Work in Progress: Removing the Vulnerable Webapp: Combining JWT and Stored Procedures to Foil SQL Injection

Falcon Darkstar Momot
Aiven Oy
Seattle, Washington, USA
0000-0001-9655-8321

***Abstract*—SQL injection remains an important vulnerability class for multiple reasons, including poor code-data separation in the access protocol, and excessive trust placed in diverse applications to enforce schema rules. Exploitation of SQL injection elevates privilege from the application user context into the application context. This work proposes to eliminate any distinction between those two contexts, but preserves the ability to use standard web authentication mechanisms and accommodates the needs of applications that manage their own users and serve a large-scale user base. The solution places responsibility for API definition and user authorization in the database schema. A comparative advantage over existing approaches is discussed.**

***Index Terms*—Structured Query Language (SQL), stored procedures, web applications, SQL injection, confused deputy attacks, cybersecurity, injection attacks, language-theoretic security**

I. Introduction

SQL injection, a class of software vulnerability wherein an attacker manipulates SQL database queries made by an application, has been a source of grief for some time.

As an injection attack, it can be viewed as a violation of the code-data boundary with respect to the serialization of a SQL query made by the vulnerable application. This viewpoint leads to several approaches to stopping it. Filtering (or "sanitizing") input is particularly common, though problematic for some input types. Some programmers opt to forego many relational database features to use an object-relational mapping (ORM) library which compiles SQL queries. Programs that do not need to generate dynamic SQL can use parametrized queries to put query parameters out of band from the data. It is also possible to write a limited-purpose compiler to generate dynamic SQL that will be constrained to the structures the application programmer intends. All of these defenses have significant drawbacks or cost. Further, all of them address how an attacker might send arbitrary queries to the database, but none of them address the consequences should an attacker manage to do so.

SQL injection can also be thought of as a confused deputy attack, because the application being attacked is confused into using its privileged access to the backend database to perform otherwise-unauthorized operations.

When an attacker succeeds at getting SQL injection to work, the effects can be as catastrophic as arbitrary read and write to the data and even its schema, resulting in a total loss of each of confidentiality, integrity, and availability. This impact is a consequence of applying a model wherein the front-end application, not the database, provides and implements the definition of what interactions are possible or permitted with the data by its user.

The purpose of a relational database management system is to provide a structure, called a schema, through which data may be stored and queried. The structure gives the types of data stored and a mechanical description of how they relate to another; one record, such as a customer, may refer to a number of associated records of different types, such as that customer's tickets. This gives rise to relational algebra, the method by which these records are efficiently queried. Databases traditionally include within the schema not only the relations (tables) in which the data are stored, but also sets of defined queries against these data in the form of views, stored procedures, functions, constraints, and triggers. Together, these schema elements ensure the well-formedness of the data in the database, including not only its type but also the concept of relational integrity. In this way, the database schema forms a protocol-like definition of data, though the data are stored and not interchanged.

The relational database was introduced in 1970 [5]. Since then, innovations have occurred along other lines, including the introduction of unstructured data stores. Although expedient for storing data from a variety of sources for later analysis among mutually trusting parties, the lack of a definition of this data renders it difficult to secure in mixed-access scenarios where a multitude of users have access to perform a subset of operations on a subset of the data. While the problem appears tractable, exchanging SQL for another query language and exchanging RDBMS for other data management paradigms do not make the problems of injection attacks and conditional access control go away [6].

A. Prior Work

Although there is some evidence for approaches like this one in industry, very little prior research concerns the encapsulation of a data API or a similar structure in a database, or authenticating application end users to such an API.

1) Many-Layered Approach: Some authors have proposed the use of parametrized queries, stored procedures, and least privilege, in combination, to prevent SQL injection, e.g. [2]. To this point, it does not appear that any have proposed an application of these techniques to create a layer that categorically prevents the success of SQL injection attacks; rather, these are to this point viewed as complimentary approaches to strategies like input filtering ("sanitization"), signature detection in a web application firewall, machine learning or clustering algorithms to characterize and winnow out malicious input, and escaping strategies. This falls short; rather than seeking a collection of partially effective measures, we should seek an architecture that solves the problem for good.

