Summary:

At the core of the TLSNotary protocol is dividing the TLS session keys between two parties (Client and Notary) and then using secure two-party computation (2PC) to generate an encrypted and authenticated request which Client sends to a TLS-enabled webserver.

During the run of the protocol neither Client nor Notary know none of the full TLS session keys, they only know their shares of those keys. This preserves the security assumptions of TLS while at the same time allowing Client to prove to Notary the provenance of the TLS transcript.

Below is an in-detail description of each step of the TLSNotary protocol:

Table of Contents

1. Computing shares of the pre-master secret.

In TLS, the first step towards obtaining TLS session keys is to compute a shared secret between the client and the server by running the ECDH protocol (https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman). The resulting shared secret in TLS terms is called the pre-master secret (PMS).

Using the notation from Wikipedia, below is the 3-party ECDH protocol between the server (S) the client (C) and the notary (N), enabling the client and the notary to arrive at shares of PMS.

1. S sends its public key Q_b to C and C passes it to N
2. C picks a random private key share d_c and computes a public key share $Q_c = d_c * G$
3. N picks a random private key share d_n and computes a public key share $Q_n = d_n * G$
4. N sends Q_n to C who computes $Q_a = Q_c + Q_n$ and sends Q_a to S
5. C computes an EC point $(x_p, y_p) = d_c * Q_b$
6. N computes an EC point $(x_q, y_q) = d_n * Q_b$
7. Addition of points (x_p, y_p) and (x_q, y_q) results in the coordinate x_r, which is PMS. (The coordinate y_r is not used in TLS)

(Using the notation from https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Point_addition)

Our goal is to compute $x_r = (\frac{y_q - y_p}{x_q - x_p})^2 - x_p - x_q$ in such a way that a) neither party learns the other party's x value, and b) neither party learns x_r. The parties must only learn their shares of x_r.

Let's start out by simplifying the equation

$$x_r = (y_q^2 - 2y_q y_p + y_p^2)(x_q - x_p)^{-2} - x_p - x_q \mod p$$

Since this is finite field arithmetic, if the result x_r is larger than p, we must "reduce x_r modulo p", i.e assign to x_r the value $x_r \mod p$. The trailing $\mod p$ is always implied from here on out but may be omitted for brevity. (If you're curious, p of the most common EC curve P-256 is a prime number and its value is $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$)

Based on Fermat's little theorem (https://en.wikipedia.org/wiki/Fermat%27s_little_theorem), $a^{-2} \mod p = a^{p-3} \mod p$. Replacing the negative power, we get:

$$x_r = (y_q^2 - 2y_q y_p + y_p^2)(x_q - x_p)^{\mathbf{p-3}} - x_p - x_q$$

- let $A = (y_q^2 - 2y_q y_p + y_p^2)$
- let $B = (x_q - x_p)^{p-3}$
- let $C = -x_p - x_q$

We will compute $x_r = A * B + C$ using the Paillier cryptosystem (https://en.wikipedia.org/wiki/Paillier_cryptosystem) which has some neat homomorphic properties which allow us to:

- add two encrypted values
- multiply an encrypted value by a cleartext value

In the beginning, Notary generates a Paillier keypair and sends the public key to Client. $E()$ denotes an encrypted value. The parties proceed to compute:

1.1. Computing $A = (y_q^2 - 2y_q y_p + y_p^2)$

Notary:

1. sends $E(y_q^2)$ and $E(-2y_q)$

Client:

2. computes $E(y_p^2)$
3. computes $E(A) = E(y_q^2) + E(-2y_q) * y_p + E(y_p^2)$
4. picks random masks M_A and N_A and computes $E(A * M_A + N_A)$
5. sends $E(A * M_A + N_A)$ and $(N_A \mod p)$

(Note that here N_A (as well as N_b and N_B below) is crucial, as without it Notary would be able to factorize $A * M_A$ and learn A.)

