Implementation of Cryptographic Attacks CTF Order of the Oracles

Noa Heller
Shoni Zamir
Daniel Knebel
Elad Raveh

Abstract

One of the things that can impact the security of a protocol, a server, and most other entities, is information leakage. Leaking the wrong information can lead to data exposure, allow impersonation or sabotage entire systems.

The problem with preventing leakage however, is that in plenty of the cases, the designers of the system do not know what information might be considered as a "leakage" or a "side channel", and allow an attacker to do all sorts of malicious things.

From timing to power consumption, error message to cache behavior or even induced faults, side channels can be found everywhere, and mitigating them is one of the things security researchers and commercial entities alike deal with the most.

In our report, we present a CTF challenge revolving around finding those side channels. Given an unknown server, find what information is being leaked, and use it to your advantage.

1 Introduction

The Bleichenbacher attack is a well-known attack on TLS versions allowing for RSA key exchange using the PKCS padding formula. The attack relies upon having an oracle, which lets the attacker know if a sent ciphertext has a valid padding or not.

The attack has returned to haunt the cyber security world a few times, due to the emergence of different oracles and downgrade attacks along the 90s, 2000s and even the 2010s.

Our CTF challenge is constructed of a series of servers, each leaking information allowing an attacker to build an oracle from. The users advance along the challenge by proving they were able to construct a reliable oracle.

After the users have constructed reliable oracles, they are provided with a fast-working oracle and are required to implement the Bleichenbacher attack using it, revealing the secret message and completing the challenge.

2 Related Work

Our implementation and ideation relies heavily on Bliechenbacher's attack, multiple forms of cache attacks, the TLS 1.0 RFC, all detailed in the bibliography.

3 Attack Description

As noted, a required component to use Bleichenbacher's attack is a padding oracle, letting the attacker know if a ciphertext guess has a valid padding under the decryption.

Our attack works in 6 stages. The first 5 stages require the user to prove he can construct a working, reliable oracle from a provided vulnerable server. The stages rise in difficulty, exposing side channels which are harder to find and/or exploit. The first 2 servers are based on historical weaknesses on older versions of TLS, whereas the last 3 are based on side channels we deemed interesting for users to explore.

In the 6th stage users tackle Bleichenbacher's attack [reference to original article] using a provided fast-working oracle. In order to save time, we have chosen to implement the attack solution in batches, meaning the user should send a large number of possible ciphertexts at once to reduce the total time of the attack. The oracle works both with a list of ciphertexts and a single ciphertext.

Descriptions of the servers' leakages and the way to exploit them are listed below.

3.1 Error Message Discrepancy

3.1.1 Problem Description. Like the original Bleichenbacher attack, our first oracle is built upon the side channel exposed by the original TLS implementation – exposing different error messages depending on the validity of the chosen ciphertext's padding.

3.1.2 Solution Description. In this stage, the vulnerable server was implemented using a minimal TLS handshake simulation built on Python sockets and OpenSSL's RSA decryption primitive. When a ciphertext is sent to the server, it attempts to decrypt the message and continue the handshake.

- If the padding is invalid, the decryption fails immediately, and the server sends back a "bad padding" alert.
- If the padding is valid but the MAC is incorrect, the server performs the MAC check and responds with a "bad MAC" alert.
- If both padding and MAC are valid, the server completes the handshake and responds with a plaintext "OK".

These three distinct responses expose a direct oracle: by observing the type of error message (or the success message), the player can immediately classify any ciphertext as invalid padding, valid padding but invalid MAC, or fully valid, in our case the valid/invalid padding was all that mattered.

The solution is an oracle classifying the cipher as valid/ invalid by the messages received. If a "bad padding" alert was received, the cipher should be declared as invalid, in any other case, we declare the cipher is valid.

3.2 Timing Difference

3.2.1 Problem Description. In older versions of TLS, the difference in error messages was patched. However, an invalid padding would abruptly stop the TLS handshake process, leading to a difference in the response time from the server given ciphertexts with valid and invalid messages.

Our oracle is built upon this difference, the user needs to measure the time it takes the server to respond to different ciphertexts, and return an answer given the elapsed time.

3.2.2 Solution Description. Here, the vulnerable server was modified to always return the same alert message regardless of the error, eliminating the direct error-message oracle. However, the order of operations during decryption still leaks information:

When the padding is invalid, the server aborts immediately after the RSA decryption step, sending an alert without doing further work.

