🔧 Security Re-engineering for Databases: Concepts and Techniques

🚧 What is Security Re-engineering?

Security Re-engineering is the process of revisiting and improving the security architecture of an existing database system — without having to rebuild it from scratch.

📌 Think of it like retrofitting an old house with modern security features (CCTV, biometric locks) — same house, new safety.

🧠 Why Is It Needed?

Traditional databases:

- Were designed without security as a core concern.
- Focused mainly on performance, availability, and functionality.
- But modern concerns — privacy, regulations (like GDPR), cloud usage — demand tighter and smarter security.

> ✅ Security Re-engineering helps adapt legacy databases to modern security requirements.

🎯 Goals of Security Re-engineering

1. Add or upgrade security mechanisms (e.g., access control, encryption).
2. Preserve existing data and applications as much as possible.
3. Minimize system downtime and performance impact.
4. Maintain data integrity and confidentiality.

🏗️ Components of Security Re-engineering

1. Security Policy Definition

- Define what to protect and from whom.
- Policies may include:
    - Who can access what data?

    - Under what conditions?
    - What to log or audit?

> 🔍 **Example:**
> **Only managers can access salary records; HR staff can edit, but not delete them.**

2. Policy Formalization

- **Convert the policy into a formal language that the system can understand.**
- **Examples:**
    - **Logic-based rules**
    - **Role/attribute-based expressions**

3. Policy Refinement

- **Map high-level security goals (e.g., "protect private info") into enforceable low-level mechanisms (e.g., encrypt `ssn` column).**
- **Ensure consistency and resolve policy conflicts.**

4. Policy Enforcement

Applies actual technical measures like:

- **Fine-grained Access Control (down to rows/columns)**
- **Encryption for confidential fields**
- **Query rewriting to enforce user-specific views**
- **Views to hide sensitive data dynamically**

> 🧑‍💼 **Example:**
> **When a user queries employee data, the system shows only non-sensitive columns unless their role is "Admin".**

🧰 Techniques Used in Security Re-engineering

✅ 1. Access Control Integration

- Apply DAC, RBAC, or ABAC to protect tables, rows, or fields.

✅ 2. Use of Views

- Create customized views that enforce policy without changing the original schema.
- Users only “see” what's safe for them.

✅ 3. Query Modification (Query Rewriting)

- Automatically rewrite user queries to include security conditions.

Example:
User runs: `SELECT * FROM Employee`
Rewritten as: `SELECT Name, Position FROM Employee WHERE Department = 'Sales'`

✅ 4. Data Encryption

- Encrypt sensitive fields (e.g., passwords, medical records).
- Ensure that unauthorized users can't read data even if they access it.

✅ 5. Audit Trails

- Keep logs of who accessed what and when.
- Helps in tracking suspicious or unauthorized behavior.

⚠️ Challenges in Security Re-engineering

Challenge	Explanation
🧩 Complex Legacy Systems	Old systems may have poor documentation or tight coupling
🧪 Conflicting Security Needs	Different users/departments may need different policies
🐢 Performance Overhead	Adding security features may slow down queries
🔁 Maintaining Data Consistency	Encryption or policy changes must not corrupt data or logic
🔄 User Acceptance	Users must still be able to work smoothly post-changes

🧠 Example Scenario

Imagine a university database where:

- **It was originally built just to store student records.**
- **Now, it must support:**
  - **Privacy rules (e.g., FERPA)**
  - **Role-based access (students vs. faculty vs. admin)**
  - **Auditing and data retention**

Security Re-engineering would:

- **Define rules like “only faculty can see grades”**
- **Encrypt student SSNs**
- **Add access logs**
- **Use views to give students access only to their own records**

🔚 Summary

Aspect	**Description**
What	**Upgrading security in existing database systems**
Why	**To adapt to new privacy laws, security threats, and requirements**
How	**Using policies, views, query rewriting, access controls, encryption**
Goal	**Protect data with minimal disruption or redesign**
Challenges	**Legacy complexity, performance, conflict resolution**

💧🔐 Database Watermarking for Copyright Protection

🎯 What Is Database Watermarking?

