EDB POSTGRES AI

Best Practices for Security

Nick Ivanov

enterprisedb.com

Table of contents

Introduction

In today's interconnected digital landscape, the lifeblood of a large organization is data, a critical and often irreplaceable asset that fuels decision-making, drives innovation, and defines competitive advantage. The integrity, confidentiality, and availability of this data are paramount to business continuity and stakeholder trust. As organizations increasingly rely on powerful and flexible database systems to manage vast quantities of information, the security of these systems becomes a nonnegotiable priority.

EDB Postgres® AI, a robust, modern data platform, is built on the foundation of the open source object-relational database system, PostgreSQL, known for its extensibility and standards compliance. It has become a cornerstone of many enterprise IT infrastructures. However, while EnterpriseDB (EDB) software provides a strong foundation for data management, its security is not a monolithic feature; it requires a proactive and deliberate strategy.

Securing business-critical data in a large-scale PostgreSQL environment demands a multifaceted approach that extends beyond basic user permissions. A single point of failure in any part of the data lifecycle—from creation and storage to access and analysis—can expose the entire organization to significant risk, including data breaches, intellectual property theft, and noncompliance with regulatory frameworks and industry standards, such as PCI DSS, GDPR, or the Gramm-Leach-Bliley Act. This white paper is dedicated to exploring the layered security model essential for safeguarding enterprise data within a PostgreSQL ecosystem. It will delve into a comprehensive framework that addresses security at the database, application, and infrastructure levels, providing a guide for large organizations to build resilient and defensible data security postures.

Layered security model

The layered security model, often referred to as defense in depth, is based on the principle that security mechanisms should be applied in multiple layers so that if one fails or is compromised, other layers can still prevent a breach. There is no single security technique that covers all possible attack scenarios, thus security requires a multilayered approach.

The fundamental principles driving the layered model include:

- **Defense in depth**: This core concept acknowledges that a single security mechanism is generally insufficient, requiring multiple layers of security to be implemented. The approach aims to reduce the risk of compromise by ensuring that the failure of one layer does not compromise the database system entirely.
- **Mitigation of attack vectors:** A multilayered strategy protects against various attack factors. By establishing multiple barriers, the system is protected consistently against threats.
- **Analogy to physical security:** This concept is comparable to the way banks operate, using several layers of protection, such as doors, vaults, and safe deposit boxes, to safeguard assets.
- **Principle of least privilege:** This is a key component of a layered security strategy. It dictates that every user, process, or task should be granted only the minimum rights required to perform its job, thereby constraining the scope of damage if that entity is compromised.

Layers of security implementation

The layered model encompasses multiple areas of infrastructure, ranging from the physical environment to the application level, that particularly concern database security:

1. Physical layer

This layer secures the physical location where the database system resides.

- **Access restriction:** Physical access must be restricted to rooms or racks using measures such as locks, card reader access, security guards, and physical intrusion detection systems (e.g., motion sensors or cameras).
- **Secure hardware:** Keep server consoles and KVMs locked, and set BIOS/UEFI passwords.
- **Media protection:** Backup media must be treated as sensitive assets, stored encrypted, and kept offsite.

2. Network layer

The network layer involves controlling who can connect to the host and how data travels.

- **Network isolation:** Databases should run on private subnets with no direct public IP exposure. In exceptional circumstances, when direct, low-latency access by external applications are required, the database server may be placed within a demilitarized zone (DMZ).
- **Access control:** Network access should be limited to specific application subnets to the database port. This is achieved by using network firewall rules and the PostgreSQL host-based authentication configuration file (`pg_hba.conf`).
- **Encryption in transit:** Database authentication handshake and data flowing over the network should be protected by SSL/TLS encryption using reliable cipher algorithms.

3. Host (operating system) layer

This layer focuses on securing the operating system where the database instance runs.