When the padding is valid, the server continues to derive keys, perform decryption, remove padding, and verify the MAC before ultimately failing or succeeding.

This creates a measurable timing difference: responses to ciphertexts with valid padding are consistently slower than those with invalid padding. We created a proxy client (so it would run on the same network as the vulnerable server) and the oracle would communicate with the server through it.

To exploit the timing difference, the proxy client records round-trip times (RTT) for server responses and dumps them to the player, along with much more data. The player must build a statistical oracle using the following steps:

(1) Sending the same ciphertext multiple times (e.g., 50 repetitions).
(2) Averaging the measured RTTs to reduce network noise.
(3) Applying a threshold to distinguish "fast" responses (invalid padding) from "slow" responses (valid padding but bad MAC, or simply valid).

By leveraging only timing information, players reconstruct a reliable padding oracle even though the error messages were unified.

3.3 Collaboration

3.3.1 Problem Description. This stage prompts the user to collaborate with another in order to move ahead to the next stage. It requires the users to understand that the leak is visible by using two different endpoints (which correspond to different users).

The server allows the user to send a message, which it processes and lets the user know when it's done. In the meantime, it won't allow the user to send any different messages to process - letting it know it's busy. These answers will not differentiate between valid and invalid ciphers.

However, when the cipher is invalid the server will continue processing messages sent by other users (using different endpoints). When the cipher is valid, the server will not be able to process other user's messages too, responding with 'busy' to everyone.

Therefore, by working collaboratively (or alternatively simply understanding the need to use different endpoints corresponding to different users), the users will be able to build the oracle.

3.3.2 Solution Description. The server uses 'Alice' and 'Bob' as the two users 'collaborating'. It sends the cipher candidate from Alice, and from Bob immediately afterwards. If the server responds to Bob with a 'busy' message, it deduces the cipher was valid. Otherwise, it will return an 'invalid' result.

3.4 Network

3.4.1 Problem Description. This stage requires the user to work on the fourth transport layer, inspecting the different ways in which a TCP connection can be closed. Here, the server's leak is rooted in the way it closes the connection after receiving a valid cipher vs receiving an invalid one.

When the cipher is valid, the server will close the connection 'gracefully' (four way close with FIN). When the cipher is invalid the connection will be closed with RST (abrupt reset - no FIN handshake). The user will be able to visibly see this using the 'Wireshark' application. Using the different connection closings, it will be able to construct the oracle.

3.4.2 Solution Description. The oracle inspects (using python socket package) the type of closing done by the server after sending to it the message in question. If closed gracefully, meaning the function "receive" doesn't return an exception, it returns a "valid padding" statement. Otherwise, it returns an 'invalid padding' statement.

3.5 Cache

3.5.1 Problem Description. Our 5th and final server is based around the fact that given a valid VS invalid padding, the server might continue along different courses of action.

If a ciphertext with a valid padding was sent, our server will call a specific function. Since calling the function requires reading it from the memory, we can expect different reading times if it were to be found in the cache or not.

In this stage, the user is required to find a way to exploit this possible difference and trigger a different timing behavior in cases where the function was called, and cases where the function was not called.

3.5.2 Solution Description. In order to create a working oracle from this server, the user must find a cache eviction set for the function called if the padding is valid. The fastest way to do so is to do the following for each cache line sized part of the function in the memory:

(1) Load the address into the cache (by reading it).
(2) Guess a large eviction set (in our solution we chose 20 times larger than the minimal one to increase the probability of finding an evicting set).
(3) Check the time it takes to read the address after the eviction attempt: a long time period indicates the set has evicted the address, and a shorter one indicates the address was not evicted.
(4) Construct a minimal eviction set

After constructing the eviction set for each address, the user can add them together to create a full minimal eviction set for the entire function, and use the same principal described above to know if the function was called or not, thus providing the user with a valid oracle.

4 Design

4.1 General Flow

The flow is built upon a server-client architecture. Assuming the 'nova' server provided by the university is secure, the server (manager) and the vulnerable servers used are run on it, while the client runs on the user's local computer, interacting with the server using api requests (flask). The server exposes every vulnerable server's URL (ip+port+ endpoint if needed), and its corresponding public key.

