PostgreSQL Security & Access Control

Hardening, Authentication, Authorization, and Compliance in Production Environments

Preface

Every database tells a story. It holds the financial records of a growing business, the medical histories of patients, the personal details of millions of users who trusted an organization with their data. **PostgreSQL**—one of the most powerful and widely adopted open-source relational databases in the world—sits at the heart of countless such stories. And yet, too often, the security of a PostgreSQL deployment is treated as an afterthought: a checkbox to be ticked after the schemas are designed, the queries are tuned, and the application is live.

This book exists because that approach is no longer acceptable—if it ever was.

Why This Book

The threat landscape facing PostgreSQL deployments has grown more sophisticated and relentless with each passing year. Ransomware campaigns target exposed database ports. Misconfigured `pg_hba.conf` files silently grant access to the wrong networks. Overprivileged roles accumulate like technical debt. SQL injection remains stubbornly prevalent. Compliance frameworks such as GDPR, HIPAA, SOC 2, and PCI DSS demand not just good intentions but *demonstrable* controls.

PostgreSQL Security & Access Control was written to bridge the gap between knowing that PostgreSQL security matters and knowing *exactly what to do about it*. Whether you are a database administrator hardening a production cluster, a developer building secure multi-tenant applications, or an architect designing systems that must satisfy auditors and regulators, this book provides the practical, PostgreSQL-specific guidance you need.

What You Will Find Inside

This book is organized into sixteen chapters and five appendices, structured to take you on a deliberate journey from foundational concepts to advanced, production-grade security practices.

We begin by establishing *why* database security is critical and examining **PostgreSQL's security architecture** from the ground up. From there, we dive deep into the mechanisms that govern who can connect and how—exploring `pg_hba.conf` configuration, authentication methods ranging from password-based schemes to certificate and GSSAPI authentication, and the PostgreSQL role and privilege system that controls what authenticated users can actually do.

The middle chapters address **encryption in transit and at rest**, equipping you to enable SSL/TLS for PostgreSQL connections and protect sensitive data stored on disk. We then turn to **monitoring and defense**: configuring PostgreSQL's logging infrastructure for security auditing, detecting intrusions, mitigating threats, and preventing SQL injection at both the database and application layers.

Later chapters tackle the architectural challenges of **secure multi-tenant design** within PostgreSQL, **hardening postgresql.conf and related configuration**, securing replication and backup pipelines, and meeting the demands of **compliance and governance** frameworks. The final chapter invites you to think beyond day-to-day administration and evolve into a *security-focused architect*—someone who embeds defense into every layer of a PostgreSQL deployment.

The appendices provide ready-to-use resources: secure `pg_hba.conf` templates, a role and privilege reference table, an SSL configuration checklist, a comprehensive security audit checklist, and a PostgreSQL hardening worksheet you can adapt to your own environments.

How to Use This Book

You can read this book cover to cover for a complete education in PostgreSQL security, or you can treat individual chapters as targeted references when facing specific challenges. Each chapter is designed to be self-contained enough to be useful on its own, while contributing to a coherent, cumulative understanding when read in sequence.

Acknowledgments

This book would not exist without the extraordinary PostgreSQL community—the developers, contributors, and documentation authors who have built and refined a database engine that takes security seriously at its core. I am also deeply grateful to the security researchers, DBAs, and architects whose real-world experiences and hard-won lessons informed every chapter. Special thanks to the technical reviewers who challenged assumptions, caught errors, and made this a stronger, more trustworthy resource.

A Final Word Before We Begin

Security is not a feature you install. It is a discipline you practice—continuously, deliberately, and with humility. PostgreSQL gives you remarkably powerful tools to protect your data. This book will show you how to use them.

Let's get started.

Thomas Ellison

Table of Contents

Chapter 1: Why Database Security Is Critical

The moment a PostgreSQL database goes live in a production environment, it becomes a target. This is not an exaggeration or a scare tactic designed to sell security products. It is a plain, observable truth that anyone who has managed a publicly accessible database server can confirm within hours of deployment. Automated scanners sweep the internet continuously, probing default ports, testing common credentials, and looking for misconfigured services. PostgreSQL, being one of the most widely adopted open-source relational database management systems in the world, sits squarely in the crosshairs of these automated attacks and, far more dangerously, of deliberate, targeted intrusions carried out by sophisticated adversaries.