B. Stored Procedures

In describing the longevity of the SQLi problem, [11] hints at the use of stored procedures to put logic out of reach of the application, and at allow-listing queries to constrain the application; the OWASP guidance cited by that work[1] does not relate the two concepts or address how to constrain database callers to allowed operations. Reference [1] approaches a description of a data API, but does not address end-user authorization either, and protects the database with input sanitization rather than permissions even though the queries are rendered as stored procedures.

1) LangSec: If the web application part of an application stack is viewed as an interface to the database code, it certainly fails to minimize the complexity of pre-validation code as recommended in [13]. Rather, it implements a multitude of validations and transforms: request input, authorization, filtering, translation of the request into one or more SQL queries, interpretation of the output, and encoding of its response. Each of these layers might give rise to some vulnerability resulting from a parser differential. It therefore follows that these steps must be either verified, or eliminated, if we are to be able to end the reign of SQL injection.

Web applications can only be viewed this way because the database trusts the web application in the typical architecture; the web application is given full access to read and modify the data within its scope, if not administrator access to perform all operations on the database. If the database instead encapsulates the intended operations and authorization rules, the complexity of the application is irrelevant with respect to SQL injection. This will be the thrust of this work.

2) PostgREST: The PostgREST library, which directly translates arbitrary REST calls into arbitrary SQL, supports JWT authentication and passes JWT claims into the database as a transaction-scoped setting, so that they can be requested with `current_setting()`. While this use of transaction-scoped settings has many features of this work, the web application (in this case, merely PostgREST) is still responsible for validating the JWT and for asserting the claims it makes to the database. Checks for JWT revocation must be performed on the basis of this data, and the database does not validate the JWT itself. Therefore, an attacker who can inject SQL into the connection could forge trusted authentication data, impersonate another user, or resurrect a revoked token. Notwithstanding, the PostgREST library could be combined with this work to directly expose the data API to HTTP callers.

[1]https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

C. Impetus to Change

Although it is a porous and restricting mitigation [15], many authors still persist in recommending input sanitization as a technique to prevent SQL injection.

As at its last publication in 2021, the OWASP included injection attacks (including SQL injection) in its Top 10 list, indicating it remains a serious problem in spite of ubiquitous advice on its prevention. MITRE assigns CWE-89 to "improper neutralization of special elements used in an SQL command ('SQL injection')", a title which indicates the historical focus on escaping. 2,503 CVE records issued in 2024 mentioned SQL injection, and this captures only data about published software that provides detailed vulnerability disclosure. Other instances of SQL injection, for example instances of SQL injection affecting the proprietary software of cloud SaaS providers, are not captured in that count.

Removing complexity and simplifying data interchange are proven methods to reduce the incidence of security vulnerabilities. As commonly used between applications and their data stores, SQL exposes excess functionality that can be used by an attacker to modify (or even redefine) data, or simply make off with all the data in an application. This is a general property of data querying and modifying languages, and "noSQL injection" also exists as a vulnerability class. It therefore follows that subsetting the query language, not necessarily replacing it as with noSQL, offers a way to address this type of vulnerability. This accords with the standard LangSec approach that the power and complexity of interfaces should be limited to the minimum necessary, in the interest of security.

II. Removing the Vulnerable Layer

If SQL injection relies on exploiting parser differentials in the web application and then using the web application as a confused deputy to carry out the attacker's bidding, then it follows that an application architecture which does not rely on the web application to defend against these threats will not be vulnerable to them.

The following sections decompose the solution into these two parts: defining a data API, and then performing authentication and authorization in that data API. This removes responsibility from the web application layer in two areas; respectively, defining the full set of operations that can be performed against the data, and authenticating and authorizing the user. The web application then serves only to marshal and present the data to the user.

This strictly reduces the attack surface of the system in general. In existing architectures, the operations that can be performed against data with access to the database connection has been a superset of those operations which can be performed via the application. Moving this complexity to the database does not increase the computational power of the database as exposed to potential attackers, but it does decrease the complexity that the web application exposes to them.

III. Defining a Data API

While much latter work has focused on defining APIs in terms of REST and HTTP, nearly any protocol can define a coherent and useful API. To implement an API, it must be possible to define logical operations, with defined and typed parameters, against some data set. If the API has different behaviour depending on the identity of the entity interacting with it, then it must also support some kind of authentication scheme.

