Securing PostgreSQL Applications

Marc Balmer <marc@msys.ch>

Prague PostgreSQL Developer Day 2015

About this presentation

- Aimed at application developers and DBAs. . .
- . . . who do client programming using libpq (or a wrapper around libpq for a language other than C)
- . . . who can use PostgreSQL features

Topics

The risks

- Data stealing (personal information, credit card details)
- Data manipulation
- Data destruction

Attack vectors

- SQL-injection
- Direct database access, e.g. as a result of . . .
- . . . a server break-in (shell access) with limited (non-root) access
- . . . a server break-in with root-access

(Speaking of Unix-like systems here. . .)

SQL-injection

- Attacker manages to inject SQL, usually using a bug in the software
- Access to the database with all privileges of the logged-in user
- Often only one single user for the whole application
- Even more so in DB agnostic applications, e.g. MediaWiki

Server break-in (non-root)

- Database can be accessed if user credentials are known
- Search in config files for credentials, a careless sysadmin may leave them world readable

Server break-in with root access

- You have lost. Game over.

SQL-injection

```
sql = string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
]],
gui.entry.firstname:GetString(),
gui.entry.lastname:GetString(),
gui.entry.town:GetString())

conn:exec(sql)
```

No problem with ,,normal” input

```
local a = 'Marc'
local b = 'Balmer'
local c = 'Basel'

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
]], a, b, c)
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', 'Basel')
```

Never construct an SQL statement in this naive way . . .

. . . an attacker will try to get malicious code into a variable that is used in an SQL statement.

Injecting SQL code

```
local a = 'Marc'
local b = 'Balmer'
local c = <- try to get our own SQL into this

sql = string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
]] a, b, c)
```

INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', 'our own SQL')

Step 1: Close the string and terminate the original SQL statement

```
');
```

INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', '');)

$\rightarrow$ **');)** gives a syntax error

Step 2:
Turn the remainder of the original SQL statement into a comment

```
'); --
```

INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', ''); −−)

→ no more syntax error, the second brace is commented out, we now have room for own SQL between `');` and `--)`

Step 3:
Add our own SQL code

```
'); truncate person; --
```

INSERT INTO person (firstname, lastname, town)
VALUES ('Marc', 'Balmer', ''); truncate person; −−)

→ we can execute arbitrary SQL commands

SQL-injection in practice

How to prevent SQL-injection

- Escape **ALL** input from **ALL** sources
- . . . also cookie valies, input of devices etc.
- Use prepared statements
- Use parametrized queries

Inserting data with escaping

```
sql = string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
]],
conn:escapeString(gui.entry.firstname:GetString()),
conn:escapeString(gui.entry.lastname:GetString()),
conn:escapeString(gui.entry.town:GetString()))

conn:exec(sql)
```

Escaping prevents SQL-injection

```
local a = 'Steve'
local b = 'B.'
local c = "'); truncate person; --"

conn:exec(string.format([[
INSERT INTO person (firstname, lastname, town)
VALUES ('%s', '%s', '%s')
conn:escapeString(a), conn:escapeString(b), conn:escapeString(c))
```

```
INSERT INTO person (firstname, lastname, town)
VALUES ('Steve', 'B.', '''); truncate person; --')
```

Inserting data with prepared statements, preparation step

```
conn:prepare('safe_entry', [[
INSERT INTO person (firstname, lastname, town)
VALUES ($1, $2, $3)
]], '', '', '')
```

Inserting data with prepared statements, execution step

```
conn:execPrepared('safe_entry',
    gui.entry.firstname:GetString(),
    gui.entry.lastname:GetString(),
    gui.entry.town:GetString())
```

Using parametrized queries

```
conn:execParams([[
INSERT INTO person (firstname, lastname, town)
VALUES ($1, $2, $3)
]], gui.entry.firstname:GetString(),
    gui.entry.lastname:GetString(),
    gui.entry.town:GetString())
```

Deny database administrator logins

```
conn = pgsql.connectdb(...)
res = conn:exec('select rolsuper from pg_roles where rolname = current_user')

if res:ntuples() != 1 then
        os.exit(1)
end

if res:getvalue(1, 1) == 't' then
        os.exit(2) -- DB super user
end
```

SQL must be composed carefully

- Whenever SQL is composed, extra care is needed
- Numbers can usually safely be converted
- **ALL** string input must be sanitized
- Even when coming from sources we assume safe at first sight (e.g. barcode scanners, cookies, etc.)
- If you can, prevent DBA logins via the application

PostgreSQL Cluster Anatomy

A PostgreSQL cluster...

- is a collection of databases
- is a collection of roles which can or can not login to the cluster
- has a config file, pg_hba.conf, that defines who can access the database using which authentication methods
- Multiple clusters can safely co-exist on the same machine

What is a PostgreSQL database?

- Resides in a cluster
- All roles in the cluster can connect to the database
- Can contain schemas (the public schema is the default and accessible by all users (roles) by default)
- Is owned by a certain user (role)

What are roles?

- Can be defined to be able to login or not
- Can only access the public schema by default
- Can be granted fine-grained access rights (or have them revoked)
- Login roles can use different methods to authenticate themselves, see pg_hba.conf
- A role's rights can be granted to (or revoked from) another roles (roles become groups)

What are schemas?