Understanding why database security is critical requires more than a surface-level acknowledgment that "security matters." It demands a deep appreciation of what is at stake, how breaches occur, what PostgreSQL offers natively to defend against threats, and what responsibilities fall on the shoulders of database administrators, developers, and architects who design and maintain these systems. This chapter lays that foundation. Before we configure a single parameter, write a single policy, or modify a single authentication rule, we must internalize the gravity of the problem and the landscape in which we operate.

The Value of What PostgreSQL Protects

Every database exists to store, organize, and serve data. That data, in a production environment, almost always has tangible value. Consider the types of information commonly housed in PostgreSQL databases across industries. Financial institutions store transaction records, account balances, and personal identification numbers. Healthcare organizations maintain patient records that include diagnoses, medications, and insurance details. E-commerce platforms hold credit card numbers, shipping addresses, and purchase histories. Government agencies manage citizen records, tax filings, and classified operational data. Even a modest startup running a SaaS application likely stores email addresses, hashed passwords, API keys, and behavioral analytics that represent both intellectual property and personal data subject to legal protection.

The value of this data is not abstract. It translates directly into financial loss when compromised. The IBM Cost of a Data Breach Report, published annually, consistently places the average cost of a data breach in the millions of dollars. These costs include incident response, forensic investigation, legal fees, regulatory fines, customer notification, credit monitoring services, lost business due to reputational damage, and the long-term erosion of customer trust. For organizations operating under regulatory frameworks such as the General Data Protection Regulation (GDPR), the Health Insurance Portability and Accountability Act (HIPAA), the Payment Card Industry Data Security Standard (PCI DSS), or the Sarbanes-Oxley Act (SOX), the penalties for failing to protect data can be severe and, in some jurisdictions, existential.

PostgreSQL, as the system entrusted with this data, is not merely a technical component. It is the custodian of an organization's most sensitive assets. When we talk about PostgreSQL security, we are talking about the protection of those assets at the layer where they are most concentrated and most vulnerable.

The Threat Landscape for PostgreSQL Deployments

To protect a PostgreSQL database effectively, one must understand the threats it faces. These threats can be categorized broadly into external attacks, internal threats, and accidental exposure, each of which manifests in specific ways within PostgreSQL environments.

External attacks include brute-force authentication attempts against the PostgreSQL listener, SQL injection through application layers that interact with the database, exploitation of known vulnerabilities in unpatched PostgreSQL versions, and network-level interception of unencrypted database traffic. An attacker who gains access to a PostgreSQL superuser account effectively owns the entire database cluster, including every database, every schema, every table, and every piece of data within it. They can read sensitive information, modify records, drop entire databases, or install backdoors using PostgreSQL's extensibility features such as custom functions written in procedural languages.

Internal threats are statistically more common and often more damaging than external attacks. A disgruntled employee with legitimate database credentials can exfiltrate data gradually over weeks or months without triggering obvious alarms. A developer with overly broad permissions might accidentally or intentionally access production data they have no business seeing. A contractor given temporary access that is never revoked becomes a persistent vulnerability. PostgreSQL's role-based access control system provides the mechanisms to mitigate these risks, but only if those mechanisms are deliberately configured and continuously maintained.

Accidental exposure represents a category of threat that is entirely preventable yet remarkably common. Databases deployed with default configurations, superuser accounts left with well-known passwords, PostgreSQL instances bound to all network interfaces without firewall restrictions, unencrypted connections transmitting credentials and query results in plaintext, backup files stored without encryption on shared storage, and verbose error messages that reveal schema details to application users are all examples of accidental exposure. Each of these represents a door left open, not by an attacker, but by the people responsible for securing the system.