PostgreSQL clearly supports both of these building blocks: applications can authenticate to the database using a role, and it is possible to define stored procedures with which the application can then interact. Those stored procedures can perform defined create, retrieve, update, and delete operations. However, the authentication directly provided by PostgreSQL is not by itself sufficient for API-oriented use cases with large numbers of different callers; this will be elaborated on in following sections.

By defining an API with which the application can interact, it is possible to encapsulate the set of operations which may be performed on the database within one body of functions and procedures. They are separate from the application, and they can be logically separated from the rest of the database schema as well.

One can define such an API by defining a series of functions (which must return a value) and stored procedures (which may not return a value, but are otherwise for most purposes equivalent to functions) in the database. These each have the same basic structure: they check whether the caller is allowed to perform the operation requested with the parameters requested, perform the operation, and return the results, if any. In order to enforce the permissions, the functions and stored procedures are defined with the `SECURITY DEFINER` option; this is equivalent to a SUID executable to which unprivileged users have execute access on a UNIX system. The permissions of the privileged user may thus be used only in a structured way by the unprivileged caller [19]. Although this approach has been used in industry, it is rarely discussed in the literature.

The use of the `SECURITY DEFINER` primitive is pivotal to the function of this architecture. SQL generally permits subqueries to substitute for values, and therefore a malicious user might still be able to make arbitrary queries by passing them as parameters. With `SECURITY DEFINER`, the function parameters are still processed with the permissions of the invoker; therefore, the database is able to enforce the requirement to pass queries through the data API layer and will refuse to evaluate such subqueries. An approach which attempts to filter queries such that they only invoke the data API, while granting all permissions to the invoker as in a traditional architecture and avoiding the use of `SECURITY DEFINER`, may not succeed at enumerating all possible ways an arbitrary query could still be made.

IV. Binding Users to Database Roles

A. Database-Native Authentication

It is clearly undesirable for each application user to be granted individual database credentials which the operator of the database must then manage. First, application users are typically a construct of the application, and relate to data within the application's database. Roles exist outside of the relations. Second, databases with millions of defined roles are unusual therefore not well supported. Third, the need to connect to the database using some role complicates the manner by which the application can mediate the connection to the database: if a user presents a JWT, how does the application authenticate to the database while connecting as that user, and how is the credential in the database maintained? These issues are awkward enough that an alternative scheme is clearly needed.

B. Database Session Binding

Instead of having users authenticate directly to the database, it is possible to bind the user's authenticated session to the open database connection which makes their queries. This has some drawbacks; queries must then be predicated on a connection identifier, which must be bound to the user with an additional relation which would then need to be queried for nearly all operations. Moreover, it would not be possible to pool connections that could be used by multiple users, but connection pooling is often used to improve practical database performance in applications with many users.

C. Data-Tier Authentication

Alternatively, it is possible for the stored procedures comprising the data API to accept an authentication token which the application passes through to the database. This token could simply be a database-resident session token which the application passes through with every query. In applications that use stateless authentication, the token could be the same JWT used to access the application; there exists a pure SQL implementation of JWT verification.[2] This alternative is preferable because it limits the authorization scope to a single request; therefore, there is no possibility of data being injected into an authenticated channel by another user, and the application has the flexibility to pool, share, and re-route connections.

If the application does not benefit from the use of JWT, it is conceptually possible to implement this strategy by passing a session token or API token instead of a JWT; the database must then store a list of valid tokens and their associated relevant metadata, and query this at authentication time.

[2]https://github.com/michelp/pgjwt. An Oracle PL/SQL implementation also exists at https://github.com/morten-egan/jwt_ninja.

