Discorso Lab Sicurezza

Il nome del mio progetto è **IdentityVault**, un prototipo di sistema di autenticazione *passwordless*. L'obiettivo del progetto è dimostrare come sia possibile eliminare le vulnerabilità legate alle password statiche sfruttando la crittografia asimmetrica e i moderni standard di sicurezza biometrica come WebAuthn.

Nel sistema che ho sviluppato, il concetto di 'conoscenza' (la password) viene sostituito dal concetto di 'possesso' (la chiave privata) e 'inerenza' (la biometria). Prima di analizzare il codice, è fondamentale chiarire che, per questa demo, il mio computer sta simulando diversi dispositivi: ogni login di un nuovo utente va immaginato come l'accesso da un dispositivo fisico differente.

---

1. La Base Teorica: RSA e Crittografia Asimmetrica (2 minuti)

Il cuore tecnologico del sistema è l'algoritmo **RSA**. A differenza della crittografia simmetrica, dove esiste una sola chiave per cifrare e decifrare, RSA usa una coppia di chiavi:

1. **Chiave Pubblica**: Può essere condivisa con chiunque (nel nostro caso, viene inviata al server).
2. **Chiave Privata**: Deve restare segreta e non lasciare mai il dispositivo dell'utente.

Il login non avviene inviando la chiave privata, ma tramite una **Firma Digitale**. Il server invia una sfida (una stringa casuale); il client la firma con la chiave privata. Se la chiave pubblica del server riesce a verificare la firma, abbiamo la certezza matematica che il client possiede la chiave privata corrispondente, senza che quest'ultima sia mai transitata sulla rete. Ho scelto chiavi a **2048 bit** perché rappresentano oggi il miglior compromesso tra sicurezza e prestazioni computazionali per il web.

---

2. Analisi Lato Client: `index.html` (3 minuti)

Passiamo al codice. Il client gestisce due fasi critiche: la registrazione e l'autenticazione.

- **Fase di Registrazione**: Quando un utente si registra, il browser deve generare una coppia di chiavi RSA tramite le `Web Crypto API`. Qui entra in gioco la prima decisione progettuale: l'integrazione biometrica. Prima di generare la coppia e mandare la pubblica al server, invoco `navigator.credentials.create()`. Questa funzione attiva il sensore biometrico del dispositivo (TouchID/FaceID). Solo se l'identità è confermata, vengono generate le chiavi e la chiave pubblica viene inviata al server.
- **Fase di Login**: Per prima cosa si chiede sempre l'autenticazione con la biometria, nel caso vada a buon fine, allora il sistema procede a recuperare la chiave private dal dispositivo dell'utente. Successivamente il client richiede una 'challenge' al server. Il client la firma con la propria chiave private e invia al server la firma.

Decisione presa: Ho implementato un **Audit Log** visibile nell'interfaccia. Ho deciso di mostrare in chiaro la challenge e la firma per rendere trasparente il processo: si può vedere come la firma cambi ogni volta anche se l'utente è lo stesso, dimostrando la dinamicità del sistema.

---

3. Analisi Lato Server: `server.js` (2 minuti)

Passando all'analisi del Backend, il server — sviluppato in **Node.js con Express** — è stato progettato secondo il principio di 'sfiducia verso il client': esso agisce come un'entità che deve verificare ogni asserzione tramite prove matematiche. La fase di registrazione comporta il semplice salvataggio della chiave pubblica, poi il processo si divide in due momenti critici:

- **La Gestione della Challenge**: Quando il server riceve una richiesta di accesso, genera un **'Nonce'** (Number used Once) di 32

byte utilizzando la funzione `crypto.randomBytes`. Questo valore casuale viene salvato temporaneamente nel file `users.json`. L'uso di un Nonce è fondamentale: garantisce l'unicità di ogni sessione di login, rendendo il sistema immune ai cosiddetti *Replay Attacks*, poiché ogni firma sarà valida solo per quella specifica sfida e per un brevissimo lasso di tempo.

- **La Verifica della Firma**: Nella rotta `/login-verify`, il server recupera la chiave pubblica dell'utente e la sfida precedentemente memorizzata. A questo punto, utilizzo la funzione `crypto.verify` del modulo nativo `crypto` per confrontare la sfida in memoria con la firma digitale ricevuta dal client. Salvo l'esito della verifica in una costante e reimposto la currentChallenge uguale a null 'bruciando' la sfida e garantendo che non possa essere riutilizzata per accessi fraudolenti. Successivamente controllo l'esito della verifica e se il valore è True, l'utente può entrare

Dal punto di vista teorico, questa operazione si basa sulla proprietà fondamentale della crittografia asimmetrica: una firma generata con una chiave privata può essere validata **esclusivamente** dalla chiave pubblica corrispondente. Se l'operazione di verifica restituisce `true`, il server ottiene la prova matematica della **non-ripudiabilità** e dell'identità dell'utente.