The following table summarizes common threat vectors specific to PostgreSQL environments and their potential impact:

Threat Vector	**Description**	**Potential Impact**	**PostgreSQL Relevance**
Brute-force authentication	Automated tools attempt thousands of username and password combinations against the PostgreSQL port	Unauthorized superuser access leading to full data compromise	PostgreSQL listens on port 5432 by default and accepts password-based authentication unless configured otherwise
SQL injection	Malicious SQL statements injected through application input fields reach the database engine	Data exfiltration, modification, or destruction; potential command execution via procedural languages	PostgreSQL executes injected SQL with the privileges of the connecting application role
Unpatched vulnerabilities	Known security flaws in specific PostgreSQL versions are exploited before patches are applied	Remote code execution, privilege escalation, denial of service	The PostgreSQL Global Development Group publishes security updates regularly; unpatched systems remain exposed
Unencrypted connections	Database traffic transmitted without TLS/SSL encryption is intercepted on the network	Credential theft, query result interception, session hijacking	PostgreSQL supports SSL natively but does not enforce it by default

Excessive privileges	Users or application roles granted more permissions than required for their function	Unauthorized data access, accidental data modification or deletion	PostgreSQL's default behavior for newly created roles varies; superuser privileges are particularly dangerous
Unrestricted network access	PostgreSQL bound to all interfaces with permissive pg_hba.conf rules	Any host on the network or internet can attempt connections	The listen_addresses parameter and pg_hba.conf file control network accessibility
Backup exposure	Database backups stored without encryption or access controls	Complete data compromise from a single stolen backup file	pg_dump and pg_basebackup produce unencrypted output by default
Insider data theft	Authorized users with legitimate access exfiltrate data beyond their role requirements	Regulatory violations, competitive intelligence loss, privacy breaches	Without granular role configuration and auditing, insider activity is difficult to detect

PostgreSQL's Native Security Architecture

One of the reasons PostgreSQL is trusted in security-conscious environments is that it provides a comprehensive, layered security architecture out of the box. Understanding this architecture at a high level is essential before diving into the detailed configuration chapters that follow.

The first layer is network-level access control. PostgreSQL uses a configuration file called `pg_hba.conf` (host-based authentication) to determine which hosts are allowed to connect, which databases they can access, which users they can authenticate as, and what authentication method is required. This file acts as a gatekeeper before any authentication credentials are even evaluated. A properly configured `pg_hba.conf` file can restrict access to specific IP addresses or subnets, require SSL for all connections, and enforce different authentication methods for local versus remote connections.

The second layer is authentication. PostgreSQL supports multiple authentication methods, including password-based methods (md5 and scram-sha-256), certificate-based authentication, LDAP, RADIUS, GSSAPI (Kerberos), PAM, and peer authentication for local connections. The choice of authentication method has profound implications for security. The older md5 method, for example, is vulnerable to replay attacks and should be replaced with scram-sha-256 in all new deployments. Certificate-based authentication eliminates passwords entirely and is considered the gold standard for machine-to-machine database connections.

The third layer is authorization, implemented through PostgreSQL's role-based access control (RBAC) system. Every connection to PostgreSQL is made as a specific role, and that role's privileges determine what actions are permitted. Privileges can be granted at the database level, schema level, table level, column level, and even at the row level using Row-Level Security (RLS) policies. This granularity allows administrators to implement the principle of least privilege with precision, ensuring that each user or application can access only the specific data and operations required for their function.

The fourth layer is encryption. PostgreSQL supports SSL/TLS encryption for data in transit, protecting credentials and query results from network interception. For data at rest, PostgreSQL does not provide native transparent data encryption (TDE) in the community edition as of the current major releases, but this can be achieved through filesystem-level encryption, third-party extensions, or enterprise distributions that add this capability.

The fifth layer is auditing and monitoring. While PostgreSQL's default logging captures certain events, comprehensive audit logging requires additional configuration or the use of extensions such as `pgaudit`, which provides detailed session and object audit logging that satisfies regulatory requirements. Understanding who accessed what data, when, and from where is not merely a compliance checkbox; it is a fundamental security capability that enables detection of unauthorized activity and supports forensic investigation after incidents.