- **Host access control:** Allowing access to the host operating system must be protected. If an individual has unrestricted access to the host, many other protections become worthless.
- **OS hardening:** Perform patching of the OS regularly and enable security mechanisms such as SELinux and AppArmor in enforcing mode.
- **Dedicated accounts:** Run the database instance under a dedicated OS account (e.g., postgres/enterprisedb), and never run it as root.
- **File permissions:** Restrict file permissions on the database data directory and transaction log directories, configuration files, and logs to the instance owner only.

4. Database layer

Within the database layer reside the native security features within the database software itself, focusing on data access and protection. Their basis is described by the so-called **AAA model**, defending authentication, authorization, and accounting (or auditing) of database activities.

- **Authentication:** Verification of the user's identity. This includes mechanisms for managing users, roles, and password profiles.
  - **Password management:** Enforcing security features such as EDB Postgres Advanced Server password profiles to manage failed login attempts, lockouts, complexity, and reuse rules.
- **Authorization:** Verification of what the authenticated user is allowed to do.
  - **Data access control:** Restrictions are applied at granular levels: tables, columns, rows (using row-level security/Virtual Private Database), and views.
  - **Data redaction/masking:** Specific sensitive data elements can be dynamically hidden or obfuscated for users who do not require full access.
- **Auditing:** All database activity can be recorded. This involves query, DML, and DDL auditing to track who accessed or changed the data.
- **Internal threat prevention:** Controls including SQL injection-attack prevention (such as EDB SQL/Protect) can be implemented to ensure secure configuration of views using security barriers.
- **Encryption at rest:** Stored data can be protected using mechanisms such as Transparent Data Encryption (TDE), column-level encryption, or file system/data file encryption, often managed through a key management system. Storage-layer encryption and application-layer encryption can provide additional layers of protection.

> **Important:**
> EDB software is just one component of a complete security solution. The first three layers of the database security model are outside the scope of this white paper. The purpose of this paper is to describe the security practices applicable to database servers.

Database and row-level security

Authentication

Authentication is the process of verifying that the user is who they claim to be. This is typically managed through the PostgreSQL host-based authentication (HBA) file, `pg_hba.conf`, which defines access rules and authentication methods.

Key guidelines for authentication include:

- **Authentication methods:** PostgreSQL supports multiple authentication methods; those most frequently used are discussed below:
  - **Password authentication:** The most secure password hashing mechanism, SCRAM-SHA-256, can be used for storing and sending the password. MD5 support has been deprecated, beginning with PostgreSQL v18, and will be removed in a future release. If older client libraries that don't support SCRAM-SHA-256 remain in use, it is recommended to upgrade them as soon as possible. Never use clear text password authentication.
  - **LDAP:** Authentication is performed by the PostgreSQL server performing a search in the LDAP directory, using the username and password provided by the client. The user must already exist in the database before LDAP can be used for authentication. To synchronize user accounts between the LDAP directory and the database instance, tools such as EDB LDAP Sync can be utilized.
  - **Kerberos/GSSAPI:** This common security implementation provides automatic authentication (single sign-on) for systems that support it. To use GSSAPI for authentication, the client principal must be associated with a PostgreSQL database username by means of the `pg_ident.conf` configuration file.
  - **Certificate authentication:** With SSL client certificates performing authentication, the server will require that the client provide a valid, trusted certificate. The CN (Common Name) attribute of the certificate must match the requested database user name or be mapped to one using the `pg_ident.conf` configuration file.
  - **Trust:** This authentication method should be avoided except in rare cases, when there is adequate operating system–level protection on connections to the server. Only local connections using trust authentication should be allowed.
- **Password policies:** When using password authentication, it is best practice to ensure that appropriate password complexity and lifetime rules are followed. EDB Postgres Advanced Server (EPAS) offers password profiles to enforce strong password management. These profiles can define:
  - Rules for password complexity using a verify function.
  - The maximum number of failed login attempts before an account is locked out.
  - Password expiration (`PASSWORD_LIFE_TIME`) and grace periods (`PASSWORD_GRACE_TIME`).
  - Rules to limit password reuse (`PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX`).
  - Passwords should never be stored in clear text. Using the `PGPASSWORD` environment variable should be avoided, as it stores credentials in clear text, making them visible in login scripts, the process list, or command history. When using the .pgpass file to store database credentials, ensure that the file is only accessible by authorized accounts, using appropriate permissions at the operating system level.

