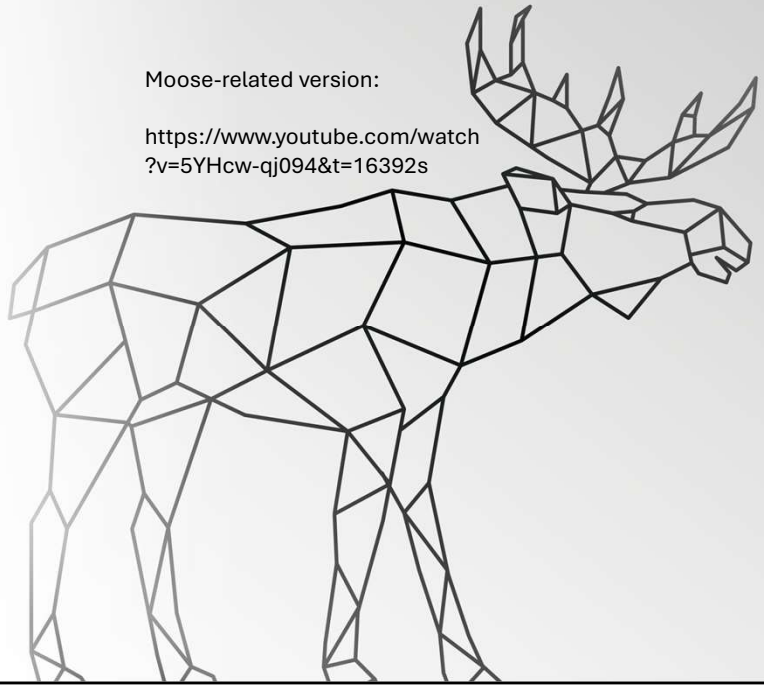
Removing the Vulnerable Webapp

Falcon Darkstar Momot

Product Security, https://aiven.io

2025 Work in Progress ed.



Moose-related version:

https://www.youtube.com/watch?v=5YHcw-qj094&t=16392s

SQLi: Confused Deputy

Web applications implement an authorization boundary
consisting of the business logic of the application

Therefore they have at least as much permission as their caller

They also represent the typical dangerous admixture of Turing-complete program logic and access control decisions that need to be verifiable

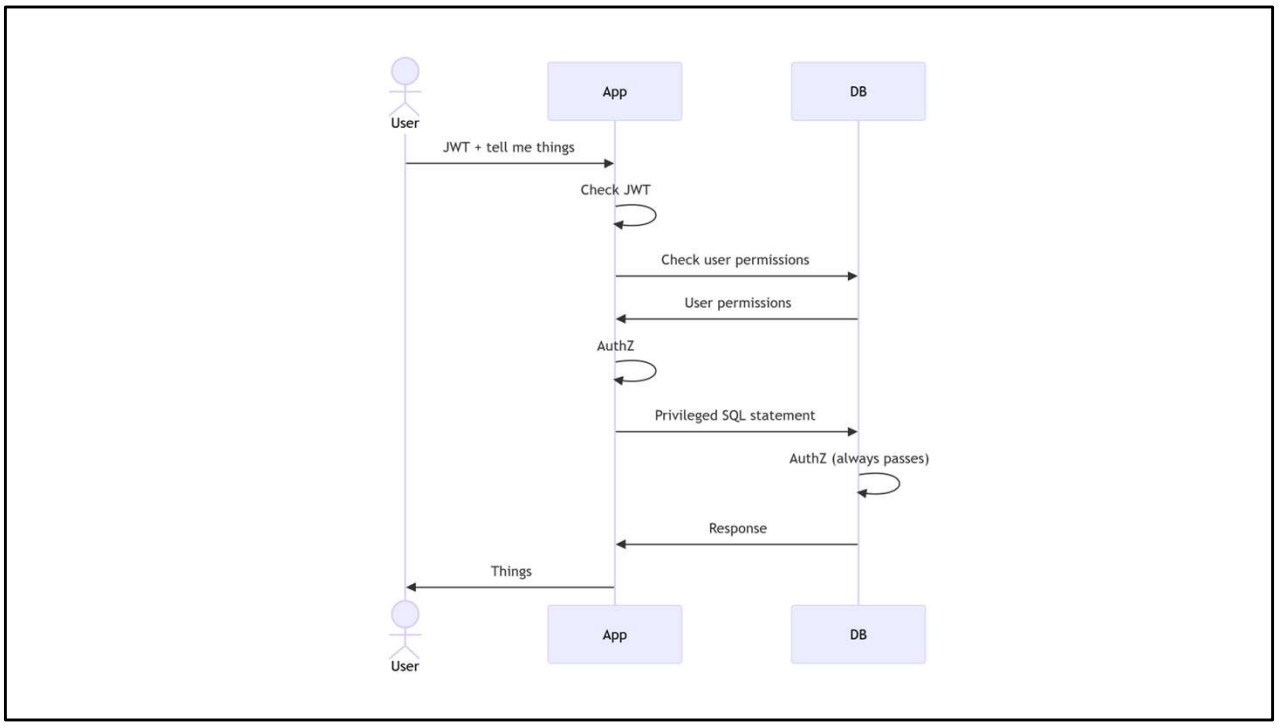
SQLi… why?

We use a powerful, almost general-purpose programming language as a data interchange format

INSERT INTO sanitizers VALUES (‘soap’, ‘bleach’, ‘alcohol’, ‘hot water’);

It is because of its power that it’s a problem.

