# Exploring Capability-based security in software design with Rust

Kenneth Fossen

June 13th, 2022

UNIVERSITY OF BERGEN

# Access Control and Today's Security landscape

# Access Control

- Critical in software design
- Responsible for securing our data
- Four models
- «It is defining how users can interact with data in our applications»
- Privacy

# OWASP

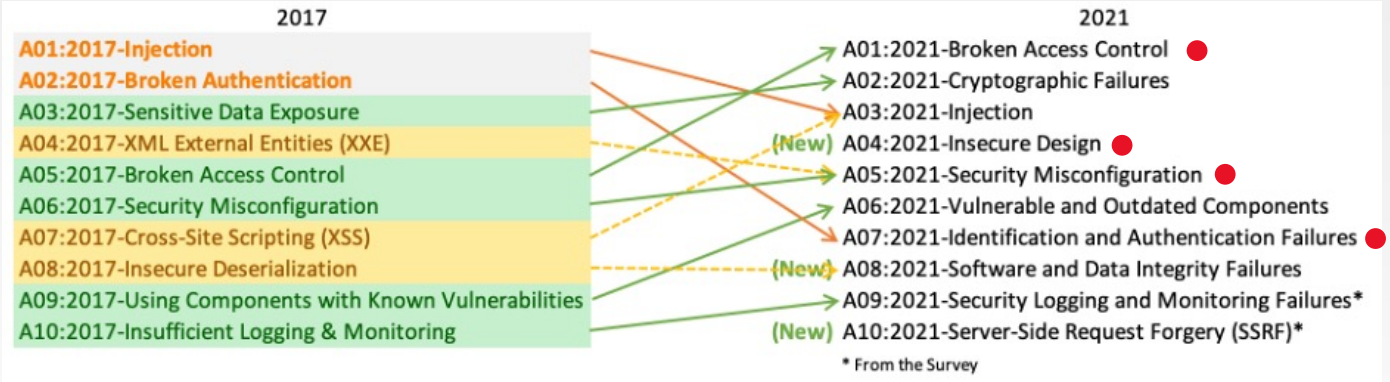
- What is OWASP?
- Their work
- Top10
- Analysis


Image: https://owasp.org/Top10/

# Important Categories

- Insecure Design
- Security Misconfiguration
- Identification and Authentication Failures
- Broken Access Control

# OWASP Top 10 Changes




Source: https://owasp.org/Top10/#whats-changed-in-the-top-10-for-2021

# A01:2021 – Broken Access Control

- Numbers from OWASP:
  - 94% broken access control
- Root causes:
  - Cross-Site Request Forgery «Confused Deputy»
  - Violation of the principle of least privilege - PoLP
  - Elevation of privilege

Source: https://owasp.org/Top10/A01_2021-Broken_Access_Control/

# Confused Deputy

- What is a confused deputy?
- What can mitigate confused deputies?

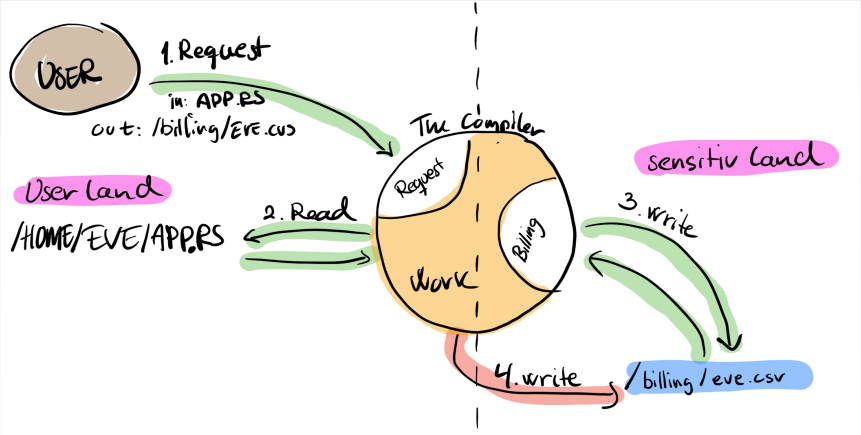
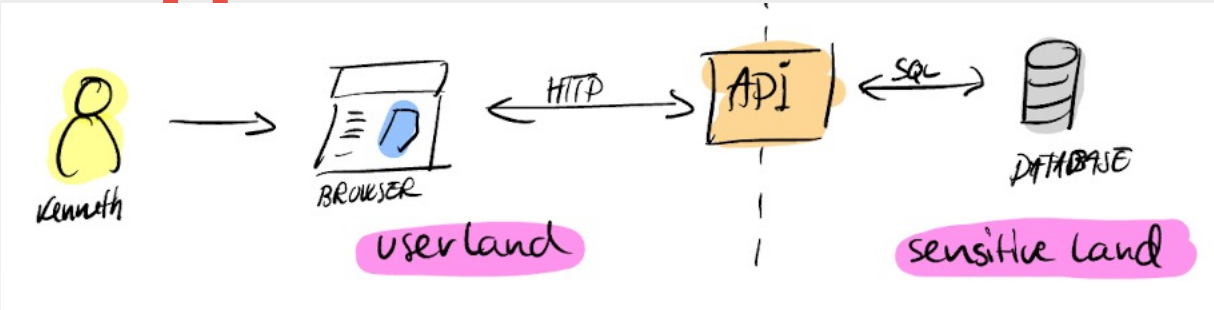
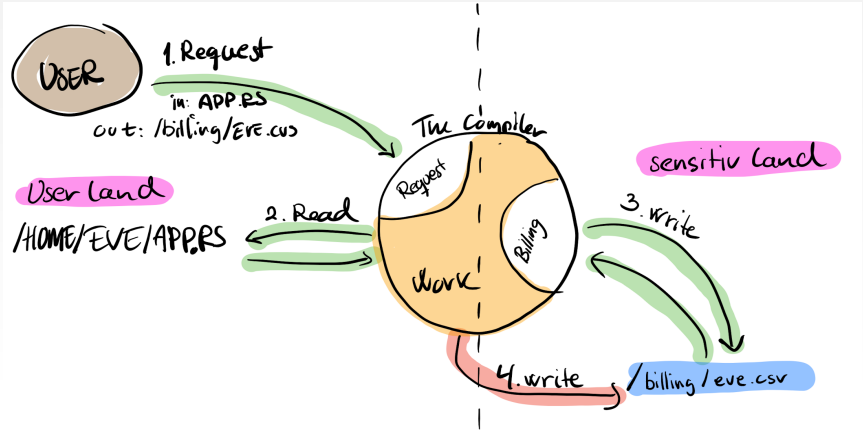