- A named collection of database objects like e.g. tables, views, functions etc. that resides itself in a database
- Access objects in a schema with a *schemaname*.objectname prefix or set the path accordingly
- Schemas esemble directories in a filesystems
- To access objects in a schema, a role must have USAGE privilege on the schema **AND** proper access privilege on the object in the schema

Security at the right layer

- Many applications handle security at the application layer, use only one database login
- Software can have bugs. What if the application gets compromised?
- Full access to the application database by the intruder!

Mitigating the effects of an unwanted database access

- From now on, we assume our software has a bug that lets an attacker access the database

Two types of applications

- Applications with real users, e.g. a ledger system or a customer relationship management system
- Each user has his own login role
- Application with anonymous users, e.g. an online ticket booking system
- Roles per user are not always possible/feasible

Security at the database layer

- PostgreSQL has a fine grained security system
- Define security at the database layer
- Define ,,model" roles with security privileges for distinct areas of an application
- GRANT the ,,model role" to the real users
- Don't let a database administrator account log in

The principle of the least power

- Give roles as little privileges as ever possible
- Truly understand what pg_hba.conf can do for your, limit access to the database as close as possible

Partition the database using schemas, model roles

- Define schemas for application areas, extensions etc.
- Create roles that must not login for each part of an application area, e.g. a read-only user, a normal user, and an administrator
- We call such roles ,,model roles" because they serve as a model for the database access privileges
- The model roles can be **GRANTED** to the real (login) roles
- Consider not using the public scheme at all

Use the security mechanism that PostgreSQL allows

- Taylor your model roles with the fine grain security PostgreSQL allows you
- Consider not only the obvious access rights like SELECT, UPDATE, INSERT, DELETE etc. but also concepts like column level security

Database application with anonymous users

- Require a database login role for the application, not the user
- That role should be practically useless for an attacker

Prevent data stealing

- Make it impossible to list the contents of a table, allow only access to the data **THIS** anonymous user has created.
- How should that work?
- Don't grant **SELECT** rights to the application's role
- Use a function to access data, use a random string or cryptographic hash, e.g. stored in a browser cookie, to retrieve the data.
- That function is defined to run with the privileges of the function owner (SECURITY DEFINER) (a model role with slightly more privileges than the application role) and can thus SELECT on the data table
- Don't forget to set the search path

Encrypt if you must

- Very sensity data can be encrypted in the database in way that it only can be retrieved if you know the key and one or more secrets („arcanum")
- Create a secure hash value over the key and secret(s)
- Use that hash to encrypt the data
- Create a second hash over the key
- Use the second hash as the key to database

Mirror user privileges in the application

```
res = conn:exec([[
SELECT groname FROM pg_group
    WHERE (
        SELECT usesysid FROM pg_user
        WHERE usename = current_user
    ) = ANY (grolist)
]]

for n = 1, res:ntuples() do
        -- use role membership to adjust UI
        -- has_role(res:getvalue(n, 1))
end
```

Mirror user privileges in the application

- Giving access to an application area (or module etc.) effectively means to **GANT** the model role
- Removing access means to **REVOKE** the model role
- If designed properly, the user interface will reflect this

Hardening the login even more

- To make sure only authorized users can access an application, techniques like OTP (OATH) can be used, but only at the application level
- This only prevents a login, but not the situation where the database can be access due to a bug

Secure the Application

- Don’t construct SQL in memory when external data **of any kind** is involved
- Use parametrized queries or prepared statements instead
- At the very least, escape all data from external sources (user input, scanner input, etc.)

Secure the Database

- Use schemas to partition an application
- Define model roles with fine grained security settings
- Use per-user login roles
- Grant only those model roles to users that they need

Secure the PostgreSQL Cluster

- Limit super user logins
- Understand how pg_hba.conf works

Source Code

The Lua interface to PostgreSQL
`https://github.com/mbalmer/luapgsql/`

The JSON encoder/decoder for Lua
`https://github.com/mbalmer/luajson/`

(There are more useful modules available at
`https://githup.com/mbalmer/`)

About the Author

After working for Atari Corp. in Switzerland where he was responsible for Unix and Transputer systems, Marc Balmer founded his company micro systems in 1990 which first specialised in real-time operating systems and later Unix. During his studies at the University of Basel, he worked as a part time Unix system administrator.
He led the IT-research department of a large Swiss insurance company and he was a lecturor and member of the board of Hyperwerk, an Institute of the Basel University of Applied Sciences.
Today he fully concentrates on micro systems, which provides custom programming and IT outsourcing services mostly in the Unix environment.
Marc Balmer is an active NetBSD developer; he was chair of the 2005 EuroBSDCon conference that was held at the University of Basel and was a member of the program committe of this conference series many times.
He is one of the main organizer of the Swiss Postgres Conference

In his spare time he likes to travel, to photograph and to take rides on his motorbike. He is a licensed radioamateur with the call sign HB9SSB.

Contact Information

Marc Balmer
micro systems
Landstrasse 66
CH-5073 Gipf-Oberfrick
Switzerland

E-mail	marc@msys.ch, mbalmer@NetBSD.org, m@x.org
Company web	`http://www.msys.ch/`
Personal web	`http://www.vnode.ch/`