Consider the following basic example that illustrates how these layers work together. When an application server attempts to connect to a PostgreSQL database:

```
Step 1: PostgreSQL checks pg_hba.conf to determine if the source
IP address
        is permitted to connect to the requested database as the
requested user.

Step 2: If the connection is permitted, PostgreSQL enforces the
authentication
        method specified in pg_hba.conf (e.g., scram-sha-256).

Step 3: The client provides credentials, which PostgreSQL
validates against
        its internal catalog (pg_authid) or an external
authentication provider.

Step 4: Upon successful authentication, the session operates
under the
        privileges assigned to the authenticated role.

Step 5: Every SQL statement executed is checked against the
role's privileges
        on the target objects. Row-Level Security policies, if
defined, further
        filter the visible and modifiable rows.

Step 6: If audit logging is configured, the statement, its
parameters, the
```

```
        executing role, and the timestamp are recorded in the
audit log.
```

This layered approach means that a failure at any single layer does not necessarily result in a complete compromise. If an attacker bypasses network controls, they still face authentication. If they obtain credentials, they are still constrained by the role's privileges. If they find a privilege escalation path, audit logging can detect the anomalous activity. Defense in depth is not just a theoretical principle in PostgreSQL; it is built into the system's architecture.

The Cost of Neglecting PostgreSQL Security

The consequences of failing to secure a PostgreSQL database extend far beyond the immediate technical impact of a breach. Organizations that experience data breaches face a cascade of consequences that unfold over months and years.

Immediate consequences include the cost of incident response. When a breach is detected, organizations must engage forensic specialists to determine the scope of the compromise, identify the attack vector, and assess what data was accessed or exfiltrated. PostgreSQL's transaction logs, if properly configured, can assist in this investigation, but if logging was minimal or logs were stored on the compromised system, forensic analysis becomes significantly more difficult and expensive.

Regulatory consequences follow. Under GDPR, organizations that fail to protect personal data of European residents face fines of up to 4% of annual global turnover or 20 million euros, whichever is greater. HIPAA violations can result in fines ranging from $100 to $50,000 per violation, with annual maximums of $1.5 million per violation category. PCI DSS non-compliance can result in fines from payment card brands ranging from $5,000 to $100,000 per month. These are not theoretical penalties; they are actively enforced.

Reputational consequences are perhaps the most enduring. Customers who learn that their personal data was compromised due to a preventable security failure, such as a PostgreSQL superuser account with a weak password or an unencrypted database connection, lose trust in the organization. Rebuilding that trust takes years and may never fully succeed.

Legal consequences include class-action lawsuits from affected individuals, contractual penalties from business partners whose data was exposed, and potential criminal liability for executives who were aware of security deficiencies and failed to act.

The following table provides a reference for common regulatory frameworks and their specific requirements that relate to PostgreSQL database security:

Regulatory Framework	**Key Database Security Requirements**	**PostgreSQL Capabilities**
GDPR (General Data Protection Regulation)	Data minimization, access controls, encryption, breach notification, right to erasure, audit trails	Role-based access control, SSL/TLS encryption, pgaudit extension, column-level privileges, Row-Level Security
HIPAA (Health Insurance Portability and Accountability Act)	Access controls, audit controls, integrity controls, transmission security, encryption of PHI	Authentication methods, pg_hba.conf network controls, SSL encryption, pgaudit, checksums for data integrity
PCI DSS (Payment Card Industry Data Security Standard)	Restrict access on a need-to-know basis, unique IDs for each user, encrypt transmission of cardholder data, track and monitor all access	Role-based access control, individual role accounts, SSL/TLS, comprehensive logging and pgaudit

SOX (Sarbanes-Oxley Act)	Internal controls over financial reporting, audit trails, access restrictions to financial data	Schema-level and table-level privileges, pgaudit for audit trails, role separation
SOC 2 (Service Organization Control)	Logical access controls, system monitoring, encryption, change management	pg_hba.conf, role management, SSL, logging configuration, pgaudit

The Mindset Required for PostgreSQL Security

Securing a PostgreSQL database is not a one-time activity. It is not something that is "done" during initial setup and then forgotten. Security is a continuous process that requires ongoing attention, regular review, and adaptation to evolving threats. The chapters that follow in this book will provide detailed, practical guidance on every aspect of PostgreSQL security, from hardening the server configuration to implementing fine-grained access controls, from configuring encryption to establishing audit logging that satisfies regulatory requirements.

However, all of that technical knowledge is only effective when applied with the right mindset. That mindset includes several key principles.