Authorization

Authorization verifies what the authenticated user is allowed to access or what actions they can perform. This must adhere to the principle of least privilege, ensuring that users and processes are granted the bare minimum rights needed for their jobs.

Key authorization guidelines include:

- **Role-based access control (RBAC):** Dedicated roles can be created to reflect application duties and separate responsibilities (e.g., read-only, read-write, admin). Privileges should be granted at the granular level (schema/table/column); it is best practice to avoid granting broad rights to the PUBLIC pseudo-role.
- **Controlling privileges:** This reduces the risk of inadvertent data loss or disclosure and helps mitigate threats posed by insiders.
  - Excessive DML privileges (`INSERT`, `UPDATE`, `DELETE`) can be revoked to prevent unauthorized changes.
  - Superuser attributes (e.g., `SUPERUSER`, `CREATEDB`, `CREATEROLE`) can be restricted to only those roles that absolutely require them.
  - It is best to use dedicated, restricted roles for schema ownership to minimize the chance of attackers modifying or dropping objects.
  - Creating separate roles for data definition language (DDL) and data manipulation language (DML) privileges can also be considered.
- **Row-level security (RLS):** RLS restricts access to individual rows in a table on a per-user basis, regardless of the overall table privileges.
  - RLS policies can be applied to specific commands (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) and roles.
  - Superusers and roles with the `BYPASSRLS` attribute always bypass RLS policies; therefore, this privilege must be strictly controlled.
  - EDB Postgres Advanced Server extends RLS to offer Virtual Private Database (VPD)–style, column-sensitive policies.
- **Data redaction/masking:** This technique dynamically hides or obfuscates sensitive data elements (such as SSNs or salaries) in query results for users who do not require full access. Redaction is applied conditionally based on user roles or session context. PostgreSQL extensions, such as PostgreSQL Anonymizer, can be used to implement data redaction.
- **Security definer functions:** Functions created with the `SECURITY DEFINER` option execute with the privileges of the owner, not the calling role. This can be useful for providing specific, restricted functionality to non-privileged database users. However, carelessly written functions can inadvertently reduce security.

Auditing

Auditing (or accounting) involves recording all database activity, including usernames, timestamps, and details of actions, which is vital for compliance, security monitoring, and post-event investigation.

Key guidelines for auditing include:

- **Comprehensive logging:** Robust logging features should record user logins, disconnections, successful and failed authentication attempts, and DML/DDL operations. Log files should capture enough detail to identify when, what, and why an event occurred. When using PostgreSQL logs for this purpose, ensure that the log line prefix is configured to provide sufficient information.
- **Advanced auditing:** In many cases, advanced auditing capabilities are required to ensure compliance with industry standards and government regulations. Such capabilities can be implemented using tools such as the PostgreSQL Audit Extension (pgAudit) and the EDB Postgres Advanced Server built-in audit facility. These tools allow fine-grained control over auditable actions and conditions, allowing administrators to maintain the balance between compliance and the cost of processing and storing audit records.

- **Log management and protection:**
  - Audit logs should be kept separate from standard server logs, as they can contain sensitive information.
  - The log file permissions must be restricted to prevent unauthorized access and inadvertent exposure of sensitive data.
  - Audit logs must be securely stored, retained for the required period, and protected from unauthorized modification.
  - Logs can be integrated with security information and event management (SIEM) systems for centralized monitoring and alerting.

Data lifecycle security and business continuity

A comprehensive security strategy must extend beyond protecting a live database to encompass the entire data lifecycle, from creation and backup to final, secure disposal. Backup repositories and high availability replicas expand the attack surface and must be secured with the same rigor as the primary database.

