Il Fallimento della PKI e la Certificate Transparency

Trascrizione da Presentazione (con approfondimenti su Merkle Trees)

1 Il Problema: Certificati Falsi (ma Validi)

Il pilastro della sicurezza web è che le **Certificate Authorities (CA) SONO fidate**. Ma questo assunto è in discussione.

Esiste un problema molto serio nel mondo reale: i **certificati falsi**.

- Numerosi casi di certificati validi ma "falsi" (cioè emessi senza l'autorizzazione del proprietario del dominio) sono stati scoperti.
- Piccole e medie CA hanno "rilasciato" certificati validi per siti importanti. È stato fatto su richiesta di governi? Per errore? O per una compromissione?
- **Esempi noti:**
  - Certificati falsi per Google emessi da **TurkTrust** (Turchia) nel Dicembre 2012.
  - Certificati falsi emessi da **ANSSI** (Francia) nel Dicembre 2013.
  - CA compromesse come **DigiNotar** (Olanda) e **DigiCert sdn. Bhd.** (Malesia).
- Con minacce potenti (come i governi) e molti attori che possono "commettere errori", la sicurezza del modello PKI sta diventando sempre più debole.

Spiegazione Didattica

Cos'è la PKI e perché la fiducia è un "pilastro"?
La **Public Key Infrastructure (PKI)** è il sistema che ci permette di fidarci dei siti web. Funziona così:

1. Il tuo browser (Chrome, Firefox, ecc.) contiene un elenco di **Root Certificate Authorities (CA)** fidate (es. "UNIVERSETRUST" nella slide).
2. Quando visiti `https://www.google.it`, il server presenta un certificato che dice "Io sono Google, e **LUNATRUST** lo garantisce".
3. Il tuo browser controlla se LUNATRUST è fidata. Magari non lo è direttamente, ma LUNATRUST presenta un suo certificato che dice "Io sono LUNATRUST, e UNIVERSETRUST lo garantisce".
4. Il browser vede "UNIVERSETRUST" nella sua lista fidata, quindi per transizione si fida di LUNATRUST e, di conseguenza, si fida del certificato di Google.

Il **problema** (il "fallimento della PKI") sorge quando una CA di cui ci fidiamo (come TurkTrust o DigiNotar) emette un certificato per "google.com" a qualcuno che *non* è Google. Questo certificato è **tecnicamente valido** (firmato da una CA fidata), ma è **logicamente falso**. Questo permette attacchi Man-in-the-Middle devastanti.

1.1 L'Idea: Un Database Mondiale Gigante

Come gestire le CA malevole o compromesse? L'idea è creare un **gigantesco database (DB) mondiale** che chiunque possa controllare!

Quando Google vede un certificato per "google.it" emesso da LUNATRUST, può controllare questo DB pubblico. Se trova un certificato che non ha richiesto, può dichiarare: **"È Falso! Stanno usando il mio nome!"**.

Ma... come implementare un simile DB gigante?

—

2 Parentesi: I Merkle Trees

Per capire la soluzione, dobbiamo prima capire una struttura dati fondamentale: il Merkle Tree.

2.1 Il Problema: Securizzare e Verificare Dati

Come si può mettere in sicurezza un file di grandi dimensioni?

1. **Step 1:** Creare un'"impronta digitale" (fingerprint) del file utilizzando una **funzione di hash crittografica**. Un file grande diventa un fingerprint piccolo (es. H(FILE)).
2. **Step 2:** "Mettere al sicuro" il fingerprint, ad esempio copiandolo su un supporto sicuro o, meglio, **firmandolo digitalmente**.

Spiegazione Didattica

Hash e Firme Digitali

- **Funzione di Hash (es. SHA-256):** È un algoritmo che prende una quantità di dati qualsiasi (un file, un messaggio) e produce un output a lunghezza fissa (es. 256 bit). È come un'impronta digitale: è (praticamente) impossibile che due file diversi producano lo stesso hash (resistenza alle collisioni) ed è impossibile risalire al file originale partendo dall'hash (resistenza alla pre-immagine).
- **Firma Digitale:** È una tecnologia a chiave asimmetrica. Il proprietario dei dati usa la sua *chiave privata* per "firmare" l'hash. Chiunque può usare la *chiave pubblica* del proprietario per verificare che la firma sia autentica e che l'hash (e quindi il file) non sia stato modificato.

