intrbiz

Protecting Your Customers With PostgreSQL

Practical Ways To Look After Your Data

PG Conf EU 2017

Chris Ellis - @intrbiz

Hello!

- I’m Chris
  - IT jack of all trades
- Been using PostgreSQL for about 12 years
- Very much into Open Source
  - Start Bergamot Monitoring - open distributed monitoring
- Been working on large smart energy analytics for the last few years
  - Strange mix of OLTP and OLAP
  - Quite a bit of customer data, hence this talk

Setting The Scene

- This talk has come from the various efforts me and my team have implemented to protect our customers data, I wanted to share my learnings
- This talk is targeted at people who are building application with PostgreSQL, rather than running third party application on top of PostgreSQL
- Please don't see this talk as a: `we should be doing this`. You need to decide for yourself which approaches work the best for you in your situation.

- I'd prefer this talk to be a discussion and not a lecture, please feel free to ask questions :)

Why Bother With This Security Stuff?

- Large scale data loss is increasingly in the headlines
  - Reputational damage cost you
  - Do you want to be the next: Talk Talk, Yahoo, Equifax?
- Legal
  - Obligations under various data protection legislation
  - GDPR comes into force next year, could be fined 5% of revenue!
- Professional
  - You don’t want data loss to follow you around, feels nice to do a good job
- Compliance
  - In a large corp, you don’t want the Info. Sec. team down your throats!
  - You may have to comply with external regulations, eg: PCI DSS

But .... Heard It All Before

- We have a firewall
  - Just protecting your perimeter - fail!
- We won’t get hacked, no one cares about us
  - Hackers certainly don’t
- Not a priority now
  - It’ll be a priority when it is too late
- Security is expensive
  - Companies spend about 1/20th of development cost on security

Security Is An Onion, Not A Balloon

- You will get hacked, plan on that assumption, not the other way around
- Defense In Depth
  - You can't just protect the perimeter, threats aren't just external
  - An attacker should need to exploit multiple layers
- Failsafe
  - Each layer should failsafe, contain an attack rather than facilitate it
  - Least privilege: I want the least amount of permission to acheive what I need to do
- Challenge
  - Security is the responsibility of everyone: devs, ops, dbas, business. Not just Info. Sec.
- Detect, Deceive
  - Would you know if you've been breached, attacks are often slow and unnoticed
  - All Warfare Is Based On Deception

Foundations

Foundations: Standing On The Shoulders Of Giants

- Crunchy Data have done awesome work on securing a PostgreSQL install
  - It's pointless me repeating it
  - Get it here: http://info.crunchydata.com/blog/postgres-stig-disa-security-guide
  - Read it
  - Implement what is sensible for your deployment
- They've also gone great work on pg_audit
  - Again pointless me repeating it

Foundations: TLSing Connections

- Running in a cloud environment, it's hard to keep tabs on where traffic will flow
- Running without TLS between our application and database wasn't an option
  - Damn info. sec.
- But Debian / Ubuntu has TLS enabled by default
  - Well, yes and no
  - TLS without a chain to trust is pretty pointless
- Doing TLS properly
  - Get a real certificate signed by a real CA
  - Set up your own CA
    - Easily done via OpenSSL

Foundations: Encrypted Storage?

- In some environments you don't have control of the storage layer
  - How can you prevent someone copying your whole database
  - How can you demonstrate the destruction of data
    - Easy to prove you shredded those $40k worth of SSDs
- You might need to run your whole database on an encrypted volume
  - On Linux this can be done via dm-crypt
  - Encryption has some overhead, every disk read and write requires additional CPU time
    - Thankfully modern CPUs have dedicated instructions to improve AES performance
  - You might be able to get away with encrypting certain table spaces
    - Be careful of temp files, temp tables, etc
  - There are schema level options, with lots of tradeoffs, out of scope for this talk

Foundations: Where Is Your Schema?