Secure backup and recovery

A resilient backup strategy is the last line of defense against ransomware and data corruption. This strategy must be driven by business requirements, specifically the recovery point objective (RPO), which defines the maximum acceptable data loss, and the recovery time objective (RTO), which defines the maximum acceptable downtime. The industry best practice is to combine periodic physical backups with continuous write-ahead logging (WAL) archiving to enable point-in-time recovery (PITR). This should be managed with mature, dedicated tools such as pgBackRest or Barman, not custom scripts. All backup artifacts are high-value targets and must be encrypted both in transit and at rest; stored in an isolated security domain; and subject to regular, automated restore testing to validate their integrity.

Secure data deletion

Standard SQL commands such as `DELETE` or `TRUNCATE` do not securely remove data from physical storage; they typically only mark space as reusable, leaving the data recoverable by forensic tools. This creates significant compliance risk under regulations including GDPR's "right to be forgotten." To ensure that data is permanently unrecoverable, organizations must follow a formal media sanitization framework such as NIST SP 800-88. For specific records within a live database, the most practical and effective technique is cryptographic erasure (or crypto-shredding). This involves encrypting sensitive data with a dedicated key and then securely deleting only the key, rendering the underlying ciphertext permanently unreadable.

The basic step-by-step method for cryptographic erasure:

1. Encrypt sensitive data before storing it in the database, using a unique encryption key (per record, table, or dataset).
2. Store the encrypted data in place of plain sensitive values, together with a reference to the corresponding key.
3. When deletion is requested:
   - Securely delete the encryption key that protects the targeted data (e.g., remove from key vault, HSM, or securely overwrite).
4. As a result, the encrypted data remains in the database—but without the key, it is computationally infeasible to recover the original values, even with access to the database.

Supply chain security

EDB helps customers maintain a secure software supply chain by providing security guidance and documentation, proactively managing vulnerabilities, and embracing transparency regarding the software components it maintains and supports. The key aspects of this work include the following:

1. In providing secure configuration guidelines (STIGs and CIS Benchmarks), EDB helps customers establish a secure configuration posture by offering authoritative guidance for their database products.

   EDB developed the EDB Postgres Advanced Server Secure Technology Implementation Guidelines, which have been certified by the U.S. Department of Defense. This guide provides customers with the guidance necessary to set up EDB Postgres securely.

   EDB also publishes best practices guides for secure PostgreSQL EPAS deployment and operation, which organizations can use to inform the development of their own security policies.

2. EDB is committed to providing timely security patches and updates for PostgreSQL, EDB Postgres Advanced Server, and other software supported by EDB:

   - **Proactive management:** EDB actively monitors for security vulnerabilities and releases patches to address them.
   - **Support for updates:** EDB's support subscriptions provide timely notifications of security updates and appropriate patches for Postgres.
   - **Open source advantage:** Open source projects, including PostgreSQL, often fix vulnerabilities and release patches much faster than commercial vendors, sometimes within a day or two for high-severity issues.

3. EDB promotes transparency and provides the necessary tools for customers to manage their components. EDB encourages tracking components through a software bill of materials (SBOM) and subscribing to EDB security advisories. Organizations should review SBOMs to manage risks associated with components.

4. EDB recommends operational hardening practices that contribute to supply chain security by restricting the environment in which the software runs:

   - **Restricting extensions:** EDB advises organizations to avoid unvetted extensions and keep the list of shared preload libraries minimal, to reduce the attack surface.
   - **Authorized repositories:** When installing software packages, it is imperative to source them only from valid and authorized repositories to prevent implementing untested, defective, or malicious software.

5. EDB has achieved U.S. Government certifications and authorizations to operate on sensitive networks (including NIPRNet, SIPRNet, and JWICS). This indicates that the software has met stringent quality and security standards required for production environments.

Auditing and compliance

Implementation of robust database auditing is critical for achieving and maintaining compliance with standards such as PCI DSS and GLBA and regulations such as GDPR. The audit trail should provide the verifiable evidence required by regulators and internal auditors to demonstrate adherence to the principle of need-to-know access, detect unauthorized lateral movement, and ensure complete accountability for every interaction with protected or sensitive data. While the PostgreSQL server log can record connect and disconnect attempts, as well as the execution of SQL commands, it is not granular enough for use as an audit trail.