Tutto questo avviene seguendo il principio di **Zero Knowledge**: il server conferma l'identità dell'utente senza che il segreto — ovvero la chiave privata — abbia mai lasciato il dispositivo client o sia transitato sulla rete.

Decisione presa (Sicurezza): Una scelta fondamentale è stata proprio il **reset della challenge**. Non appena una firma viene verificata (o fallisce), il server imposta la challenge a `null`. Questo previene i **Replay Attacks**: anche se un attaccante intercettasse una firma valida, perchè magari non è stato ancora creato un altro NONCE, non potrebbe riutilizzarla perché quella specifica sfida è già stata 'bruciata' dal server.

---

4. Simulazione del Secure Enclave: `wallets.json` (1 minuto)

Un aspetto distintivo della mia implementazione è la gestione del file `wallets.json`. In un'architettura di produzione, le chiavi private risiederebbero in componenti hardware dedicati come il chip **TPM** (*Trusted Platform Module*) o il **Secure Enclave**, aree isolate e inaccessibili al sistema operativo stesso.

Nel mio progetto, ho simulato questo isolamento fisico attraverso una **separazione logica** dei database:

- **`users.json`**: Rappresenta il database del server e contiene esclusivamente le identità e le chiavi pubbliche.
- **`wallets.json`**: Funge da 'portachiavi' del client e custodisce le chiavi private.

Questa distinzione è fondamentale per dimostrare che, in caso di violazione del server, l'attaccante otterrebbe solo materiale pubblico, totalmente inutile per compromettere l'identità degli utenti.

Tuttavia, la sicurezza del wallet non si limita alla sua posizione. Ho implementato una protezione attiva: **l'accesso al `wallets.json` è vincolato al TouchID** (o alla biometria di sistema). Grazie alle API WebAuthn, anche se un malintenzionato avesse accesso al file system del client, non potrebbe utilizzare la chiave privata per firmare una sfida senza lo sblocco biometrico dell'utente legittimo. In questo modo, il sistema garantisce che il possesso fisico del dispositivo non sia sufficiente per l'autenticazione, richiedendo sempre il fattore biometrico come prova di presenza e intenzione dell'utente.

EVENTUALI DOMANDE

L'algoritmo RSA non è una singola stringa, ma un insieme di componenti matematiche. Il formato JWK permette di separarle chiaramente:

- **"n" (Modulus)** e **"e" (Public Exponent)**: Insieme formano la chiave pubblica.

- **"d" (Private Exponent)**: È il cuore della chiave privata. Senza questo valore, è impossibile firmare i dati.
- **"p", "q", "dp", "dq", "qi"**: Sono parametri aggiuntivi (fattori primi e coefficienti) utilizzati per velocizzare i calcoli matematici tramite il "Teorema Cinese del Resto".

2. Codifica Base64URL

Tutti i valori che vedi (come le lunghe stringhe in "d" o "n") sono dati binari convertiti in testo tramite **Base64URL**. Questo viene fatto per due motivi:

- **Compatibilità**: I dati binari puri potrebbero corrompersi se salvati in un file di testo o inviati tramite JSON.
- **Sicurezza nel Web**: La codifica Base64URL evita caratteri speciali che potrebbero creare problemi negli URL o nei database.

3. Metadati e Sicurezza (Il "Contesto")

Il formato JWK non salva solo i numeri, ma spiega anche **come** la chiave deve essere usata:

- **"alg": "RS256"**: Specifica che la chiave deve usare l'algoritmo RSA con l'hash SHA-256.
- **"key_ops": ["sign"]**: Indica esplicitamente che questa chiave può essere usata solo per **firmare** e non, ad esempio, per cifrare file, limitando i danni in caso di uso improprio.
- **"ext": true**: Indica che la chiave è "estraibile", ovvero che il browser permette di esportarla dalla memoria protetta per salvarla (come hai fatto tu in `wallets.json`).

In un sistema a **chiave asimmetrica** come il tuo, la non-ripudiabilità è fortissima perché:

1. **L'unico modo** per generare quella firma specifica è possedere la chiave privata.
2. **La chiave privata** non è mai uscita dal tuo dispositivo (e nel tuo caso è protetta dal TouchID).

3. Quindi, se il server verifica la firma con successo, la "firma" è la prova matematica che l'azione è stata compiuta **proprio da quel dispositivo e da quell'utente**.