The insert statement is a DML statement in SQL, a programming language, but we send it around as though it were a piece of data. Stop.




```
sequenceDiagram
    actor User
    User->>App: JWT
    App->>App: Check JWT
    App->>DB: Check user permissions
    DB->>App: User permissions
    User->>App: Tell me things
    App->>App: AuthZ
    App->>DB: Privileged SQL statement
    DB->>DB: AuthZ (always passes)
    DB->>App: Response
    App->>User: ThingssequenceDiagram
    actor User
    User->>App: JWT + tell me things
    App->>App: Check JWT
    App->>DB: Check user permissions
    DB->>App: User permissions
    App->>App: AuthZ
    App->>DB: Privileged SQL statement
```

```
DB->>DB: AuthZ (always passes)
DB->>App: Response
App->>User: Things
```

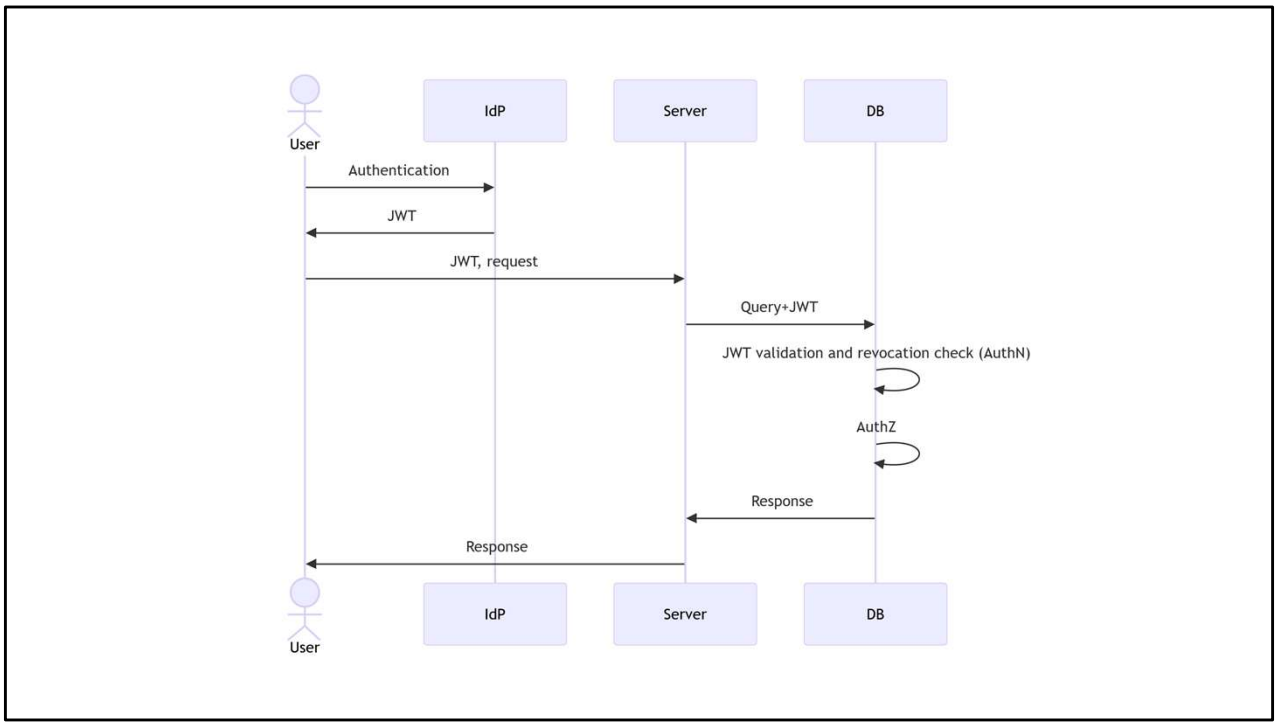
A Short Manifesto

1. Encapsulate the access control decisions in a layer
2. Force all request and data flow through that layer

If that layer is in the DB, the app no longer needs to be trusted

Also, we can make access control part of the data model:

No more checking the wrong access control predicate


User
IdP
Server
DB
Authentication
JWT
JWT, request
Query+JWT
JWT validation and revocation check (AuthN)
AuthZ
Response
Response
User
IdP
Server
DB

Example

```
CREATE PROCEDURE updateProfile (jwt TEXT, updata JSON)
      SECURITY DEFINER AS $$
DECLARE uid UUID := checkUser(jwt);
BEGIN
      WITH n AS (SELECT * FROM
            json_populate_record(NULL::profile, updata))
      UPDATE profile SET foo=n.foo, bar=n.bar
      FROM n
      WHERE id = uid;
END; $$ LANGUAGE plpgsql;
```

Notice that the record to access is inferred from the JWT, and we don't include the UID in the copy so the user can't change their UID, etc.
This defines the allowed operations on the data as part of the schema, where the user parameter is intrinsically authenticated

At this point you could actually deploy your thing with postgrest, or even give external users DB creds, and it would not matter. (of course, PCI for example bans this, because old ideas never die).

SECURITY DEFINER?!?!?!

Yes. I am completely serious.

The worst-case scenario is the status quo.

Real applications use this pattern.

Innovations here:
- Combine with pgJWT
- SQL channel ceases to be a privileged context

Aren’t Prepared Statements Enough?

```
listRelation = ‘sanitizers’
foreach {‘bleach’, ‘soap’} as listEntry:
     db.query(‘INSERT INTO {} VALUES ($a)’
          .format(listRelation),
          {‘a’: listEntry})
```

It is possible to use a compiler / SAST to make dynamic SQL safe

But the app still has the privileged DB credential.

LFI, env disclosure, creds in source code, etc.

Prepared statements only protect direct escapes from the data channel
Consider code execution, SSRF, debug code left in prod, mistakes made after a SQLi rule is suppressed in static analysis
Consider complex queries; see my 2023 presentation The Un-parsing Manifesto which used SQL as an example for discussing output encoding (unparsing)
AKA writing compilers to restrict possible output