Advanced auditing

For PostgreSQL databases, the extension `pgAudit` provides a structured, highly configurable method for tracking specific activities within the database. It is designed to capture every action an authenticated user takes, including detailed information about the executed SQL statements, parameters used, and rows affected. It allows administrators to configure auditing based on command type (e.g., `READ`, `WRITE`, `DDL`), object (e.g., specific tables or functions), or role, reducing log noise while ensuring accountability.

EDB Postgres Advanced Server offers an enhanced auditing feature, EDB Audit Logging, that can track DML and DDL activity, connect/disconnect attempts, specific errors, and export audit logs in formats suitable for integration with log analysis applications: CSV, XML, or JSON. Audit logs can be automatically archived according to the defined schedule.

Session tags in EDB Audit Logging allow linking database activities to middle-tier application data (such as application names, user IDs, or IP addresses) that are distinct from the database users these applications connect as. Session tagging, along with the command tagging, allows filtering of audit records by various criteria suitable for advanced analysis of events.

Passwords in audit records, such as those potentially present in `CREATE ROLE` and similar commands, are automatically obfuscated by EDB Audit Logging.

Regulatory and standard compliance

Auditing is just one of the tools necessary for implementing a robust set of controls across the entire IT infrastructure. Organizations can be subject to a wide range of laws, regulations, and industry standards that are applicable to different aspects of their business. In addition to the PCI DSS, GLBA, and GDPR, as mentioned earlier, an organization may be subject to the Sarbanes-Oxley Act (SOX) and Service Organization Control 1 (SOC 1) report requirements, which can focus on meeting the Internal Controls over Financial Reporting (ICFR) rules, or to the demands of the broader information security management standard ISO/IEC 27001. Essential aspects of compliance with all of these include:

- **Strict authentication and identity management:** Along with granular **authorization** controls, this is the other critical component of the AAA model. Regulations increasingly mandate multifactor authentication (MFA) for anyone with privileged access to sensitive systems (including the database and its log files), as well as centralizing identity management (e.g., using LDAP or Active Directory) to ensure consistent user termination and policy enforcement.
- **Data encryption and protection, both at rest and in transit:** Disk-level or Transparent Data Encryption and the use of SSL/TLS with strong ciphers for database connections are almost universally mandatory. For cardholder data (PCI DSS), certain information—such as the full magnetic stripe data—cannot be stored at all, and primary account numbers (PANs) must be masked or tokenized.
- **Vulnerability management and patching:** Systems must be continuously monitored and maintained to eliminate known weaknesses. This involves a regular schedule of vulnerability scans and penetration testing. Crucially, the organization must maintain a rigorous patch management process, ensuring that the PostgreSQL server, operating system, and all associated software and extensions (such as pgaudit) are kept up to date to protect against exploits.
- **Secure development and change management:** Changes to systems and applications must be controlled and tested. All custom software (including application code and database functions/stored procedures) that interacts with sensitive information must be developed using secure coding practices. Furthermore, a strict change management process must be in place, ensuring that every infrastructure and database change—from a schema update to a firewall rule adjustment—is reviewed, tested, documented, and approved before being implemented in the production environment. This prevents accidental introduction of security flaws.
- **Confidentiality and auditability:** Access to data must be logged and monitored, especially for privileged users. Enable controls to ensure that sensitive data, such as that used in financial reporting, is shared only with authorized parties (e.g., auditors, management) and that all access is traceable. Consider implementing the just-in-time (JIT) access model by granting access to the sensitive data only for a limited period of time, and revoke it immediately after the task requiring access is complete. This relies on tight integration between database authorization management and centralized user authentication systems such as Active Directory with Kerberos or other single sign-on (SSO) systems.