V. Example Code

This example gives a pl/pgsql listing of a stored procedure which effects a field update if and only if the user is authenticated and is performing the operation on a row they own. The authorization layer could be abstracted into a sub-procedure describing the check to be performed; the use of exceptions prevents further execution of the procedure in the event of a failure, even in a sub-procedure. This represents a recognizer pattern implementation of the authorization logic [4].

```
CREATE PROCEDURE AuthUser(
    jwt VARCHAR,
    targetUser INT
)
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
DECLARE
    authInfo RECORD;
BEGIN
    authInfo = verify(jwt, 'secret');
    IF NOT authInfo.valid
        THEN
        RAISE EXCEPTION
        'auth failed';
    END IF;
    IF NOT targetUser =
        (authInfo.payload->>'sub')::INT
        THEN
        RAISE EXCEPTION
        'unauthorized';
    END IF;
    -- Could also check JWT revocation here
END; $$;

CREATE PROCEDURE APIUpdateProfile(
    jwt VARCHAR,
    targetUser INT,
    newText VARCHAR
)
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
    -- Authorization Call:
    CALL AuthUser(
        jwt, targetUser);
    -- Could also make checks here and throw if they fail

    -- Application logic:
    UPDATE profile
        SET text = newText
        WHERE userid = targetUser;
END; $$;
```

A. Analysis

It is advantageous to choose a prefix for exposed data API functions and for authentication functions, in a manner similar to Hungarian notation, to allow trivial mechanical verification of the layers, since standard SQL does not offer a type system that can be used to verify these constraints.

The objectives of verification should be to show that:

1) The application user is only allowed to call functions and procedures from the data API, e.g. those starting with `API`
2) Every function in the data API, e.g. those starting with `API`, begins with one or more calls to a procedure from the authorization layer, e.g. those starting with `Auth`, or an annotation that no authorization rules apply to it
3) All authorization happens before other logic, i.e., authorization happens at the beginning, and then no authorization decisions are made following the start of non-authorization logic; this prevents shotgun parsing, making the next step much more tractable
4) The applied authorization rules embody the intent of the application and its security model

The fourth item is of course not mechanically verifiable, as it is a question of intent. The first three can be easily and deterministically verified through static analysis, as they are regular in complexity and have no runtime components. Verification of the fourth can be effected by a systematic manual review, but linear review of the code will suffice because it necessarily reduces to statements of the form "callers may perform this operation given these authorizations, with parameters such that...". This pattern offers an opportunity to prevent authorization bypass vulnerabilities caused by developer oversight.

A more general discussion of verification that the recognizer pattern is properly applied and effective follows in section VII.

VI. Comparison with Other Techniques

This method can be contrasted with other schemes used to provide a compensating control against application compromise or to remove responsibility for data protection from the application.

A. Application Firewalls

An application firewall, such as a web application firewall, is used to filter user queries and reject input that is suspected to be malicious. Such a firewall relies on definitions of known attack types; for instance, user input that contains SQL keywords, that contains substrings that are syntactically valid SQL statements, or that contain the SQL comment sequence `--` can be rejected. User input passes through the application firewall before being processed.

The application firewall is thus a validating parser. It could act as a validation layer for the underlying application if, and only if, its definitions represent a correct and restrictive parser for the application's valid input grammar. Unfortunately, application firewalls are typically unrelated to an application's input semantics and computation model; they seek to identify and block generic "malicious" input. They contain definitions for common attacks generally, but unless specific definitions for some application are amenable to description to the web application firewall and are written by developers for that purpose, they are limited to these generic definitions [17]. They are therefore subject to both false positive and false negative errors.

The error potential results from a fundamental differential between the firewall's definition of malicious input and what input might be malicious in the context of the application at hand. This is diametrically opposed to the present approach of incorporating permissions into the database schema, which avoids questions of input validity altogether and instead makes an application-specific decision as to whether some operation respecting the data should be allowed.

Later work in application firewalls magnifies the problem by attempting to dispense with the need to make definitions altogether in favour of stochastic machine learning approaches [7]. While this produces better results than attempts to build a general grammar for malicious input by hand, it still fails to address the fundamental application-specific nature of exploitation, and it is unsurprising that no work in this vein takes application specifications or code as part of its input.

Where domain-specific semantics are applied, it is often only in feature extraction and only with reference to a broad domain such as HTML because the desire is to build a non-application-specific model, for example [18].

Just as applications that do not rely on a multi-layered proxy architecture (as applications such as those built in Java Spring do, for example) do not benefit from protection against "request smuggling" offered by a WAF, applications built against a data API instead of a general SQL interface to the database will not benefit from such generic protection against SQL injection. The false positives remain a hindrance.