The principle of least privilege dictates that every role in PostgreSQL should have the minimum permissions necessary to perform its intended function. Application roles should not be superusers. Read-only reporting roles should not have write access. Administrative roles should be used only for administrative tasks, not for routine application connections.

The principle of defense in depth means that security should not depend on any single control. Network restrictions, strong authentication, granular authorization, encryption, and audit logging should all be implemented together, each providing protection even if another layer fails.

The principle of secure defaults means that every new PostgreSQL deployment should start from a hardened baseline configuration rather than relying on the default settings, which are designed for ease of initial setup rather than production security.

The principle of continuous monitoring means that security events should be logged, aggregated, analyzed, and alerted upon in real time. A breach that is detected in minutes causes far less damage than one that persists undetected for months.

As we proceed through this book, each chapter will build upon these principles, translating them into specific PostgreSQL configurations, commands, policies, and practices. The goal is not merely to understand PostgreSQL security in theory but to implement it in practice, creating production environments that protect the data entrusted to them against the full spectrum of threats they face.

The stakes are real. The threats are active. The tools that PostgreSQL provides are powerful. What remains is the knowledge and discipline to use them effectively. That journey begins now.

Chapter 2: PostgreSQL Security Architecture

Understanding the security architecture of PostgreSQL is not merely an academic exercise. It is a foundational requirement for every database administrator, developer, and security engineer who is entrusted with protecting organizational data. PostgreSQL, unlike many commercial database systems, was designed from its earliest days with a layered security model that gives administrators granular control over who can connect, how they authenticate, and what they are permitted to do once inside the system. This chapter takes you on a thorough journey through the internal security architecture of PostgreSQL, examining each layer in detail, explaining how the components interact with one another, and providing practical examples that you can apply directly in production environments.

When we talk about the "security architecture" of PostgreSQL, we are referring to the entire ecosystem of mechanisms, configurations, processes, and internal structures that collectively determine how the database system protects itself and the data it stores. This is not a single feature or a single configuration file. It is a carefully orchestrated set of layers, each serving a distinct purpose, and each depending on the others to create a comprehensive defense. To truly harden a PostgreSQL installation, you must understand every one of these layers, how they fit together, and where the boundaries of each layer begin and end.

The Layered Security Model

PostgreSQL implements what is best described as a defense-in-depth strategy. Rather than relying on a single point of security enforcement, the system applies multiple independent layers of protection. Each layer acts as a checkpoint, and a request must successfully pass through every layer before it can access or modify data. If any single layer denies access, the request is stopped regardless of what the other layers might allow.

The outermost layer is the network and operating system layer. Before PostgreSQL even becomes aware of a connection attempt, the operating system's firewall rules, network segmentation, and TCP/IP configuration determine whether a packet can reach the PostgreSQL server process. This is not technically part of PostgreSQL itself, but it is an inseparable component of the overall security architecture because PostgreSQL cannot protect against connections that it never sees, and conversely, a misconfigured network layer can expose the database to threats that PostgreSQL's internal mechanisms were never designed to handle alone.

The next layer inward is the connection and authentication layer. This is where PostgreSQL's `pg_hba.conf` file plays its critical role. When a client connection arrives at the PostgreSQL server, the system consults this file to determine whether the connection should be allowed and, if so, what authentication method should be used. This layer answers two fundamental questions: Is this client allowed to connect at all, and how must this client prove its identity?

Beyond authentication lies the authorization layer, which is the realm of roles, privileges, and permissions. Once a client has successfully authenticated, PostgreSQL must determine what that client is allowed to do. This is where the role-based access control system comes into play, governing access to databases, schemas, tables, columns, functions, and virtually every other object within the system.

Finally, at the innermost layer, PostgreSQL provides row-level security policies, column-level permissions, and security-defining functions that allow administrators to implement fine-grained access control at the data level itself. This innermost layer ensures that even users who have general access to a table can be restricted to seeing or modifying only specific rows or columns.