Database Watermarking is a technique used to embed hidden copyright information into a database to:

- **Prove ownership**
- **Deter or detect unauthorized use**
- **Provide legal evidence in case of data theft**

> 🔏 **Similar to watermarking an image or video, but applied to structured data in relational databases.**

🧠 Why Watermark Databases?

- **Digital data is easy to copy, share, or sell.**
- **Ownership disputes over data are hard to prove.**
- **Security mechanisms like access control or encryption can prevent unauthorized access, but don't help if someone leaks or copies the data.**
- **Watermarking adds a "signature" inside the data itself — invisible but verifiable**

🔐 Key Goals of Database Watermarking

Goal	Explanation
🔐 Robustness	Watermark should survive common database operations (like sorting, joins, or slight edits)
🔎 Imperceptibility	Watermark should not noticeably alter the data
🧪 Verifiability	Owner should be able to extract and prove the watermark
🛡️ Security	Attackers should not be able to detect or remove the watermark easily

---

🧬 Types of Watermarks

1. Bit Watermarks

- **Embed a binary string (like `101010`) representing owner ID or license number.**

2. Image Watermarks

- **Embed a digital image/logo into numerical data (less common in DBs, more in multimedia).**

🧩 Where Is the Watermark Embedded?

Usually in non-key attributes of numeric or text fields:

- **Numerical fields: Slightly modify the last decimal digits**
- **Text fields: Introduce tiny, non-visible edits (like extra spaces or punctuation)**

> ✅ **Must not affect query results or data integrity significantly**

🧰 Techniques for Database Watermarking

✅ 1. Distortion-Based Techniques

- **Modify selected data values using an algorithm based on:**
  - **Secret key**
  - **Watermark bits**

> 📌 **Example:**
> **In a sales DB, change the last digit of the "price" field for certain rows to embed bits of the watermark.**

✅ 2. Noise-Based or Signal Encoding

- **Embed a pattern or statistical noise into the data distribution.**

✅ 3. Tuple Reordering

- **Change the order of rows in a non-functional way to encode data.**

✅ 4. Content-Based Watermarking

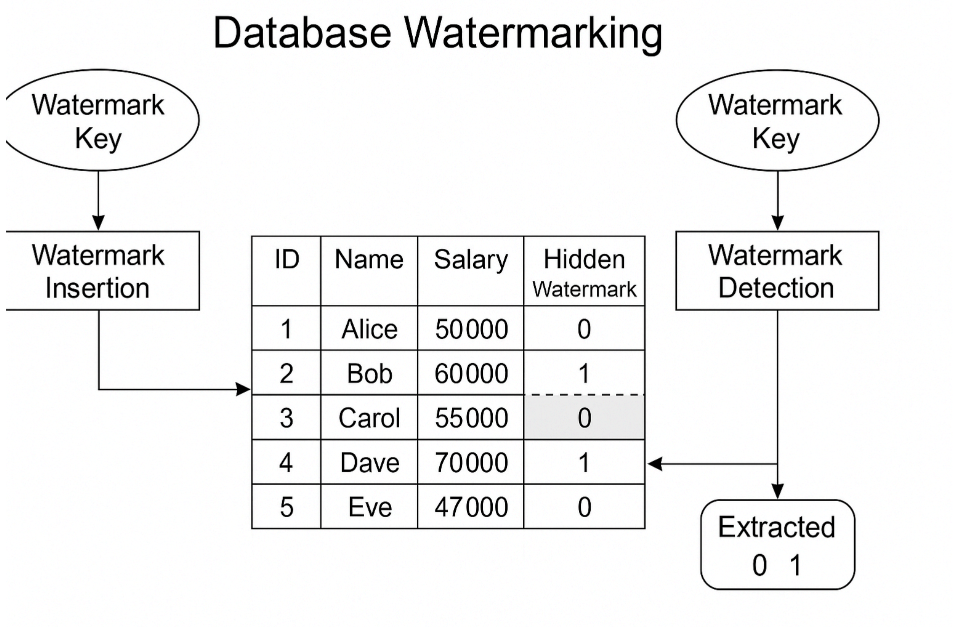
- **Use data content (like employee ID or timestamps) to decide where to embed the watermark.**

