Efficient Data Structures for Tamper-Evident Logging

Scott A. Crosby       Dan S. Wallach

*Rice University*

Presented at USENIX Security 2009
Reliance on logs

Assume the adversary doesn’t tamper with the logs.
Can we trust the logs?

The attacker may modify the log file to cover their traces!

**Goal**: An event, once correctly logged, cannot be *undetectably* hidden or modified.
Send logs to a **trusted** central server
Allow the central server to be untrusted.

Ingredients:

1. Auditing
2. History Tree
High-level design

• Logger (central server)
  - Stores logs
• Clients
  - Generate logs
• Auditors
  - Verify the correct operation of the logger
• Logs come in
• Commitments go out
Commitments

• Each commits to the entire past. Example construction [Kelsey, Schneier]:
  \[ C_n = H(C_{n-1} || X_n) \]

• They are signed by the logger

\[ \begin{align*}
& \quad C_{n-4} \quad \quad \quad \quad C_{n-3} \quad \quad \quad \quad C_{n-2} \quad \quad \quad \quad C_{n-1} \\
& \downarrow \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \downarrow \\
& X_{n-4} \quad \quad \quad \quad X_{n-3} \quad \quad \quad \quad X_{n-2} \quad \quad \quad \quad X_{n-1}
\end{align*} \]
We don’t trust the logger!

- Does $C_{n-3}$ really contain $X_{n-3}$?
- Do $C_{n-2}$ and $C_{n-1}$ commit the same historical events?
- Is the event at index $i$ in the log defined by $C_n$ really $X_i$?
Example: log forks

- What if the logger rolls back the log and adds on different events?
• Check the returned commitments
  - For correct event lookup
  - For consistency
Two kinds of audits

- Membership auditing
  - Verify proper insertion
  - Lookup historical events

- Incremental auditing
  - Prove consistency between two commitments
Who does what?

• **Clients must redistribute their received commitments from the logger to auditors.**

• A host can be both client and auditor at the same time.

• Auditing strategies are not discussed in detail.
Making audits cheap

- Logs are stored in a **history tree**
Given $(3, c_7)$ return $(x_3, P)$, where $P$ is:

Valid if root $== c_7$.

$P$ takes $O(\log n)$ to build
Given \((C_3, C_7)\) return \((P)\), where \(P\) is:

Valid if:
- \(P\) is consistent with \(C_7\)
- \(P\) is consistent with \(C_3\)

\(P\) takes \(O(\log n)\) to build
Merkle Aggregation

History trees can be extended to annotate events with attributes.

**Application**: support content searches.

- Max()

Find all transactions over $6$
Performance

• Insert performance: 1,750 events/sec
  - 2.4%: Parse the log event
  - 2.6%: Insert the event to the tree
  - 11.8%: Get root commitment
  - 83.3%: Sign commitment

• Proof generation:
  - With locality (all events in RAM):
    • 10,000-18,000 incremental proofs/sec
    • 8,600 membership proofs/sec
  - Without locality
    • 30 membership proofs/sec
• History trees allow the logger to store log events and generate integrity proofs efficiently.

• Other hosts (auditors) need to demand those proofs to ensure the logs are not tampered.

• Result: the logger can be untrusted (but at least one auditor needs to be honest).
Discussion

• No security analysis: what happens if a client colludes with the logger? What if the secret key of the logger is compromised?

• No full-system evaluation with multiple hosts. Network overhead? Overhead of redistributing commitments with gossip? Scalability?

• No auditing strategies are presented. What kind of audits, from whom and how often should be asked to the logger? What happens when tampering is detected? Lying auditors?