B. Delegated Authorization

A delegated authorization microservice makes real-time authorization decisions based on some predicate, and is invoked by application software when it needs to compute authorization policy [16]. Unlike the proposed model, the delegated authorization microservice does not offer structural protection against any type of attack; its purpose is to encapsulate complex policy evaluation logic. The proposed solution offers the same benefits provided that the relevant policy can be implemented in pl/pgsql or similar; though it is included in the database schema, structured programming techniques can separate it from other schema components.

Even if a delegated authorization microservice is used so that policy evaluation is encapsulated away from other application code, it still remains that some part of the application, whether that is its own data-interfacing microservice or several different microservices in the application, still has access to perform at least the union of all operations required by all its callers. Therefore, an attacker could still gain elevated access through an injection attack, provided that the predicates of the query as parsed by the application result in a positive authorization decision by the delegated authorization microservice.

In theory such a service could act as a security monitor by inspecting the user query for maliciousness, but this would be essentially identical to an application firewall; to be successful, it would need a precise definition of the protected application's input language. This would muddy the abstraction intended by making a delegated authorization microservice in the first instance.

C. Row-Level Security

One scheme for authorizing access to data within a relational database is to apply row-level security. This effectively adds a where-condition to DQL statements, and an additional check constraint to DML statements, depending on the user's role and what relations they have incorporated in their query. The condition and constraint can be arbitrary SQL, which allows use cases ranging from constraining a role to records only within one business unit to adding a security descriptor to each record in a relation. Though similar to the method described in this work, this scheme has two important drawbacks: it assumes every application user has a unique identity in the database's access control system, and it requires that rules for data access be expressed in the ways described. This latter drawback can reduce encapsulation; for example, consider a rule that an update time column must be set to the current time upon every user query that updates a record. This cannot be enforced by any of the security mechanisms, but rather by a trigger that updates the relevant timestamp after the update. If some user must be allowed to override these timestamps, still more complexity arises.

Row-level security as implemented in PostgreSQL, using the `CREATE POLICY FOR SELECT` and `CREATE POLICY FOR UPDATE` statements, makes use of a query modification approach. In essence, the terms of the policy are added to every query to which the policy applies. This offers a way to constrain some of the power offered by exposing a SQL interface, but the complexity and the questions attendant to it remain. For example, a query so restrained may appear superficially to be a self-contained query, but will return different results based on outside factors unrelated to the selection parameters in the query, and unrelated to the content of the relations that are relevant to those selection parameters. This annuls the soundness of the DBMS [20].

D. Input Sanitization

Although often mentioned, filtering user input, often called input "sanitization", is an ineffective technique for the prevention of SQL injection. There are two basic strategies for performing it. The first strategy is to filter out any characters which might be interpreted by one stage of the data processing pipeline as control, and either remove them or reject the input. Because SQL is a human-readable protocol, these characters encompass many printable characters which reasonably appear in data strings; therefore, it is often necessary to resort to a second strategy, escaping. Escaping either places an escape character before what might otherwise be a control character, or encloses data that might contain control characters which are not to be interpreted as control characters within some control suppression block like square brackets or quotes.

Among other problems with this approach, it is occasionally the case that a parser differential exists either between different stages of the pipeline or between the transform or filter and the database. Therefore, it may be possible to bypass input sanitization via double escaping, or by placing multi-character control tokens (in SQL, for example, the comment sequence `--`) separated by characters which will be removed. It is common to search for input sanitization bypasses using fuzz testing, the sending of random input, as it is difficult to prove that nothing in the input string will be executed as SQL code by the database (or that it will not modify or truncate some existing code).

For this reason, the sanitization approach is expensive and pourous, and violates encacpsulation by exposing SQL grammar to the application (for example, if "special characters" are prohibited in some input fields). Although it was once the consensus recommendation, it should be avoided.

A parallel discussion of why this approach is ineffective to protect HTML parsers is available at [12]. Though HTML processing is different from SQL processing, it is relevant that different dialects of SQL are differently injectable; while this work focuses on PostgreSQL, it could in principle be applied to any database that supports a JWT-parsing function, the definition of functions and stored procedures, and a restrictive permission model similar to PostgreSQL.

E. Parametrized Queries

Although parameterized queries[3] are very effective at preventing SQL injection [14], the application still remains responsible for enforcing authorization constraints and for protecting the database credential. Additionally, just as a stored procedure that contains an `EXECUTE` call may be injectable, some dynamically-constructed parametrized queries may be injectable. The same reasons call for dynamic construction in each case, particularly in scenarios like accepting dynamic user filters for data tables, where both table names and where-clauses would result in combinatorial explosion if not dynamically constructed.