🛠️ Watermark Insertion Process

1. **Choose watermark key and encoding strategy**
2. **Select target tuples/fields based on the key**
3. **Embed watermark bits by modifying the data**
4. **Store/record the key for future verification**

🔍 Watermark Detection (Verification)

- **The owner runs a detection algorithm using the original key**
- **If watermark bits are extracted correctly, ownership is proven**
- **Even partial matches can support ownership claims (robust detection)**



ID	Name	Salary	Hidden Watermark
1	Alice	50000	0
2	Bob	60000	1
3	Carol	55000	0
4	Dave	70000	1
5	Eve	47000	0

🧪 Example Scenario

Use Case:

A company publishes a product pricing database online for partners.

Problem:

Someone downloads it, changes the logo, and republishes it as their own.

Solution:

The company:

- Adds a hidden watermark by adjusting price values' last decimal
- Keeps a secret key and embedding algorithm

Later, the company extracts the watermark and proves it in court. ✅

⚠️ Challenges & Considerations

Issue	Description
⚖️ Legal Validity	Not all courts may accept watermarking as proof
🐢 Performance Overhead	Embedding should not slow down DB operations
💥 Resistance to Attacks	Attackers may try to remove or distort the watermark
🧮 Data Utility	Watermarked data must still be accurate and useful

---

✅ Advantages

- Doesn't rely on external protection
- Works even if data is stolen and republished
- Can be applied to publicly shared datasets

🔚 Summary

Aspect	Description
What	Embed hidden info in DB to prove ownership
How	Slight modifications in data fields
Why	Deter theft, prove IP rights, track leaks
Where	Numeric fields, text fields, tuple order
Verification	Use secret key to extract embedded info

Challenge	Must balance visibility, robustness, and legality

🗃️🔐 Trustworthy Records Retention

📁 What is Records Retention?

Records Retention refers to the practice of storing data for a legally or operationally required period, and ensuring it can be trusted, retrieved, and used as evidence when needed.

🔍 Examples of records:

- Tax filings
- Medical records
- Legal documents
- Employee contracts
- Financial transactions

🤝 What Does "Trustworthy" Mean?

A trustworthy record is one that is:

1. Authentic – It is what it claims to be.
2. Reliable – Accurately reflects the activity or fact it records.
3. Integrity-preserved – Hasn't been altered or tampered with.
4. Usable – Can be retrieved and understood when needed.

📜 Why Is It Important?

- 🔒 For legal defense (e.g., proving contract terms)
- 📅 To meet regulatory requirements (e.g., GDPR, HIPAA, SOX)
- 🧾 To support audits and investigations
- 📉 To minimize risks of data loss, forgery, or liability

🎯 Goals of Trustworthy Retention

Goal	Purpose
🛡️ Security	Prevent unauthorized changes or deletions
⏳ Longevity	Ensure data remains accessible long-term
🧾 Auditability	Enable tracing of who accessed/modified records
📥 Controlled deletion	Remove data after legal retention period ends

🛠️ Key Techniques and Concepts

✅ 1. Access Control

- Define who can read, write, delete, or update records.
- Prevent unauthorized actions.
- Example:
    - Only HR can modify employee contracts.
    - Only auditors can view financial logs.

✅ 2. Tamper-Evident Storage

- Use cryptographic hashes, digital signatures, or blockchains to detect changes.
- If a record is changed, it's detectable and provable.

Example:
A document is hashed and stored with its hash. If tampered, the hash won't match — trust is broken.

✅ 3. Retention Policies

- Define how long a record must be kept.

- **Example:**
  - **Tax records: 7 years**
  - **Medical records: 10 years after patient's last visit**

✅ 4. Immutable Storage

- **Store records in append-only form (cannot be overwritten).**
- **Some systems use WORM (Write Once, Read Many) storage.**

Example:
A bank log file is saved so no one can modify past transactions.

