Proxying is Enough

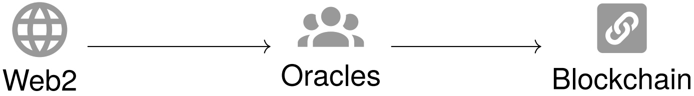
Security of Proxying in TLS Oracles and AEAD Context Unforgeability

Zhongtang Luo[1], Yanxue Jia[1], Yaobin Shen[2], Aniket Kate[1,3]

[1]Purdue University, [2]Xiamen University, [3]Supra Research

SBC 2024

Oracle



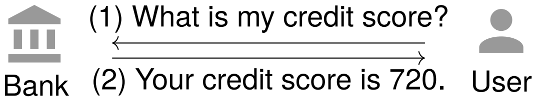
Oracles pull in information from Web2.
(e.g. exchange rate, prediction market, etc.)

Chainlink SUPRA Polyhedra

reclaim Polymarket MANIFOLD

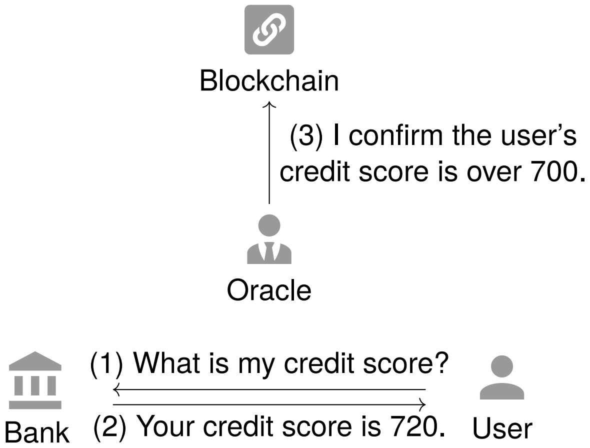
...

Oracle



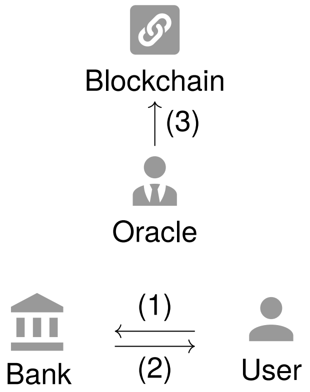
How can we pull in more information?

TLS Oracle



Root of trust: TLS certificate

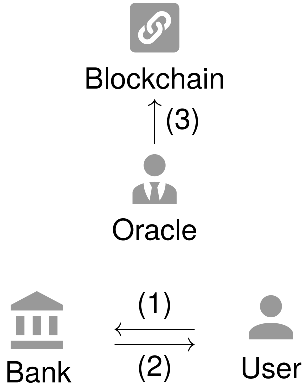
TLS Oracle



Root of trust: TLS certificate

- Caveat:
  TLS is a symmetric encryption!

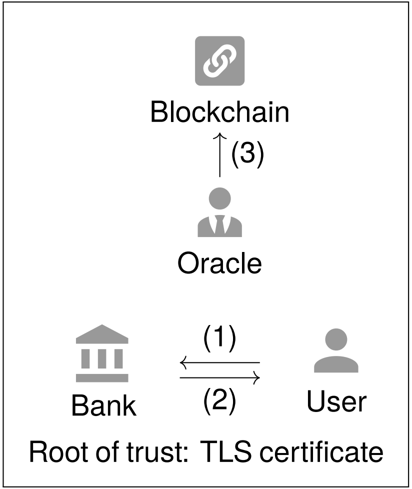
TLS Oracle



Root of trust: TLS certificate

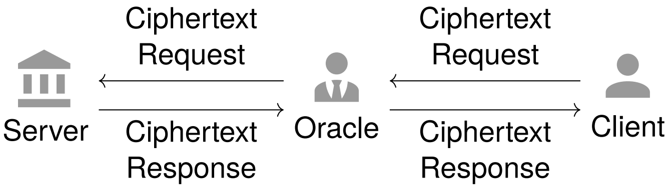
- Caveat:
  TLS is a symmetric encryption!
- An adversarial user can produce **any** transcript from the key.

TLS Oracle



- Caveat:
  TLS is a symmetric encryption!
- An adversarial user can produce **any** transcript from the key.
- Oracle has to be involved in the communication without changing the TLS protocol.