- You should manage your database schema as you would code
  - Put it in source control
  - Make it visible, reviewable, manageable
  - One SQL file per entity
  - Wrap it in a simple build process: a little bash script, maybe make
  - You can easily do single shot migration scripts using a function, no need for fancy tools
  - Use transactions: make your deployment atomic
- Don’t
  - Only keep your schema in a database
    - Then trying to do pg_dump | psql to patch other dbs
  - Store it as a set of patch scripts
    - Where deploying consists of applying script after script in order

Building Blocks

Building Blocks: Roles, The More The Better

- Roles are things which can be given permission over your database
  - Roles could be people, teams, or more abstract
  - Roles in PostgreSQL are super flexible and super powerful for controlling access
  - You can never have enough roles
    - Define roles for each logical group of functionality in your database
    - Aggregate your fine grained roles into higher level roles to simplify maintenance
  - Got multiple applications using your database, each application should have its own role
  - Does your application role have the least privilege it needs?
- Don’t:
  - Your application role is a super user - just no!
  - Your application role is the database owner
  - Your developers are super users - sure, they occasionally need it, but not by default

Building Blocks: Grant, Revoke

- Once you've got roles defined, you want to control what they can do
  - Revoke all permissions from public
    - No point in controlling what roles can do, if public can do everything
  - Grant roles only the access they need
    - Define and manage your roles and grants in your schema repository
    - Only grant usage to schemas that the role needs
    - Only grant the specific permissions that a role needs over a table
    - Start with the minimum, deploying schema updates is easy, databases are in flux
  - If your tables contain sensitive data, you might want to grant access at the column level
    - This will break: SELECT * FROM …

Building Blocks: Row Level Security

- Row Level Security is very powerful, however didn’t suit our use case
  - Hard to define a separate user for each customer in our system
    - Don’t really want to deal with PostgreSQL with 11M users
    - Had I found out about set_user before, might have been more possible
- Row Level Security still great for enforcing least privilege
  - You can filter certain rows from certain applications
  - You can filter certain rows for types of users in your application

Functional Interfaces

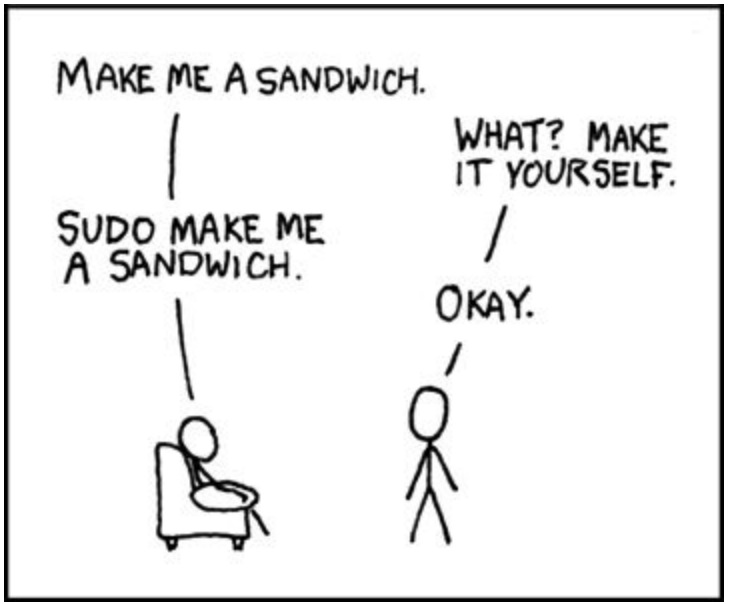
Functional Interfaces: What?

- Your database provides an API for your application via functions, rather than directly querying entities
  - SELECT * FROM get_user_by_username('chris');
- Gives your database developers flexibility, they can change entities without impacting the application, reduced coupling
- Provides a strict, enforced and controllable interface for what your application can do with your database. Just like we do at an application API level
- PostgreSQL has many features to provide rich APIs, EG: JSON in and out
- At first it will take more development effort, depends on your team make up, if you don't have dedicated database developers it won't be ideal

Functional Interfaces: Secure?