✅ 5. Logging and Auditing

- **Track who accessed or changed a record and when.**
- **Helps in forensic investigation or compliance reporting.**

✅ 6. Metadata Management

- **Store details about the records:**
  - **Who created it?**
  - **When?**
  - **Under what context?**
  - **Retention period?**
  - **Access level?**

Metadata helps interpret records correctly over time.

📂 Challenges in Trustworthy Retention

Challenge	Description
🧮 Data Volume	Large data makes storage expensive
🔁 Format Obsolescence	Old file formats may not be readable
❌ Human Error	Improper labeling, deletion, or migration
🕵️ Insider Threats	Authorized users misusing access
⚖️ Legal Uncertainty	Laws vary by country and may change over time

---

🏥 Example Use Case

A hospital must retain:

- Medical records for 10 years after a patient's last visit.
- Billing records for 7 years.

The system must:

- Prevent doctors from deleting history.
- Allow only billing team to access invoices.
- Detect if records are changed after creation.
- Automatically delete records after retention period expires.

✅ If audited, the hospital must prove that records are authentic, complete, and untampered.

🛠️🦠 Damage Quarantine and Recovery in Data Processing Systems

❓ What Is This About?

In any data processing system, malicious actions or system failures can cause data damage — intentionally (e.g., by hackers) or unintentionally (e.g., bugs or crashes).

This topic focuses on:

- Detecting that damage has occurred
- Quarantining (isolating) damaged data
- Recovering the system back to a safe state

> ✅ The goal is to ensure data integrity, availability, and trust after a security incident or failure.

🔥 Examples of Data Damage

- A virus changes salary records in a payroll system.
- A malicious user updates grades in a university database.
- A buggy update deletes product prices in an e-commerce site.

🚨 Why Is This Important?

Because damage can:

- Corrupt critical data
- Spread across multiple tables and systems
- Cause wrong decisions or actions
- Lead to data loss or leaks

> 🧠 The longer damage goes undetected, the bigger the impact.

🎯 Goals of Damage Quarantine & Recovery

Goal	Description
🔍 Detect damage	Identify what was changed and when
🧪 Isolate damage	Prevent it from spreading further

🔁 Recover safely	Restore only damaged data — not everything
🧼 Clean propagation	Fix data affected by damaged records

🛡️ Key Concepts

✅ 1. Intrusion Detection Systems (IDS)

- Detect suspicious activities (e.g., unauthorized SQL queries)
- Alert admins when anomalies occur

> 🛠️ Example: An IDS sees a student trying to update the `grades` table without proper permissions.

✅ 2. Damage Assessment

- Identify exactly which records or parts of the database were affected.
- May use:
    - Audit logs
    - Transaction logs
    - Dependency tracking

> Example: If one infected record affected others via a trigger, those downstream records must also be marked.

✅ 3. Damage Quarantine

- Temporarily isolate or lock damaged data to prevent use or further contamination.
- Users are prevented from:
    - Reading infected rows

    - Executing dependent queries
    - Writing over affected areas

🧼 Think of it like putting damaged rows into digital "isolation" until cleaned.

✅ 4. Data Recovery

Two main types:

🔄 A. Full Restore (Traditional Backup)

- Revert to a previous full backup
- Problem: You lose all recent data, even if only a small part was damaged

🧠 B. Selective Recovery (Preferred)

- Use undo logs and redo logs to:
    - Undo only malicious transactions
    - Keep legitimate ones

✅ More precise and efficient — minimizes data loss

✅ 5. Damage Propagation Tracking

- Tracks how damaged data spread across the system via:
    - Triggers
    - Foreign key relationships
    - Applications

Example: An infected price value may corrupt totals in sales reports — those reports also need fixing.