Contrasted with the method of this paper, merely using parametrized queries with otherwise application-defined queries leaves the database open to attack by several classes of attacker:

1) Those who gain remote code execution on the application server, even if they can only see their own worker and therefore cannot steal other users' credentials.
2) Those who can disclose the server's configuration, including the database password, and also connect to the database either directly or through some proxy or alternative channel.
3) Those who can modify the application.

[3]Some authors call these prepared statements; for the avoidance of confusion with the `PREPARE` command, they are referred to here as parametrized queries.

Though the ability to modify the application implies a high degree of trust, using a data API with data-tier authentication can prevent threats in scenarios where multiple applications access the database and not all are under the same administrative control, even where the applications require overlapping access to relations and cannot be mutually isolated via the database's permission system.

F. Identifier Masking

In 2004, reference [3] proposed to include a random string, configured in the application, appended to every SQL identifier. A validating proxy then checks for the presence of the correct random string on every SQL identifier in incident queries and strips it out. This approach has several drawbacks. First, it does not protect against payloads that do not state any identifiers. Second, the random string is either hardcoded or configured in the application, so that an attacker who can disclose local files could also disclose the configured random string. Third, an attacker given unlimited attempts could guess the string and their guess would remain valid until the application is re-configured.

This approach is, as the authors state, essentially an application of the same strategy as instruction set randomization. It could be made more general by creating an arbitrary substitution map for identifiers in the application, so that injected strings would be nonsense (it is not clear whether a block or stream cipher would succeed, since the challenge is to find those specific *characters* which have been introduced by an attacker rather than by the application). This generalization would have the same drawbacks.

Additionally, the proxy introduces an additional layer of authentication and another parser with the attendant threats from incorrect implementation. Authentication to either the database or the proxy would suffice to grant access to the database, and the proxy represents another software package that might store the database credential somewhere an attacker could get it. Database SQL parsers are notoriously complex, and parser differentials between the validating proxy and the DBMS are probable unless the same parser can be used in both cases, since the SQL is being parsed twice; it is therefore not out of the question that there could be SQL injection in the validating proxy.

G. Control Flow Analysis

Some authors propose to build a model of intended application program execution, either using static analysis [9] or taint tracking [10], and check whether queries that are sent to the database could have been constructed by the application. If they could not have been, then they must be SQL injection. This approach has two critical problems. First, not all instances of SQL injection must make queries that the application would not make; an attacker may simply wish to make queries that they personally are not allowed to make, or even simply to make the queries out of order or with invalid parameters. Such a model, to succeed, would therefore need to incorporate not only the full complexity of the likely Turing-complete application code, but also the complexity of the data processed by the application and of the application's authorization functionality. The model evaluation could therefore be even more complex even than the vulnerable application was in the first place. Second, this approach can only be applied where the target program is amenable to the static analyzer, placing limitations on how the application can be written and what languages it can be written in.

If it is possible to build a model of all queries generated by an application, it follows that it should be possible to incorporate these queries into the schema of the database and dispense with the complexity of the checker.

VII. Security Analysis

A database architected in this way requires only two analyses to be secure:

1) For all functions and stored procedures, that the operations they perform are the intended ones; that is, that authentication is checked if required and the code is predicated upon it according to the design of the API.
2) For only those functions and stored procedures which contain an execute primitive, such as the pl/pgsql `EXECUTE` keyword or any DDL statements, that the resulting operations are within the gamut of what is intended by the application.

Both analyses are predicated on a complete threat model, and therefore this does not provide an absolute guarantee against security issues, but rather only guarantees that those considered in the threat model are detectable. The latter analysis additionally requires, to preserve this detectability, that the allowed execution be amenable to analysis; for example, it should be as computationally simple as possible. Ideally it should be simple enough to verify using inductive logic or exhaustive enumeration, and in no case should it be more complex than deterministic context-free.

It is still possible that a user might be able to compose functions and SQL operations, for example by using one data API function as a parameter to another, or by making queries against the results of a table-valued function. While this causes the database to perform additional computation on the user's behalf, this does not constitute a security threat with respect to the data because the operations performed in this way are a subset of those which could be performed client-side anyway. This holds provided that general threats against the DBMS, described in the following section, are addressed.