2.2 Il Problema dei Messaggi "a Pezzi" (Chunked)

Questo approccio (un hash per l'intero file) funziona male in sistemi come il peer-to-peer, dove un messaggio è diviso in "chunk" (pezzi) indipendenti che vengono ricevuti spesso fuori ordine e da peer diversi.

- Per verificare la firma digitale dell'intero file, bisogna attendere la ricostruzione COMPLETA del messaggio.
- Cosa succede se un attaccante inietta dei chunk falsi? Non lo si scopre fino alla fine.
- E se volessimo verificare solo una *parte* di un DB enorme? Dovremmo scaricare l'intero DB per verificare la firma singola?

Soluzione debole: Firme per-chunk? Si potrebbe firmare ogni singolo chunk. **PESSIMA IDEA.**

- **Overhead Computazionale:** Una firma (crittografia asimmetrica) è migliaia di volte più costosa (lenta) di un calcolo di hash.
- **Overhead di Storage:** Una firma RSA-2048 è 256 byte, una ECDSA (Bitcoin) è 64 byte. È troppo se i chunk sono piccoli.
- **Nessuna integrità totale:** Un attaccante potrebbe rimuovere o riordinare i chunk (attacchi "strip-type"). Non c'è modo di sapere se abbiamo ricevuto *tutti* i chunk.

2.3 L'Idea di Merkle: L'Albero di Hash (Hash Tree)

La soluzione migliore è una struttura ad albero:

1. Si hanno i dati (le foglie), ad esempio 8 chunk: A, B, C, D, E, F, G, H.
2. Si calcola l'hash di ogni foglia: H(A), H(B), H(C), ... H(H).
3. Si raggruppano gli hash a coppie e si calcola l'hash della loro concatenazione:
   - H[H(A), H(B)]
   - H[H(C), H(D)]
   - H[H(E), H(F)]
   - H[H(G), H(H)]
4. Si ripete il processo salendo di livello:
   - HH[H(A),H(B)], H[H(C),H(D)]

- HH[H(E),H(F)], H[H(G),H(H)]

5. Si continua fino ad avere un singolo hash: la **ROOT**.

È solo questa **singola ROOT** che viene **firmata digitalmente**.

2.4 Verifica di un Singolo Chunk

Il vantaggio? Ora possiamo verificare un singolo chunk (es. C) senza avere tutti gli altri. Abbiamo solo bisogno dei suoi "siblings" (fratelli) lungo il percorso verso la radice.

Per verificare C, un client riceve:

1. Il chunk **C** stesso.
2. Il client calcola **C1 = H(C)**.
3. Il server (o un peer) fornisce i "siblings":
   - **S1 = H(D)** (il fratello di C1)
   - **S2 = H[H(A), H(B)]** (il fratello del genitore di C1)
   - **S3 = HH[H(E),F], H[H(G),H(H)]** (il fratello del nonno di C1)
4. Il client ora può ricalcolare la radice:
   - Calcola **C2 = H[C1, S1]**
   - Calcola **C3 = H[S2, C2]**
   - Calcola **ROOT = H[C3, S3]**
5. Il client confronta la ROOT calcolata con la ROOT firmata digitalmente. Se corrispondono, il chunk C è autentico e appartiene all'albero.

Questo è incredibilmente efficiente: sono necessari solo **Log(N)** "siblings" (hash) per la verifica. Per un milione di chunk, ne bastano circa 20.

Spiegazione Didattica

Applicazioni dei Merkle Trees
Questa struttura è rivoluzionaria e usata ovunque:

- **Blockchain (Bitcoin, Ethereum, ecc.):** I blocchi contengono un "Merkle Tree Root" di tutte le transazioni. Questo permette a un client "leggero" di verificare che una transazione sia inclusa in un blocco senza scaricare l'intero blocco (e l'intera blockchain).
- **Sistemi di file (es. ZFS) e P2P (es. BitTorrent):** Per la validazione efficiente dei "chunk" di file.
- **...e la Certificate Transparency di Google**.

—

3 Certificate Transparency (CT)

Torniamo al nostro problema: un DB gigante per i certificati.

3.1 Perché la Certificate Transparency?

- **Obiettivo:** Rendere (quasi) impossibile per una CA emettere un certificato SSL per un dominio senza che questo sia **visibile al proprietario** di quel dominio.
- **Idea:** Mettere tutti i certificati in una lista (Log) pubblica gigante, che tutti possono consultare.
- **Nuova minaccia:** Come ci assicuriamo di vedere *tutti la stessa lista*? Un attaccante (es. un governo) potrebbe mostrare a noi una lista "pulita" e al resto del mondo una lista diversa (attacco "split world").