Notary:

6. decrypts and gets $(A * M_A + N_A)$
7. reduces $(A * M_A + N_A) \mod p$
8. computes $(A * M_A) \mod p = (A * M_A + N_A) \mod p - N_A \mod p$

1.2. Computing $B = (x_q - x_p)^{p-3}$

Notary:

1. sends $E(x_q)$

Client:

2. lets $b = x_q - x_p$
3. computes $E(-x_p)$
4. computes $E(b) = E(x_q) + E(-x_p)$
5. picks random masks M_b and N_b and computes $E(b * M_b + N_b)$
6. sends $E(b * M_b + N_b)$ and $(N_b \mod p)$

Notary:

7. decrypts and gets $(b * M_b + N_b)$

8. reduces $(b * M_b + N_b) \mod p$

9. computes $(b * M_b) \mod p = (b * M_b + N_b) \mod p - N_b \mod p$

10. sends $E((b * M_b)^{p-3} \mod p)$

Client:

11. computes multiplicative inverse $inv = (M_b^{p-3})^{-1} \mod p$

12. computes $E((b * M_b)^{p-3} \mod p) * inv = E(b^{p-3} * M_b^{p-3} * (M_b^{p-3})^{-1}) = E(b^{p-3}) = E(B)$

13. picks random masks M_B and N_B and computes $E(B * M_B + N_B)$

14. sends $E(B * M_B + N_B)$ and $N_B \mod p$

Notary:

15. decrypts and gets $(B * M_B + N_B)$

16. reduces $(B * M_B + N_B) \mod p$

17. computes $(B * M_B) \mod p = (B * M_B + N_B) \mod p - N_B \mod p$

1.3. Computing A*B+C

Notary

1. sends $E(A * M_A * B * M_B)$ and $E(-x_q)$

Client:

2. computes $E(A * B) = E(A * M_A * B * M_B) * (M_A * M_B)^{-1}$

3. computes $E(-x_p)$

4. computes $E(A * B + C) = E(A * B) + E(-x_q) + E(-x_p)$

5. applies a random mask S_q and sends $E(A * B + C + S_q)$

6. computes his additive PMS share: $s_q = (S_q \mod p)$

Notary:

7. decrypts and gets $A * B + C + S_q$

8. computes his additive PMS share: $s_p = (A * B + C + S_q) \mod p$

The protocol described above is secure against Notary sending malicious inputs. Indeed, because Client only sends back masked values, Notary cannot learn anything about those values.

2. Using garbled circuit to derive shares of TLS session keys from shares of PMS.

Using Garbled Circuit (GC) (https://en.wikipedia.org/wiki/Garbled_circuit), each party provides their PMS share as an input to the circuit. Notary acts as the garbler. Client acts as the evaluator. The circuit combines the PMS shares, computes TLS's PRF function and outputs XOR shares of TLS session keys to each party. Also the circuit builds the Client Finished message and checks the Server Finished message.

In order to prevent malicious garbling, a dual execution protocol is used where Notary and Client take turns garbling/evaluating the same circuit. This reduces the bandwidth requirement by a factor of 5 compared to the state-of-the-art Authenticated Garbling (AG) (https://eprint.iacr.org/2017/030). (For reference: a standard 2KB request using AG would require Client to download 320MB of data from Notary). The dual execution protocol is as follows:

1. N acts as the garbler and C acts as the evaluator of a circuit. C evaluates the circuit, gets encoded output OUT_c. He decodes it and gets $Plaintext_c$. Now C knows what the N's encoded output OUT_n should be.
2. C acts as the garbler and N acts as the evaluator of the same circuit. N evaluates the circuit, gets encoded output OUT_n. He decodes it and gets $Plaintext_n$. Now N knows what the C's encoded output OUT_c should be.
3. C computes $Check_c$ \$ = H(\color{blue}OUT_c\$ \$ | \color{brown}OUT_n\$). C does not reveal $Check_c$ yet but sends a commitment $Comm(Check_c$).
4. N computes $Check_n$ \$ = H(\color{blue}OUT_c\$ \$ | \color{brown}OUT_n\$) and sends it to C.
5. C checks that $Check_c$ \$ == \$ $Check_n$ and decommits $Comm(Check_c$) by sending $Check_c$ to N.
6. N checks the decommitment and then checks that $Check_n$ \$ == \$ $Check_c$.

3. Encrypting the client request.

With each party having their shares of client_write_key and client_write_IV as inputs to the garbled circuit, we use the same technique as in 2. to compute the AES-ECB-encrypted counter blocks. The circuit outputs this value only to Client who xors the request's plaintext with the encrypted counter blocks to get the ciphertext. Client passes the ciphertext to Notary in order to jointly compute a MAC (see below 4.). After MAC is computed, Client sends the encrypted request with MAC to the webserver.