💥 Example Scenario

A university's student database is hacked:

- **A hacker changes one student's grade to “A+”**
- **That change affects:**
    - **GPA**
    - **Class rank**
    - **Scholarship eligibility**

Response Steps:

1. **Detect unusual update via audit logs**
2. **Quarantine the grade, GPA, rank, and scholarship data**
3. **Assess which rows were affected downstream**
4. **Undo only the hacker’s transaction**
5. **Verify system state is back to normal**
6. **Log the incident for future protection**

⚠️ Challenges

Challenge	Description
🕵️ Late detection	Damage may go unnoticed for a long time
🧩 Complex dependencies	Damage can spread in subtle ways
💽 Log overhead	Fine-grained logging can slow down performance
🔁 Partial failures	Some data may be partially recoverable, others not
🛑 Stopping propagation	Needs fast response to isolate contaminated data

🏥🔐 Hippocratic Databases: Current Capabilities and Future Trends

🧠 What Is a Hippocratic Database?

A Hippocratic Database (HDB) is a type of database designed to protect the privacy of users' personal data by enforcing data minimization and purpose-based access control.

> 🧾 **Inspired by the Hippocratic Oath in medicine, which says: "First, do no harm."**

In this context:

> **"First, don't misuse personal data."**

🧠 What Is a Hippocratic Database?

A Hippocratic Database (HDB) is a type of database designed to protect the privacy of users' personal data by enforcing data minimization and purpose-based access control.

> 🧾 **Inspired by the Hippocratic Oath in medicine, which says: "First, do no harm."**

In this context:

> **"First, don't misuse personal data."**

🎯 Goal of Hippocratic Databases

To make privacy a core part of database behavior:

- **Collect only necessary data**
- **Allow access only for specific, approved purposes**
- **Keep a record of what data is used for what purpose**

> ✅ **Enforce Privacy as a Policy, not just a technical feature.**

📜 Core Principles of Hippocratic Databases

According to the original concept, there are 10 key principles (like commandments) that HDBs should follow:

#	Principle	Explanation

1	Purpose Specification	Data must be collected for specific reasons
2	Consent	Users must give permission
3	Limited Collection	Only collect what's necessary
4	Limited Use	Only use data for approved purposes
5	Limited Disclosure	Don’t share data without consent
6	Limited Retention	Don’t store data longer than needed
7	Accuracy	Keep data correct and up to date
8	Security	Protect data from unauthorized access
9	Transparency	Users can see what data is collected and how it’s used
10	Accountability	Organizations must be responsible for enforcing policies

---

🧩 How Is It Different from Traditional Databases?

Feature	Traditional DB	Hippocratic DB
Focus	Performance, availability	Privacy, purpose-aware access
Access control	Based on identity/role	Based on purpose + consent
Logs	Optional	Mandatory and transparent
Consent management	External	Integrated in the DB

🛠️ Current Capabilities (As per the Book)

✅ 1. Purpose-Based Access Control

- Every query includes a purpose.
- DB checks whether the user has consented to that purpose.
- If not → access is denied.

Example:
A hospital staff queries a patient’s phone number.
If the purpose is “appointment reminder” — allowed.
If the purpose is “marketing” — denied (no consent).

✅ 2. Consent Storage

- **The DB stores:**
  - **What data a user has consented to share**
  - **For what purpose**
  - **For how long**

📌 **Example:**
John allows his email to be used only for billing from 2023 to 2025.

✅ 3. Query Annotation

- **Queries must include metadata like:**
  - **Who is accessing?**
  - **Why (purpose)?**
  - **Is there consent?**

✅ 4. Auditing & Logging

- **Logs every access attempt.**
- **Useful for:**
  - **Compliance**
  - **Investigating data misuse**

✅ 5. Policy Enforcement Engine

- Checks whether each data access respects privacy policies.

🔮 Future Trends & Research Directions

🚀 1. Automated Policy Learning

- Use machine learning to learn consent patterns and suggest policies.

📊 2. Data Usage Transparency Dashboards

- Allow users to see how their data is being used, by whom, and for what.

🤝 3. Interoperability

- Allow different systems (e.g., banks, hospitals, insurance) to share data with cross-platform consent and control.

🔒 4. Privacy-Preserving Computation

- Combine HDBs with techniques like:
    - Homomorphic encryption
    - Secure multi-party computation
    - Differential privacy

📦 5. Integration with Blockchain

- Immutable logs of consent and data usage.
- Greater transparency and tamper-resistance.