The following table summarizes these layers and their primary responsibilities:

Layer	**Primary Mechanism**	**Configuration Location**	**Purpose**
Network and OS	Firewall rules, TCP wrappers, network segmentation	iptables, firewalld, OS configuration	Prevent unauthorized network access to the server
Connection and Authentication	Host-based access control, authentication methods	pg_hba.conf, postgresql.conf	Control who can connect and how they prove identity
Authorization	Roles, privileges, GRANT and REVOKE statements	SQL commands, system catalogs	Determine what authenticated users can do
Data-Level Security	Row-level security, column privileges, security definer functions	SQL policies, GRANT on columns	Restrict access to specific rows and columns

Understanding this layered model is essential because security failures almost always occur when one or more layers are misconfigured or missing entirely. A common mistake, for example, is to focus exclusively on strong passwords while leaving the network layer wide open, or to carefully configure `pg_hba.conf` while neglecting to set appropriate object-level privileges.

The PostgreSQL Server Process and Security Context

To appreciate how security enforcement actually works at runtime, you need to understand the PostgreSQL process architecture. When the PostgreSQL server starts, it launches a primary process traditionally called the postmaster. This process listens for incoming connections on a configured port, which by default is 5432. When a client connection arrives, the postmaster forks a new backend process dedicated to serving that specific client.

Each backend process runs under the security context of the operating system user that owns the PostgreSQL installation, which is typically a user named `postgres`. However, within the database system itself, each backend process operates under the identity of the database role that the client authenticated as. This distinction is important: at the OS level, all backend processes look identical, but at the database level, each one carries the identity and privileges of a specific role.

You can observe this behavior by querying the `pg_stat_activity` system view, which shows all active backend processes along with the role they are running as:

```sql
SELECT pid, usename, datname, client_addr, backend_start, state
FROM pg_stat_activity
WHERE backend_type = 'client backend';
```

This query returns output similar to the following:

```
  pid   | usename  | datname    | client_addr   | backend_start
| state
-------+----------+------------+---------------
+----------------------------+--------
 12345 | app_user | production | 192.168.1.100 | 2024-01-15
10:23:45.123+00 | active
 12346 | analyst  | reporting  | 192.168.1.101 | 2024-01-15
10:25:12.456+00 | idle
```

```
 12347 | postgres | postgres   | 127.0.0.1     | 2024-01-15
09:00:01.789+00 | active
```

Each row represents a separate backend process, and you can see that each one is associated with a specific user, database, and client address. The security decisions that PostgreSQL makes for each of these processes are entirely independent. The `app_user` process cannot access objects that only `analyst` has privileges on, even though both processes are running as the same OS user.

The postmaster process itself plays a security role during connection establishment. When a new connection arrives, the postmaster consults `pg_hba.conf` before forking the backend process. If the connection is rejected at this stage, no backend process is ever created, which means that rejected connections consume minimal server resources. This design is intentional and helps protect against certain types of denial-of-service attacks.

System Catalogs and Security Metadata

PostgreSQL stores all security-related metadata in system catalogs, which are special tables that exist in every database. These catalogs are the authoritative source of truth for all security decisions. When PostgreSQL needs to determine whether a role has permission to perform an action, it consults these catalogs.

The most important security-related system catalogs are described in the following table:

Catalog Name	Purpose	Key Columns
pg_authid	Stores all roles and their properties	rolname, rolsuper, rolcreaterole, rolcreatedb, rolcanlogin, rolpassword

pg_auth_members	Records role membership relationships	roleid, member, grantor, admin_option
pg_database	Stores database-level properties including access control	datname, datdba, datacl
pg_namespace	Stores schema information including permissions	nspname, nspowner, nspacl
pg_class	Stores table and index information including permissions	relname, relowner, relacl
pg_proc	Stores function and procedure information	proname, proowner, proacl, prosecdef
pg_default_acl	Stores default access privileges	defaclrole, defaclnamespace, defaclobjtype, defaclacl
pg_policy	Stores row-level security policies	polname, polrelid, polcmd, polroles, polqual, polwithcheck

You can query these catalogs directly to understand the current security configuration of your database. For example, to see all roles and their key security properties:

```sql
SELECT rolname,
       rolsuper,
       rolcreaterole,
       rolcreatedb,
       rolcanlogin,
       rolreplication,
       rolbypassrls,
       rolconnlimit,
       rolvaliduntil
FROM pg_authid
ORDER BY rolname;
```

This query reveals critical information about each role. The `rolsuper` column indicates whether a role has superuser privileges, which effectively bypass all access checks. The `rolcanlogin` column distinguishes between roles that can directly authenticate (login roles) and roles that exist solely for privilege grouping (group roles). The `rolbypassrls` column shows whether a role can bypass row-level security policies, which is a powerful and potentially dangerous privilege.