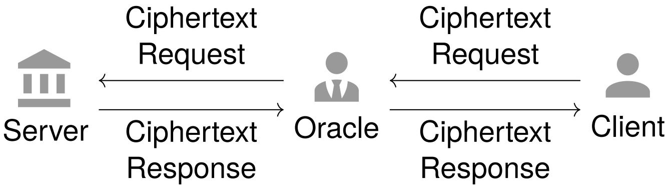
Proxy-Based TLS Oracle



The user reveals the needed part of the plaintext at the end (with some proof).

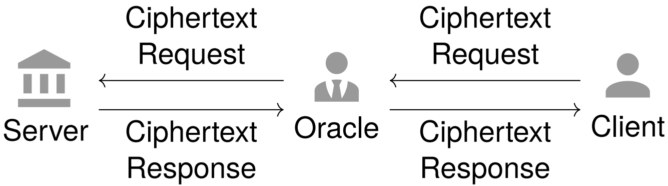
Big question: Is it secure?

Proxy-Based TLS Oracle



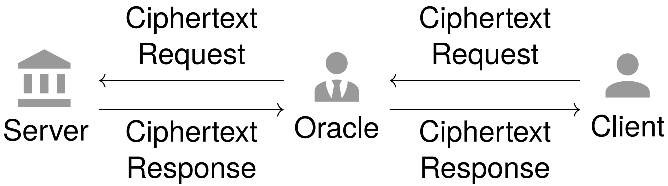
- Conjectured **insecure** since DECO in 2020

Proxy-Based TLS Oracle



- Conjectured **insecure** since DECO in 2020
- Key commitment attack: The ciphertext may decrypt to a different plaintext with a different key.

Proxy-Based TLS Oracle



- Conjectured **insecure** since DECO in 2020
- Key commitment attack: The ciphertext may decrypt to a different plaintext with a different key.
- User can decrypt the same ciphertext into different plaintexts with different keys.

Proxy-Based TLS Oracle

- Conjectured **insecure** since DECO in 2020
- Key commitment attack: The ciphertext may decrypt to a different plaintext with a different key.
- User can decrypt the same ciphertext into different plaintexts with different keys.
- A whole plethora of work on ensuring key commitment:
  - *DECO: Liberating Web Data Using Decentralized Oracles for TLS*
  - *DIDO: Data Provenance from Restricted TLS 1.3 Websites*
  - *Janus: Fast Privacy-Preserving Data Provenance for TLS*
  - *Lightweight Authentication of Web Data via Garble-Then-Prove*
  - *ORIGO: Proving Provenance of Sensitive Data with Constant Communication*
  - ...

But is it really insecure?

Entering the Cryptography Territory...

- Popular fix on key commitment: **Padding**[1]
  i.e. add 128 bytes of 0s to the front of the plaintext.

[1] https://eprint.iacr.org/2020/1456

Entering the Cryptography Territory...

- Popular fix on key commitment: **Padding**[1]
  i.e. add 128 bytes of 0s to the front of the plaintext.
- Rationale: Hard to decrypt the same ciphertext to the same plaintext (0s) with different keys.

[1] `https://eprint.iacr.org/2020/1456`

Entering the Cryptography Territory...

- Hard to decrypt the same ciphertext to the same plaintext (0s) with different keys.
- Concrete example: AES-GCM
  - In AES-GCM, ciphertext block is encrypted by XOR'ing with the AES block cipher:

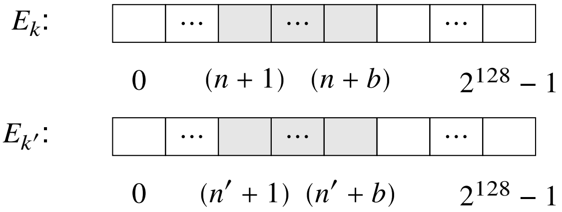
$$\underbrace{c_i}_{\text{ciphertext}} = \underbrace{m_i}_{\text{plaintext}} + \underbrace{E_k}_{\text{cipher}}(\underbrace{n}_{\text{nonce}} + i).$$

  - Since the same ciphertext goes to the same plaintext

$$E_k(n+i) = E_{k'}(n'+i) \ (1 \le i \le b).$$

Entering the Cryptography Territory...

- Hard to decrypt the same ciphertext to the same plaintext (0s) with different keys.
- AES-GCM: $E_k(n+i) = E_{k'}(n'+i)\ (1 \le i \le b)$.
- If we model AES as an ideal cipher (no way to know the permutation without testing the key):



Pretty hard to get b 128-bit blocks to be the same!

