expected as the closing brackets ")" for the numerator in the equation is not present and is in the wrong location, which is towards the end of the statement. This was then corrected as below.

   ```
   >>>     s = -(-(2 * B + r * N) // b)
   ```

After making the above corrections, we observe that there is marginal improvement in the attack efficiency. Taking the example of the sample output given in Section-4.2, it performs 16834 oracle queries during the attack. After the corrections mentioned above are made, the script now makes 14001 oracle queries to finish the attack, which shows a marginal improvement in this case, by approximately 2000 oracle queries. This margin of improvement might even be better for attacks that involve sending more than a million oracle queries to the server, and this shows an increase in the efficiency of the attack.

6 Conclusion

In this chapter we will first discuss the results of our demo, we will then discuss what the implications of our findings are. Followed by some lessons learned during the project. The Bleichenbacher's attack shows us how verbose errors or inconsistent behaviour by an application server can be used to create an oracle that reveals sensitive data, which in this case was decrypted data or arbitrary signed output. We immediately recognize that such attacks can be very dangerous because computers will recognize a connection as secure and legitimate. In a way that an attacker will be able to obtain sensitive data such as Personally Identifiable Information, banking information, passwords etc. from the intercepted SSL sessions. Or even impersonate the server to trick the user into handing such information over unbeknownst to the user. We also realize that such a concept can be used for exposing oracles using other attack vectors as well. Such as in timing attacks used for bruteforcing the sign of a forged JWT authentication token, wherein the response time differs depending on the flow taken by the program which in turn depends on the input data.

6.1 Discussion

After a long journey of trying different implementations of servers and a lot of hours debugging servers to get it to work we finally got a working PoC with the Erlang server. For us the main lesson learned in the implementation of this attack is that it is pretty difficult to set up a vulnerable server. This is good news for the industry as this means a lot of cryptography libraries have since resolved issues regarding the Bleichenbacher attack. On top of that it means that it is very difficult for someone currently setting up a server to set one up that is still vulnerable to this attack.

6.2 Lessons Learned

For us as a group it has been an interesting period where we started off with being able to meet in person and discuss the project and ended up having to work together remotely. All things considered we still learned a lot about how to actually implement a PoC. Because at first we were afraid to look at the Erlang server due to a lack of experience we did not consider it until we were very stuck. It is difficult to take a lesson from this as we were confident the BouncyCastle implementation would work if we spent more time on it. The central question being: When to give up on a path and when to explore another path? This however is a good representation of setting up a PoC in real life. Where you do not always know if a path taken will lead to results at all. It is therefore important to set deadlines and look for other options if you are not successful, a lesson we've learned by doing.

References

Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In H. Krawczyk (Ed.), *Advances in cryptology — crypto '98* (pp. 1–12). Berlin, Heidelberg: Springer Berlin Heidelberg.

Böck, H., Somorovsky, J., & Young, C. (2018, August). Return of bleichenbacher's oracle threat (ROBOT). In *27th USENIX security symposium (USENIX security 18)* (pp. 817–849). Baltimore, MD: USENIX Association. Retrieved from `https://www.usenix.org/conference/usenixsecurity18/presentation/bock`

Klíma, V., Pokorný, O., & Rosa, T. (2003). Attacking rsa-based sessions in ssl/tls. In C. D. Walter, Ç. K. Koç, & C. Paar (Eds.), *Cryptographic hardware and embedded systems - ches 2003* (pp. 426–440). Berlin, Heidelberg: Springer Berlin Heidelberg.

Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., & Tews, E. (2014a). *Detection script for the robot vulnerability.* Retrieved from `https://github.com/robotattackorg/robot-detect`

Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., & Tews, E. (2014b, August). Revisiting ssl/tls implementations: New bleichenbacher side channels and attacks. In *23rd USENIX security symposium (USENIX security 14)* (pp. 733–748). San Diego, CA: USENIX Association. Retrieved from `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer`

A Python code for AES keys computation

Below are the functions used for computing the master secret from decrypted pre-master secret and subsequently the keys from the obtained master secret, as mentioned in Section-3.1

```
def compute_master_secret(self, client_random, server_random):
    seed0 = client_random + server_random
    label = b'master secret'
    seed1 = label + seed0
    A0 = seed1
    A1 = hmac.new(self.pms, A0, digestmod=hashlib.sha256).digest()
    A2 = hmac.new(self.pms, A1, digestmod=hashlib.sha256).digest()
    P_hash = hmac.new(self.pms, A1 + seed1, digestmod=hashlib.sha256).digest() + hmac.new
    (self.pms, A2 + seed1,digestmod=hashlib.sha256).digest()
    self.master_secret = P_hash[:48]
    self.compute_keys(client_random, server_random)
```

```
def compute_keys(self, client_random, server_random):
    master_seed = b'key expansion' + server_random + client_random
    master_0 = master_seed
    master_1 = hmac.new(self.master_secret, master_0, digestmod=hashlib.sha256).digest()
    master_2 = hmac.new(self.master_secret, master_1, digestmod=hashlib.sha256).digest()
    master_3 = hmac.new(self.master_secret, master_2, digestmod=hashlib.sha256).digest()
    master_4 = hmac.new(self.master_secret, master_3, digestmod=hashlib.sha256).digest()
    master_5 = hmac.new(self.master_secret, master_4, digestmod=hashlib.sha256).digest()
    keys = hmac.new(self.master_secret, master_1 + master_seed, digestmod=hashlib.sha256)
    .digest() + hmac.new(self.master_secret, master_2 + master_seed, digestmod=hashlib.
    sha256).digest() + hmac.new(self.master_secret, master_3 + master_seed, digestmod=
    hashlib.sha256).digest() + hmac.new(self.master_secret, master_4 + master_seed,
    digestmod=hashlib.sha256).digest() + hmac.new(self.master_secret, master_5 +
    master_seed, digestmod=hashlib.sha256).digest()

    """
    computing AES keys and IVs
    """
    self.mac_length = 20
    if (self.cipher == 0x0035):
        # TLS_RSA_WITH_AES_256_CBC_SHA
        self.client_write_key = keys[40:72]
        self.server_write_key = keys[72:104]
        self.client_write_iv = keys[104:120]
        self.server_write_iv = keys[120:136]
    elif(self.cipher == 0x002f):
        # TLS_RSA_WITH_AES_128_CBC_SHA
        self.client_write_key = keys[40:56]
        self.server_write_key = keys[56:72]
        self.client_write_iv = keys[72:88]
        self.server_write_iv = keys[88:104]
    else:
        print("Unknown cipher!")
```

B Erlang OTP18 source modifications

A few modifications were made to the Erlang source code in order to make it easier to debug the attack script. The changes involve printing out the secret for each connection so that it becomes easy to verify the master secret and other keys that needed to be calculated. The following changes were made:

File	**Location**	**Change**
ssl_handshake.erl	Line 344 (creation of finished message)	Print the Master Secret
ssl_handshake.erl	Line 564 (creation of premaster secret for RSA cipher only)	Print the Premaster Secret
ssl_handshake.erl	Line 1636 (in master_secret function)	Print ClientWrite and ServerWrite keys
ssl_handshake.erl	Line 692 (PreMaster secret)	Master Secret Function