- Let’s consider what happens if your application is vulnerable to SQL injection
  - An attacker provides `‘; SELECT * FROM customer; --` as an input
  - The attacker has then managed execute: `SELECT * FROM customer`
    - You’ve just exposed lots of customer data
- If we go via a function instead
  - An attacker provides `‘; SELECT * FROM customer; --` as an input
  - Sadly still the attacker has managed to execute `SELECT * FROM customer`
    - You’ve still exposed lots of customer data
- Using a functional interface will prevent certain attacks
- But it won’t mitigate SQL injection attacks
  - However….

Functional Interfaces: Security Definer



Functional Interfaces: Security Definer

- Security definer means your function executes with the permissions of its owner rather than the permissions of the role executing it
- This provides privilege separation
  - Just like sudo
  - We provide a tight, reviewed and secure interface to lower privileges roles
- Now I can grant the application role only permission to execute the function
- The role which owns the function can be granted access to the underlying tables
- We fail safe, the application cannot directly access data, it can only do what the function allows it

Functional Interfaces: Security Definer Example

```
CREATE ROLE talk_api_function WITH
  NOSUPERUSER NOCREATEDB NOCREATEROLE NOLOGIN NOREPLICATION
  NOBYPASSRLS;

CREATE ROLE talk_api WITH
  NOSUPERUSER NOCREATEDB NOCREATEROLE NOLOGIN NOREPLICATION
  NOBYPASSRLS;

CREATE USER app_talk WITH
  LOGIN NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE NOREPLICATION;
GRANT talk_api TO app_talk;
```

Functional Interfaces: Security Definer Example

```
CREATE SCHEMA api AUTHORIZATION cellis;
REVOKE ALL PRIVILEGES ON SCHEMA api FROM public;
GRANT USAGE ON SCHEMA api TO talk_api;

CREATE SCHEMA customer AUTHORIZATION cellis;
REVOKE ALL PRIVILEGES ON SCHEMA customer FROM public;
GRANT USAGE ON SCHEMA customer TO talk_api_function;
```

Functional Interfaces: Security Definer Example

```
CREATE TABLE customer.customer (
  id             UUID NOT NULL PRIMARY KEY,
  full_name      TEXT,
  preferred_name TEXT,
  email_address  TEXT,
  mobile_number  TEXT,
  postal_address TEXT,
  postcode       TEXT
);

ALTER TABLE customer.customer OWNER TO cellis;
REVOKE ALL PRIVILEGES ON TABLE customer.customer FROM public;
GRANT SELECT ON TABLE customer.customer TO talk_api_function;
```

Functional Interfaces: Security Definer Example

```
CREATE OR REPLACE FUNCTION api.get_customer(p_id UUID)
RETURNS SETOF customer.customer
LANGUAGE plpgsql SECURITY DEFINER AS $$
BEGIN
    RETURN QUERY SELECT * FROM customer.customer WHERE id = p_id;
END;
$$;
ALTER FUNCTION api.get_customer(UUID) OWNER TO talk_api_function;

REVOKE ALL PRIVILEGES ON
  FUNCTION api.get_customer(UUID) FROM public;

GRANT EXECUTE ON FUNCTION api.get_customer(UUID) TO talk_api;
```

Functional Interfaces: Secure?

- Let’s go back to our SQL injection vulnerable application
  - An attacker provides `‘; SELECT * FROM customer; --` as an input
  - The function call is still bypassed, by the injection
  - However the application does not have permission to directly access the table
  - Instead of exposing customer data the attacker gets an error message
  - The data layer has failed safe and contained the attack, rather than facilitated it
- Let’s think about a bigger application vulnerability, imagine the attacker has gained total control of your application, how might they extract your customer data
  - The functional interface restricts them to accessing 1 record at a time
  - The attacker needs to guess your identifiers
    - Don’t expose sequential identifiers publicly