Additionally, without extension, this work does little to prevent timing attacks and query composition attacks. These are more general problems that are outside the scope of SQL injection. However, it is possible that some cases of query composition attacks could be defended against by defining authorization rules using this work, where those authorization rules are predicated upon a user's previous queries.

Some authors have noted the potential for SQL injection vulnerabilities in JWT processing, which is one of the major attack surfaces relevant to JWT use [21]. These vulnerabilities occur as a result of trusted, valid JWTs issued with SQL injection payloads in some relevant field such as the audience or the claims. Since this work treats the JWT as a data string in transit and then the database parses it according to the JSON specification, there is no potential for SQL injection; the code-data boundary between application code and the JWT contents is maintained.

A. PostgreSQL General Threats

The various means required to secure a PostgreSQL database against privilege escalation are not within the scope of this work, but will be discussed non-exhaustively in brief.

PostgreSQL implements a search path across several schemae to bind symbols; if users are permitted to define schema objects in a schema that precedes that of another schema object in the search path, it is possible for malicious users to rebind those schema objects when they are called by a more privileged caller. This rebinding attack may be invoked on operators or defined functions, for example, and could lead to elevation of privilege that would in turn permit the control described in this work to be bypassed.

Additionally, a user who is allowed to install extensions in the database with the `CREATE EXTENSION` statement could install an extension which performs operations that exceed the scope of intended access. Though this requires the plugin code to be present on the database server's disk, there are several strategies (including the large object API in PostgreSQL) to cause arbitrary data to be written to disk.

Those permissions which are gated behind the `pg_execute_server_program` permission also allow the user to execute arbitrary programs on the database server, which must also be avoided for the method described in this work to be effective.

It is also imperative that the application user not be granted permissions to read, write, or execute server-side files as described in the `COPY` command, and that the application not trust the standard output of the process handling the database connection. In the latter case, `COPY TO STDOUT` could result in response injection.

It is for these reasons that a user of the data API should not be allowed to perform any operations except calling the data API. Fortunately, PostgreSQL supports a default-deny permission model that lends itself to this task.

VIII. Conclusion

This work has proposed replacing the diverse schemes currently used to prevent SQL injection by creating a layer in the database which implements a prescriptive API, including authorization rules. This removes the responsibility for enforcing authorization and access procedures from the application entirely, so that even if an attacker can send requests directly to the database using the application's credentials, the attack will yield no more access than the user already obtains by using the application. The authorization rules become an atomic part of the schema, eliminating the possibility for injection attacks because the application is no longer trusted by the database. The database, in turn, uses the recognizer pattern on authorization rules to reject as invalid any unauthorized access prior to side effects.

This categorically prevents confidentiality and integrity attacks, and permits verification of the correctness of the application's data model using only the database schema.

Some extant applications could be retrofitted with this strategy if the non-data components of the database queries they make are a finite enumerable set (which they arguably should be, as follows from [15]), and if the application's authorization rules can be evaluated by the database. To do so requires significant engineering effort, but successful application of this strategy yields improved encapsulation and permits developers to dispense with other mitigations like input sanitization and web application firewalls, at least in the context of SQL injection.

References

[1] K. Ahmad and M. Karim, "A Method to Prevent SQL Injection Attack using an Improved Parameterized Stored Procedure," Int. J. of Advanced Computer Science and Applications, vol. 12, no. 6, pp. 324-332, 2021. doi: 10.14569/ijacsa.2021.0120636

[2] N. D. Bobade, V. A. Sinha, and S. S. Sherekar, "A diligent survey of SQL injection attacks, detection and evaluation of mitigation techniques," 2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS), pp. 1–5, Feb. 2024. doi: 10.1109/sceecs61402.2024.10481914.

[3] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks" in Appl. Cryptography Netw. Secur., Berlin, Germany: Springer, pp. 292-302, 2004.

[4] S. Bratus et al., "Curing the Vulnerable Parser: Design Patterns for Secure Input Handling," ;login:, vol. 42, no. 1, pp. 32-39. [Online]. Available: https://langsec.org/papers/curing-the-vulnerable-parser.pdf

[5] E. F. Codd, "A relational model of data for large shared data banks," Communications of the ACM, vol. 13, no. 6, pp. 377-387, Jun. 1970. doi: 10.1145/362384.362685.