4. Computing MAC of the request using Oblivious Transfer.

In AES-GCM, MACs are computed using the GHASH function described in the NIST publication https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38d.pdf in section 6.4: "In effect, the GHASH function calculates $X_1 \bullet H^m \oplus X_2 \bullet H^{m-1} \oplus \ldots \oplus X_{m-1} \bullet H^2 \oplus X_m \bullet H$." Here X_n is the n-th 16-byte ciphertext block and H^n are powers of H (AES-ECB-encrypted zero), $\oplus$ is XOR, and $\bullet$ is block multiplication as defined in section 6.3.

Our goal is to compute block multiplication of each block X_n with the corresponding power H^{m-n}. The pseudocode of $x \bullet y$ (as per section 6.3) is:

```
1. let res = 0;
2. let R = 0xE1000000000000000000000000000000;
3. for (let i=127; i >= 0; i--)
4.    res = res ^ (x * ((y >> i) & 1));
5.    x = (x >> 1) ^ ((x & 1) * R);
6. return res;
```

Suppose, Client's value is y and Notary's value is x. As can be seen in line 4, depending on whether a bit of y is 1 or 0, a modified x value is either xored to the result or not. The modified x values in line 5 can be pre-computed by Notary in advance: a total of 128 values $x_0, x_1, \ldots, x_{127}$

Notary picks 128 distinct xor masks and appllies them to each value x_n. Using the 1-out-of-2 Oblivious Transfer (OT)(https://en.wikipedia.org/wiki/Oblivious_transfer#1%E2%80%932_oblivious_transfer), for every bit of y, Client asks Notary to obliviously send to him either a masked x_n if the bit is 1 or the mask itself if the bit is 0. In the end, Notary's sum of all xor masks and Client's sum of all values received will be their shares of the product.

This technique is sufficient to compute shares of any power of H, e.g. let's see how the parties compute H^3, where H_n and H_c are Notary's and Client's shares of H.

$$H^3 = (H_c + H_n)^3 = H_c^3 + 3H_c^2H_n + 3H_n^2H_c + H_n^3$$

Since addition is defined as xor, then $3H_c^2H_n = H_c^2H_n \oplus H_c^2H_n \oplus H_c^2H_n = H_c^2H_n$, thus

$$H^3 = H_c^3 + H_c^2H_n + H_n^2H_c + H_n^3$$

The parties will compute locally H_c^3 and H_n^3 respectively. Then, using OT they will compute shares of $H_c^2H_n$ and $H_n^2H_c$. H^3 is the xor-sum of locally computed values and shares computed with OT.

Additional optimizations:

A) Free squaring.

Suppose the parties start with shares of H^x, then the shares $H^{x^2}, H^{x^3}, H^{x^4}, \ldots$ can be computed locally without OT. To demonstrate, if parties start with shares of H^2 and want to compute shares of H^4, it can be seen that the middle terms are all equal to 0, because xoring a value to itself an even amount of times results in 0:

$ H^4 = (H^2_c+H^2_n)^2 = H_c^4 + 4(H^3_cH_n) + 6(H^2_cH^2_n) + 4(H^3_nH_c) + H_n^4 = H_c^4 + H_n^4$

B) Block aggregation.

With parties having H_n and H_c and having computed shares locally using free squaring, they proceed to compute, e.g. $H^3X_3 + H^5X_5$, where X_3 and X_5 are ciphertext blocks corresponding to the powers of H for the GHASH function. (H^2_n means Notary's share of H^2)

Expanding:

$H^3X_3 = (H^2_n + H^2_c)(H_n + H_c)X_3 = H^2_nH_nX_3 + H^2_nH_cX_3 + H^2_cH_nX_3 + H^2_cH_cX_3$ and
$H^5X_5 = (H^4_n + H^4_c)(H_n + H_c)X_5 = H^4_nH_nX_5 + H^4_nH_cX_5 + H^4_cH_nX_5 + H^4_cH_cX_5$

Let $K1_n, K2_n, \ldots$ be values which Notary can compute locally

Let $K1_c, K2_c, \ldots$ be values which Client can compute locally

Given that all values of X_n are known to both parties, we rewrite the expanded sum of $H^3X_3 + H^5X_5$:

$$K1_n + K2_nH_c + K2_cH_n + K1_c + K3_n + K4_nH_c + K4_cH_n + K3_c$$

The parties only need to do two block multiplication using OT (the rest can be computed locally):

$$K2_nH_c + K2_cH_n + K4_nH_c + K4_cH_n = (K2_n + K4_n)H_c + (K2_c + K4_c)H_n$$

The parties could have aggregated more X values, e.g. $H^3X_3 + H^5X_5 + H^9X_9 + H^{17}X_{17}$ would still require only 2 block multiplication using OT.