Functional Interfaces: Caveats

- If your security definer functions are generating dynamic SQL, be careful
  - Just like with sudo, if you allow a user to run a user defined command as root, your owned
- Make sure you use:
  - `quote_ident`
  - `quote_literal`
  - `EXECUTE … USING ...`

Functional Interfaces: PL/Proxy

- If you've gone down the functional interface road, then you can leverage extensions such as PL/Proxy
- PL/Proxy allows you to define functions which proxy to a remote PostgreSQL database, this is great for sharding and scaling your database
- PL/Proxy also allows you to place another layer between your application and your actual data
  - The application can only connect to the proxy database, which is stateless, contains no data
  - The proxy database then proxies function calls to the backend database
  - To extract data, an attacker now needs to exploit multiple layers
- For sure, this requires more development effort, you now need to write two function definitions for everything

Masking Data: Views

- You can easily use views to hide portions of data
  - Or apply one way transformations (eg: hashing)
  - Revoke privileges from the underlying table
  - Grant permissions to the view
- When using a view to mask data, you need to be careful
  - The view needs to be marked security definer
  - Functions used by the view should be marked leakproof

Detecting And Deceiving: pg_decoy Example

```
CREATE OR REPLACE VIEW api.customer
WITH (security_barrier=true)
AS
 SELECT id, full_name, preferred_name, md5(email_address) AS
email_address, md5(mobile_number) AS mobile_number, NULL AS
postal_address, postcode
 FROM customer.customer;

ALTER TABLE api.customer OWNER TO cellis;

REVOKE ALL PRIVILEGES ON api.customer FROM public;
GRANT SELECT ON api.customer TO talk_api;
```

Detecting And Deceiving

Detecting And Deceiving

- We operate on the assumption we will be hacked
- Therefore detecting that we've been hacked is important
  - There are various tools which can help
    - IDS/IPS either network or host based
      - Got one of them right
    - Log analysis systems
      - But you're already doing this right
- Honeypots are an interesting option
  - Designed to lure in attackers and keep them occupied
    - Allows you to detect and observe them
    - Keep the occupied and away from the real deal

Detecting And Deceiving

- What if we can create fake table, which when queried raise the alarm
- PostgreSQL is flexible enough to give us some options
  - Using functions and views
  - Using a foreign data wrapper
- Both methods have pros and cons
- I figured writing a FDW with Multicorn would be a simple proof of concept
  - Couple of hours, 70 odd lines of python
  - On select of a table will fire call to Bergamot Monitoring or any HTTP webhook
  - On Github: https://github.com/intrbiz/pg_decoy

Detecting And Deceiving: pg_decoy Example

```
CREATE SERVER my_decoy FOREIGN DATA WRAPPER multicorn OPTIONS (
    wrapper 'PGDecoy.PGDecoyFDW',
    driver 'bergamot',
    host 'demo.bergamot-monitoring.org',
    key 'SSmV5Zxq54SLS28OM3sNFPNaHlQTb',
    trap '2979259f-9599-44e5-b797-670458141c84'
);
```

Detecting And Deceiving: pg_decoy Example

```
CREATE FOREIGN TABLE customers (
    id UUID,
    username TEXT,
    password_hash TEXT,
    email TEXT,
    full_name TEXT,
    pref_name TEXT,
    mobile TEXT
)
SERVER my_decoy
OPTIONS (
    pot 'customer'
);
```

Developer Two Factor Authentication

Developer Two Factor Authentication

- Developers and especially DBAs usually have a lot of access to the database
  - Hands up who has super user access to production
- These accounts are very valuable to attackers
  - PostgreSQL's MD5 auth is pretty bad, move to SCRAM with 10
- Lots of systems are moving towards two factor authentication, what if we would do that with PostgreSQL?
  - Would love to be able to use my Yubikey to authenticate with PostgreSQL
- PostgreSQL supports multiple authentication systems
  - We can use RADIUS to delegate the password verification to an external system