[6] P. Colombo and E. Ferrari, "Fine-Grained Access Control Within NoSQL Document-Oriented Datastores," Data Science and Engineering, vol. 1, no. 3. Springer, pp. 127–138, Aug. 2016. doi: 10.1007/s41019-016-0015-z.

[7] S. Dhote, A. Magdum, S. Singh and D. Raigar, "ML based Web Application Firewall for Signature and Anomaly Detection Using Feature Extraction," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-6, doi: 10.1109/ICCCNT61001.2024.10725511.

[8] N. Gharpure and A. Rai, "Vulnerabilities and Threat Management in Relational Database Management Systems," 2022 5th International Conference on Advances in Science and Technology (ICAST), Mumbai, India, 2022, pp. 369-374, doi: 10.1109/ICAST55766.2022.10039599.

[9] W. G. J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183, Nov. 2005. doi: 10.1145/1101908.1101935.

[10] W. Halfond, A. Orso and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation", IEEE Transactions on Software Engineering, vol. 34, no. 1, pp. 65-81, Jan. 2008. doi: 10.1109/TSE.2007.70748.

[11] M. Horner and T. Hyslip, "SQL Injection: The Longest Running Sequel in Programming History," The Journal of Digital Forensics, Security and Law. ERAU Hunt Library Digital Commons Journals, 2017. doi: 10.15394/jdfsl.2017.1475.

[12] D. Klein and M. Johns, "Parse Me, Baby, One More Time: Bypassing HTML Sanitizer via Parsing Differentials," 2024 IEEE Symposium on Security and Privacy (SP), pp. 203–221, May 2024. doi: 10.1109/sp54263.2024.00177.

[13] F. Momot, S. Bratus, S. M. Hallberg and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," 2016 IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 2016, pp. 45-52, doi: 10.1109/SecDev.2016.019.

[14] J. O. Okesola, A. S. Ogunbanwo, A. Owoade, E. O. Olorunnisola and K. Okokpuji, "Securing web applications against SQL injection attacks - A Parameterised Query perspective)," 2023 International Conference on Science, Engineering and Business for Sustainable Development Goals (SEB-SDG), Omu-Aran, Nigeria, 2023, pp. 1-6, doi: 10.1109/SEB-SDG57117.2023.10124613.

[15] E. Poll, "LangSec Revisited: Input Security Flaws of the Second Kind," 2018 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 2018, pp. 329-334, doi: 10.1109/SPW.2018.00051.

[16] D. Preuveneers and W. Joosen, "Access Control with Delegated Authorization Policy Evaluation for Data-Driven Microservice Workflows," Future Internet, vol. 9, no. 4. MDPI AG, p. 58, Sep. 2017. doi: 10.3390/fi9040058.

[17] A. Razzaq, A. Hur, S. Shahbaz, M. Masood and H. F. Ahmad, "Critical analysis on web application firewall solutions," 2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS), Mexico City, Mexico, 2013, pp. 1-6, doi: 10.1109/ISADS.2013.6513431.

[18] A. Shaheed and M. H. D. B. Kurdy, "Web Application Firewall Using Machine Learning and Features Engineering," Security and Communication Networks, pp. 1–14, Jun. 2022. doi: 10.1155/2022/5280158.

[19] B. Shaik, "User Management and Securing Databases," PostgreSQL Configuration. Apress, pp. 61–92, 2020. doi: 10.1007/978-1-4842-5663-3_3.

[20] Q. Wang et al., "On the correctness criteria of fine-grained access control in relational databases," Proceedings of the 33rd International Conference on Very Large Data Bases, Sep. 2007, pp. 555-566. [Online]. Available: https://www.vldb.org/conf/2007/papers/research/p555-wang.pdf

[21] I. Zulkarneev and K. A. Basalay, "JSON Web Tokens Lifecycle-Based Threat Classification," 2024 IEEE 25th International Conference of Young Professionals in Electron Devices and Materials (EDM). IEEE, pp. 1920–1924, Jun. 28, 2024. doi: 10.1109/edm61683.2024.10615042