Take 1: HTTPS

- Popular fix for key commitment: **Padding**
- Now let us look at HTTPS...

Take 1: HTTPS

- Popular fix for key commitment: **Padding**
- Now let us look at HTTPS...

```
HTTP/1.1 200 OK
Date: Wed, 24 Jul 2024 23:41:36 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
...
```

`https://google.com`

```
HTTP/1.1 200 OK
date: Wed, 24 Jul 2024 23:47:49 GMT
perf: 7402827104
expiry: Tue, 31 Mar 1981 05:00:00 GMT
pragma: no-cache
...
```

`https://twitter.com`

Take 1: HTTPS

- Popular fix for key commitment: **Padding**
- Now let us look at HTTPS...

```
HTTP/1.1 200 OK
Date: Wed, 24 Jul 2024 23:41:36 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
...
```

https://google.com

```
HTTP/1.1 200 OK
date: Wed, 24 Jul 2024 23:47:49 GMT
perf: 7402827104
expiry: Tue, 31 Mar 1981 05:00:00 GMT
pragma: no-cache
...
```

https://twitter.com

- **It turns out that HTTPS is (kind of) padded!**

Take 1: HTTPS

- It turns out that HTTPS is (kind of) padded!
- Specifics vary, but most start with status code and date
  - also recommended by RFC 7231

Take 1: HTTPS

- It turns out that HTTPS is (kind of) padded!
- Specifics vary, but most start with status code and date
  - also recommended by RFC 7231
  - If we consider all status codes (63) and the last hour (3600)...
  - Only 63×3600 possibilities for the first 56 bytes!
  - Define as **variably padded**

Take 1: HTTPS

- It turns out that HTTPS is (kind of) padded!
- Specifics vary, but most start with status code and date
  - also recommended by RFC 7231
  - If we consider all status codes (63) and the last hour (3600)...
  - Only 63×3600 possibilities for the first 56 bytes!
  - Define as **variably padded**
- **We proved that proxy-based TLS is secure for HTTPS.**
  - Covers almost all websites!

Take 2: Non-HTTPS

- Attacks incoming...

(1) Handshake and get key k

$\longleftrightarrow$

(2) User derives plaintexts: $k \rightarrow (\texttt{0xdead}, \texttt{0xbeef})$

Government

(3) Please send me `0xdead`

$\longleftarrow$

(4) Sends `0xdead`

$\longrightarrow$

User

(5) User somehow 'proves' `0xbeef`

Take 2: Non-HTTPS

- Attacks incoming...

(1) Handshake and get key k

$\longleftrightarrow$

(2) User derives plaintexts: $k \rightarrow (\texttt{0xdead}, \texttt{0xbeef})$

Government | (3) Please send me `0xdead` $\longleftarrow$ | User

(4) Sends `0xdead` $\longrightarrow$

(5) User somehow 'proves' `0xbeef`

How likely will this attack happen?

Take 2: Non-HTTPS

Malleable

Account balance
Bank statement
...

Insecure

Fixed

Account number
Age
...

Insecure?

Take 2: Non-HTTPS

- For fixed data, we need only a weaker key commitment property for the cipher suite.
  - We define as **context unforgeability (CFY)**.
  - Informally: For fixed plaintext, hard to find another plaintext that matches the ciphertext
  - Like second-preimage resistance in hash functions

Take 2: Non-HTTPS

AES-GCM	**Chacha20-Poly1305**
AES is a block cipher (reversible).	Chacha20 is based on PRF (not reversible).
Not secure under CFY	Secure under CFY
Cannot be used in non-HTTPS scenarios	**Can** be used in non-HTTPS scenarios with fixed data
✗	✓

Takeaways

Proxy-Based TLS Oracles

HTTPS

Secure!

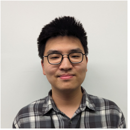
Almost all use case

Non-HTTPS

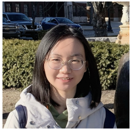
Secure?

Make sure data is fixed
Use Chacha20-Poly1305

Authors



Zhongtang Luo



Yanxue Jia

Yaobin Shen

Aniket Kate

Links

Paper

https://eprint.iacr.org/2024/733

Slides

https://zhtluo.com/paper/Proxying_is_Enoug
h__Security_of_Proxying_in_TLS_Oracles_and
_AEAD_Context_Unforgeability_Slides.pdf