The best strategy (which involves the least amount of OT) is for the parties to first compute shares of powers 3,5,7,9,11 ..., then apply "free squaring" to those shares and then run block aggregation.

5. Receiving the server response.

When Client receives a response from the webserver, he sends to Notary a hash commitment to the response. Notary reveals to Client Notary's TLS session keys' shares. Client combines his key shares with Notary's key shares and gets full TLS session keys. Client proceeds to authenticate the response and to decrypt it.

6. Contents of the notarization document.

The notarization document includes the following elements signed by Notary:

1. Client's hash commitment to the server response
2. Client's hash commitment to the Client's shares of TLS session keys
3. Client's hash commitment to the Client's share of the PMS
4. GHASH inputs used when computing MAC for the request
5. Webserver's ephemeral pubkey used when computing shares of PMS
6. Notary's PMS share
7. Notary's TLS session keys
8. Notarization timestamp

In addition to the elements signed by Notary, Client also includes in the notarization document the following:

9. Notary's signature over the above 8 elements
10. x509 certificate chain from the webserver's "Certificate" TLS message
11. client_random and server_random values of the TLS session
12. Webserver's signature over ephemeral pubkey and random values from the "Server Key Exchange" TLS message
13. Client's TLS session keys
14. Client's share of the PMS
15. Webserver response with MAC

7. Verifying the notarization document.

Any third party who trusts the Notary's pubkey and has a list of trusted root CA certificates can verify the notarization document by performing the following:

- Check the Notary's signature(9)
- Check x509 certificate chain(10) validity at the time in timestamp(8)
- Verify that signature(12) was made using the public key from the leaf certificate over the random values(11) and over the ephemeral pubkey(5)
- Check that commitment(3) matches Client's PMS share(14)
- Combine PMS shares(6&14) and derive full TLS session keys
- Check that commitment(2) matches Client's session keys(13)
- Check that the full TLS session keys match the sum of Notary's keys(7) and Client's keys(13)
- Decrypt the request contained in GHASH inputs(4) and rule out domain fronting by making sure that HTTP header "Host:" matches the Common Name from the leaf certificate

- Check that the commitment(1) matches the webserver response(15)
- Check MAC of the response(15) and decrypt the response

8. Extending the protocol to hide sensitive data.

The protocol can be extended to enable Client to keep any sensitive data of the TLS transcript hidden from a verifier (a third party) while still proving the authenticity of the TLS transcript. The previous assumption that Notary was a malicious garbler is not valid anymore in this extended protocol. Now, any malicious garbling will be immediately detected by the Client by checking the evaluated circuit's output against the correct output. This allows to use a less expensive semi-honest model where Notary acts exclusively as the garbler and Client as the evaluator. For simplicity of explanation, the below protocol described hiding sensitive data in the webserver response but it can just as easily be applied to hide sensitive data in the Client's request.

The protocol runs exactly as the base protocol up until and including Step 4 after which the protocol changes:

- Parties use GC 2PC to compute encrypted counter blocks for the webserver response. Each party inputs their xor share of the server_write_key (swk) and server_write_iv (siv) into the circuit. The circuit's plaintext output is encrypted counter blocks. The circuit's encoded outputs go to Client.

- Notary must ensure that Client uses the same swk/siv share as the input to the circuit which was used when checking the Server Finished message.

- After Client evaluates the circuit, he obtains encoded outputs for each bit of the encrypted counter blocks. Client selects only those encoded outputs which correspond to the part of the webserver response which Client wants to make public and sends to Notary a commitment to those encoded outputs. (Optionally, Client may also send to Notary a commitment to the secret part of the webserver response if the Client intends to prove in zero knowledge the content of the secret part.)

- Notary sends back the decoding table and his TLS key shares.

- Client gets full TLS keys and decrypts the webserver response. Client uses the decoding table to decode his encoded output from the circuit and checks that the decoded values from the circuit match the decrypted webserver response. If not, the protocol aborts because Notary was cheating.

- Notary signs the decoding table and Client's commitment to the encoded outputs.

(Note that the format of the notarization document in Section 6 must be modified to reflect this extended protocol).

- When the verifier obtains the notarization document, he decodes Client's encoded outputs (for public data) using the Notary's decoding table and obtains encrypted counter blocks (ECB). The verifier xors ECB with the webserver response to get the plaintext which Clients wants to reveal. The remaining part of the webserver response for which Client did not provide encoded outputs remains redacted from the verifier's point of view.