The user will be able to ask for hints for every stage, if there are several the last hint will be the most helpful. After understanding what is the server's leak and building an oracle corresponding to it, the client will receive 10 messages generated by the server.

For the messages generation, the server 'flips a coin', deciding whether the message will be of a valid cipher or not (obviously, this information is not passed to the client):

- If valid - it will encrypt a random message using the public key of the vulnerable server.
- If invalid - it will choose a random message as the invalid cipher.

In order to make sure it is not 'accidentally' a valid cipher, it will decrypt the message using the vulnerable server's private key, checking the validity, choosing a different random message until it is indeed invalid (since the odds of accidentally choosing a valid cipher are very low, we do not consider this as a time consuming/computational complexity issue).

The user will need to respond for each message, stating whether it is a valid cipher or not. The answers will be sent to the server, which will either respond to the client with a 'passed' message, exposing the next stage's details, or with a failed message, ending the game.

Lastly, the user will be given the final message to decrypt, using the actual Bleichenbacher attack (more detailed explanation in the attack stage description).

The final message is 'YOU ARE THE MASTER OF ORACLES', making it pretty easy for the user to understand it won without needing additional verifications that might jeopardize the security of the game.

4.2 Error Message Stage

For the error message stage, we designed a minimal TLS server that emulates the behavior of vulnerable OpenSSL versions prior to the Bleichenbacher countermeasures.

The server is implemented in Python using sockets for communication and the cryptography library for RSA decryption. The handshake logic is simplified but faithfully mirrors the critical vulnerability path:

(1) Receive a ClientHello and respond with ServerHello, Certificate, and ServerHelloDone.
(2) On ClientKeyExchange, decrypt the ciphertext using RSA with PKCS1#v1.5 padding.
(3) If the padding is invalid, immediately send a TLS Alert with description bad padding.
(4) If padding succeeds, proceed to MAC verification:
    (a) If the MAC check fails, send a TLS Alert with description bad MAC.
    (b) If the MAC succeeds, complete the handshake and respond with a plaintext "OK".

The client is structured so that players can interact with the server by submitting base64-encoded ciphertexts. The client then parses the server's response, which may include fields such as version, length, level, and description code in the case of a TLS Alert, or a simple plaintext acknowledgment.

In this way, players are exposed to the different error messages (and must interpret the meaning of the alert description codes) or, in rare cases, the success message. This design cleanly exposes the classical Bleichenbacher "error message oracle" and provides an approachable first stage for the CTF.

4.3 Timing Difference Stage

The timing difference stage extends the previous design by simulating the post-patch TLS behavior: error messages are unified, but the order of operations leaks information via timing. To achieve this, the server always responds with the same bad record MAC alert for any failure, whether caused by invalid padding or invalid MAC. However, the computational work differs: Invalid padding: RSA decryption fails early, and the server aborts immediately.

Valid padding but bad MAC: the server derives session keys, decrypts the Finished record with AES, strips padding, and computes a MAC before finally rejecting.

Valid padding and valid MAC: same as above, but the MAC check succeeds and OK is sent back.

This difference naturally produces measurable response time gaps. To make timing more reliable in a remote CTF setting, we introduced a proxy client that runs on the same network as the server. The proxy sits between the player and the server, forwarding ciphertexts but also measuring round-trip time (RTT) using:

```
time.perf_counter().
```

It then reports RTT along with detailed debug information (hashes, randoms, histograms, etc.) back to the player.

Players must implement a statistical oracle: sending the same ciphertext multiple times through the proxy, averaging the RTTs, and classifying responses into "fast" (invalid padding) or "slow" (valid padding). This forces participants to move beyond error messages and exploit side-channel timing differences, as was required in real-world TLS attacks after error messages were patched.

4.4 Network Stage

The vulnerable server uses socket python package to perform its fourth layer functionalities. After checking whether the cipher sent by the user (using normal socket send functions) is valid or not, it responds accordingly with a graceful finish to the connection, using

```
conn.setsockopt(socket.SOL_SOCKET,
                socket.SO_LINGER,
                struct.pack('ii', 1, 0))
```

4.5 Collaboration Stage

The vulnerable server exposes two types of endpoints to the user (using flask): check_status/<user_id> and send_cipher/<user_id>.