3.2 CT in breve

Il sistema aggiunge alcuni passaggi al processo TLS/SSL esistente:

1. **Emissione (lato CA):**
   - La Certificate Authority (CA), prima di emettere il certificato, lo invia (come "Precertificate") a un **Log Server**.
   - Il Log Server aggiunge il certificato al suo Merkle Tree e restituisce un **Signed Certificate Timestamp (SCT)**. L'SCT è una promessa firmata dal Log che dice "Ho ricevuto questo certificato in questo momento e lo includerò".
   - La CA emette il certificato finale per `example.com`, *includendo* l'SCT.
2. **Handshake TLS (lato Client/Browser):**
   - Il browser riceve il certificato da `example.com`.
   - Controlla la validità del certificato (come al solito).
   - Esegue un controllo aggiuntivo: **verifica l'SCT** e che il certificato sia effettivamente presente nella lista pubblica.

Spiegazione Didattica

Chi sono gli attori?
Il sistema CT si basa su tre componenti principali:
- **Log Servers:** Sono i server (gestiti da Google, Cloudflare, DigiCert, ecc.) che mantengono il Merkle Tree "append-only" dei certificati.
- **Monitors (Monitor):** Sono servizi (spesso gestiti dai proprietari dei domini, come Google stessa) che sorvegliano costantemente i Log per trovare certificati sospetti emessi per i loro domini.
- **Auditors (Auditori):** Sono componenti (spesso integrati nei browser) che verificano crittograficamente l'integrità dei Log, controllando che un Log non abbia modificato la sua storia (Consistency Proof) e che i certificati abbiano prove di inclusione valide (Audit Proof).

3.3 Come funziona (tecnicamente)?

- **Usa i Merkle Trees!** Questo garantisce scalabilità e complessità di verifica logaritmica (log(n)).
- È un tipo speciale di Merkle Tree, a volte chiamato *chron tree*:
  - È **"append-only"** (si può solo aggiungere in fondo).
  - Le voci sono *timestamped* (hanno una marca temporale).
  - Questo rende impossibile inserire retroattivamente un certificato.
  - L'albero viene aggiornato periodicamente (es. ogni 24 ore).
  - **Merkle Consistency Proof (Prova di Consistenza):** Questa è una prova crittografica che verifica che due versioni di un Log siano consistenti (cioè, la versione più recente include *tutto* ciò che era presente in quella precedente). Questo garantisce che nessun certificato sia stato modificato, rimosso o inserito "all'indietro" (back-datato) nel log.
  - **Merkle Audit Proof (Prova di Audit):** Questa è la verifica standard del Merkle Tree: usando il certificato e i suoi "siblings" (ottenuti dal log), si può provare che quel certificato è incluso nella radice del log.

3.4 La CT è una Blockchain?

È una domanda comune, perché l'architettura è *identica* a quella del ledger di Bitcoin. Potrebbe anche essere distribuita con un meccanismo di consenso.

Ma la risposta è NO, NON è una Blockchain.

- **Differenza Chiave:** Una Blockchain, per design, impone la **validità**. Se un dato è "nella catena", è considerato valido e non falso.

– La Certificate Transparency **NON contiene necessariamente solo certificati VALIDI**. Google è molto chiara su questo punto.

– Nella CT, la sicurezza deriva dalla **TRASPARENZA**, non dalla validazione a priori.

Spiegazione Didattica

Trasparenza vs. Validità
Questo è il punto cruciale.

– **Blockchain (Logica):** "Se è archiviato, NON è falso". L'atto di inserire un blocco (es. tramite Proof-of-Work) è un atto di validazione.
– **Certificate Transparency (Logica):**
  1. "Se un certificato **NON** è archiviato, allora **È falso** (o meglio, i browser non si fideranno)".
  2. "Ma se **È** archiviato, **potrebbe ANCORA essere falso**!" (come il certificato Google emesso da TurkTrust).

Il punto è che, essendo *pubblicamente* archiviato, il vero proprietario (Google) può trovarlo e agire immediatamente per **revocarlo**, ad esempio pubblicandolo in una Certificate Revocation List (CRL). La CA che lo ha emesso subisce un danno reputazionale enorme e rischia di essere rimossa dalla lista fidata dei browser.