- **Data retention and disposal:** Regulations often dictate how long data must be kept for audit and reporting purposes. Retained data, including backup copies, must be protected from accidental or malicious deletion and modification, for example, by utilizing write once read many (WORM) storage. Once the mandated retention period ends, the data must be securely deleted and the corresponding storage media sanitized using well-documented procedures, to prevent unauthorized recovery.

Appendix

Mastering postgresql.conf: A security-first approach

Table 1: Critical postgresql.conf Security Parameters

Parameter	Recommended Value	Default Value	Rationale and Risk of Misconfiguration
`listen_addresses`	'localhost, <trusted_ip>'	'*' or 'localhost'	Restricts connections to only necessary interfaces, preventing exposure on public or untrusted networks. A value of '*' is a critical vulnerability.
`ssl`	on	off	Enforces encryption for all connections, protecting data in transit from eavesdropping and man-in-the-middle attacks.
`password_encryption`	scram-sha-256	md5	Uses a modern, salted, and computationally expensive hashing algorithm, preventing offline password cracking. md5 is deprecated and considered insecure.
`log_connections`	on	off	Creates an audit trail of all successful connection attempts, essential for forensic analysis and intrusion detection.
`log_disconnections`	on	off	Logs session duration, providing context for activity analysis and helping to identify unusually long or short sessions.
`log_line_prefix`	'%m [%p] %u@%d'	'%m [%p]'	Adds critical context (user, database) to every log entry, enabling effective parsing, alerting, and correlation in a SIEM.
`log_statement`	'ddl'	'none'	Logs all data definition language (DDL) statements (e.g., `CREATE`, `ALTER`, `DROP`), which is critical for change control auditing. 'all' may be too verbose for production.

Architecting pg_hba.conf for zero trust

Table 2: pg_hba.conf Rule Analysis: From Insecure to Zero Trust

Scenario	Insecure Rule (and Why It's Bad)	Secure Rule (and Why It's Good)
Local admin access	`local all postgres trust` – Allows anyone with OS access as the postgres user to connect as a database superuser without a password. A compromised OS account leads to an immediate, total database compromise.	`local all postgres peer` – Requires the connecting OS user to be postgres. While better, a password is still recommended. local all postgres scram-sha-256 is the most secure option for local superuser access.
Application access	`host all all 0.0.0.0/0 md5` – Allows any user to connect to any database from anywhere on the internet using a weak password hash. This is an open invitation for brute-force attacks and is a critical vulnerability.	`hostssl app_db app_user 10.5.0.0/24 scram-sha-256` – Requires SSL, for a specific database and user, from a specific private subnet, using a strong password hash. This rule enforces the principle of least privilege.
Replication	`host replication replicator 10.10.0.0/16 trust` – Allows any host on a large internal network to connect as a replication user without a password. An attacker on this network can impersonate a standby and exfiltrate the entire database.	`hostssl replication replicator 10.10.2.5/32 scram-sha-256` – Requires SSL and a strong password for the replication user, and only permits the connection from the specific IP address of the known standby server.
Monitoring	`host all monitor 127.0.0.1/32 password` – Uses the clear text password method, which is insecure. While only from localhost, it still exposes credentials unnecessarily if traffic were ever captured.	`host all monitor 127.0.0.1/32 scram-sha-256` – Enforces the use of a strong, modern authentication mechanism even for local monitoring connections, maintaining a consistent security standard.

About the author

Nick Ivanov
Solutions Architect, EDB
Connect on LinkedIn

Nick Ivanov is a seasoned solutions architect at EDB. Since April 2022 he has brought extensive expertise in database architecture and analytics from a notable tenure at IBM from May 2015 to March 2022. He holds a Dipl.-Ing. degree in computing systems and networks from Bauman Moscow State Technical University.

About EDB Postgres AI

EDB Postgres AI is the first open, enterprise-grade sovereign data and AI platform, with a secure, compliant, and fully scalable environment, on premises and across clouds. Supported by a global partner network, EDB Postgres AI unifies transactional, analytical, and AI workloads, enabling organizations to operationalize their data and LLMs where, when, and how they need them.

EDB POSTGRES AI