Implementation relies on a 'time lock' for each user connection to the server, and a 'global lock'. When the user sends a valid cipher, the server adds 30 to the global lock, ensuring every other attempt to send a cipher to the server will be blocked for the next 30s (a time check is executed every time a user tries to send a cipher). During this time the server will return to every send_cipher POST request a 'server busy' reply.

If an invalid cipher is sent, the server adds 30s to the current user id, such that its reply to another send_cipher from this user will be 'server busy'. To other users' requests it will allow processing.

Check_status endpoint allows the users to check if their cipher (if processed) has finished, in order to be able to effectively follow the server's output. It returns 'processing' for 30s, and then returns a 'done processing' message.

4.6 Cache Stage

The cache server is built upon a simulation of a cache. The cache is defined by 4 elements:

(1) Mapping: A function which maps DRAM addresses to cache sets.
(2) Line Size: The minimal size of a unit saved in the cache.
(3) Associativity: The number of different cachelines that can be saved in a single set.
(4) Sets Number: The amount of different sets the cache has.

The simulation acts like a regular cache, meaning when a new address is mapped to a cache set, if the current set is full, the line with the oldest access is evicted. To simulate cache hits and misses, reading an address from the RAM triggers different waiting times depending on if the line was present in the cache or not.

The server reveals 4 endpoints for the user:

(1) Config: Returns the configuration of the memory (associativity, cache line length, DRAM size) and the cache altering function (size and location in the DRAM)
(2) Read: Allows the user to simulate a reading of a list of addresses in the memory. The endpoint returns a read notice, which tells the user if the read action was successful and why, and the time it took to read all given addresses. The action can fail on an invalid address, or if the user tries to read too many addresses at once.
(3) Write: Like the read endpoint, but doesn't take into account the timing. This is meant to be used when the user wants to load addresses to the cache before checking for timing.
(4) Oracle: The user sends a ciphertext. If the cipher is an encryption of a message with valid padding, the server calls the function. When accessing an endpoint the user must pass as a path argument its own user ID, and the server creates a different cache for each user.

5 Attack Stage

The attack resources given to the user are the cipher of the encrypted solution to the CTF, and the endpoint of a fast oracle, which simply tries to decrypt the message and check if its padding is valid or not.

6 CTF Instructions

To run the CTF, all the user needs to do is:

(1) Make sure it has a connection to the nova (or our chosen remote server to run the CTF servers on)
(2) Install crypto and requests packages
(3) Run in terminal: python client.py

The client scinstructs the user during the game, prompts him to use hints, 'talks' to the server, and tests him when asked to. Clues given to the player for each stage could include an application he should run to observe different behaviors, different help functions, textual clues, etc. Each stage has a different amount of clues as the host sees fit.

7 Conclusions

In our work, and from the history of Bleichenbacher's attack, we can see each good cryptography based attack has 2 parts - the math used to retrieve the desired information, and the means to exploit this mathematical weakness.

If not for side channels and leaks, most attacks won't be possible, and without mathematical weaknesses, side channels would be rather pointless. During our research and brainstorming sessions we have learned that there is no limit to the side channels one can think of. The smallest piece of information can be all that is necessary to indicate some minute difference that can be used in some mathematical algorithm to reveal meaningful information.

To conclude, side channels are everywhere and in every form, and while some may not be useful, there are surely many more that are, and were just not discovered yet.

8 Contributions

Noa: Attack stage, network stage.
Shoni: Collaboration stage, CTF infrastructure and overall flow.
Daniel: Error message server, error message oracle (with Elad), timing stage.
Elad: Cache stage, error message oracle (with Daniel).

References

[1] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, Lecture Notes in Computer Science, vol. 1462, Springer, 1998, pp. 1–12.
[2] Daniel Page. 2002. The Prime and Probe Cache Side-Channel Attack. In *IEEE Transactions on Computers* 51, 5 (2002), 589–602.
[3] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, Lecture Notes in Computer Science, vol. 3860, Springer, 2006.
[4] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 2014)*, 2014, pp. 719–732.
[5] Pepe Vila, Boris Köpf, and José Francisco Morales. 2019. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy (IEEE SP 2019)* (Oakland '19), 2019.
[6] T. Dierks and C. Allen. 1999. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force (IETF), January 1999.