To examine the access control lists on specific tables, you can query `pg_class`:

```
SELECT n.nspname AS schema_name,
       c.relname AS table_name,
       c.relacl AS access_privileges,
       pg_get_userbyid(c.relowner) AS owner
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind = 'r'
  AND n.nspname NOT IN ('pg_catalog', 'information_schema')
ORDER BY n.nspname, c.relname;
```

The `relacl` column contains the access control list in PostgreSQL's compact ACL notation. For example, an entry like `{app_user=arwdDxt/postgres}` means that the role `app_user` has been granted SELECT (r), INSERT (a), UPDATE (w), DELETE (d), TRUNCATE (D), REFERENCES (x), and TRIGGER (t) privileges by the role `postgres`. Understanding this notation is essential for auditing database security.

The following table explains each privilege character in PostgreSQL's ACL notation:

Character	**Privilege**	**Applicable Objects**
r	SELECT (read)	Tables, views, sequences
a	INSERT (append)	Tables, views
w	UPDATE (write)	Tables, views, sequences
d	DELETE	Tables, views
D	TRUNCATE	Tables
x	REFERENCES	Tables
t	TRIGGER	Tables, views

X	EXECUTE	Functions, procedures
U	USAGE	Schemas, sequences, types, domains, foreign data wrappers, foreign servers
C	CREATE	Databases, schemas, tablespaces
c	CONNECT	Databases
T	TEMPORARY	Databases

Note: The asterisk (*) after a privilege character indicates that the role has the ability to grant that privilege to others (WITH GRANT OPTION). For example, `r*` means the role has SELECT privilege and can grant SELECT to other roles.

The Authentication Pipeline in Detail

When a client attempts to connect to PostgreSQL, the authentication process follows a precise sequence of steps. Understanding this sequence is critical for troubleshooting connection issues and for ensuring that your authentication configuration is secure.

First, the client establishes a TCP connection to the server (or a Unix domain socket connection on the local machine). The postmaster accepts this connection and reads the startup message from the client, which contains the requested database name, the role name, and various connection parameters.

Second, the postmaster searches `pg_hba.conf` from top to bottom, looking for the first entry that matches the connection type, client address, requested database, and requested role. This is a crucial point: PostgreSQL uses the first matching entry, not the best matching entry or the most specific entry. The order of entries in `pg_hba.conf` matters enormously.

Consider the following example `pg_hba.conf` configuration:

```
# TYPE  DATABASE        USER            ADDRESS
METHOD
local   all             postgres
peer
local   all             all
md5
host    production      app_user        192.168.1.0/24
scram-sha-256
host    all             all             192.168.1.0/24
reject
host    all             all             10.0.0.0/8
scram-sha-256
hostssl all             all             0.0.0.0/0
scram-sha-256
host    all             all             0.0.0.0/0
reject
```

In this configuration, a connection from `192.168.1.100` as `app_user` to the `production` database would match the third line and be required to authenticate using SCRAM-SHA-256. However, a connection from the same address as `analyst` to any database would match the fourth line and be rejected outright, even though the fifth line would allow connections from the broader `10.0.0.0/8` network. This is because the fourth line matches first.

Third, once the matching `pg_hba.conf` entry is found, PostgreSQL applies the specified authentication method. The available methods range from completely trusting the client (the `trust` method, which should never be used in production) to requiring cryptographic proof of identity through methods like `scram-sha-256`, `cert`, or integration with external authentication systems like LDAP, GSSAPI, or RADIUS.

Fourth, if authentication succeeds, PostgreSQL checks whether the authenticated role actually exists in `pg_authid` and whether it has the `LOGIN` privilege. A role without `LOGIN` cannot be used for direct connections even if authentication succeeds.