Image: https://commons.wikimedia.org/wiki/File:Don_Knotts_Barney_and_the_bullet_Andy_Griffith_Show.jpg

# Web application architecture




- RESTful API
- CSRF
- PoLP



# Motivation

- OWASP - Broken Access Control
  - 94% of 500 000 applications have some form of broken access control (OWASP)
- Most access control is RBAC or ABAC
- Almost no use of capability-based access control
  - Literature shows little or no use of capabilities in web applications

# Goals

# Goal: Capability Library

Find a software design for the API that gives us capability-based access control

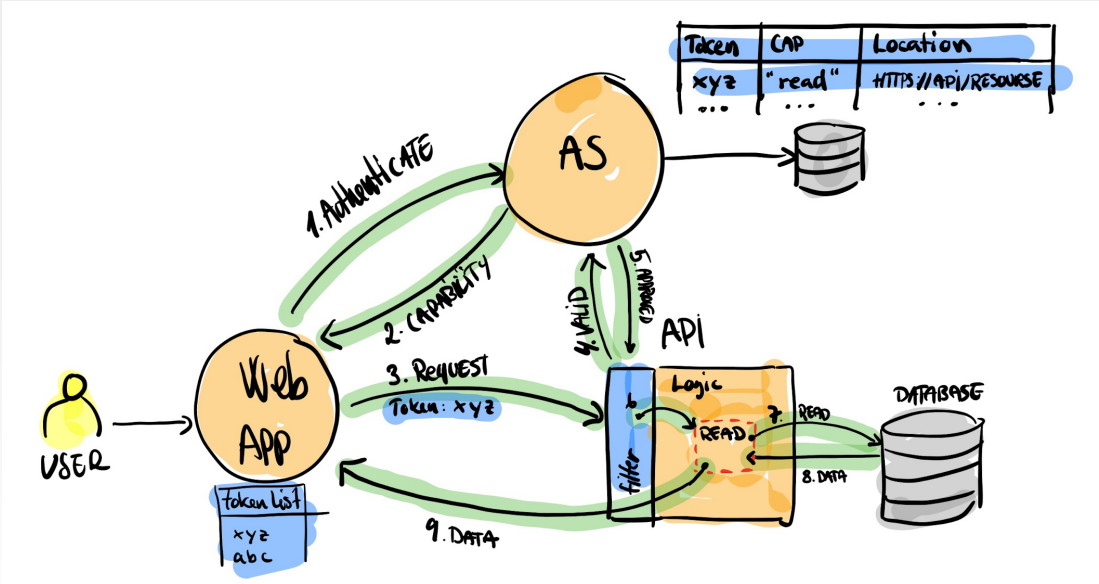
Generalize this to a library

```rust
#[capabilities(Delete, id = "id")]
pub struct Orders {
    id: i32,
    name: String,
}

#[service(SqliteDb, name = "db")]
#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let connection_string = "sqlite::memory:".to_string();
    let _pool = CapService::build(connection_string)
        .await
        .expect("Failed to create database");
    Ok(())
}

#[capability(Delete, Orders)]
fn delete_order(order: Orders) -> Result<(), CapServiceError> {
    let res = sqlx::query!(r#"DELETE FROM orders WHERE id = $1"#, order.id)
        .execute(&self.db)
        .await
        .map_err(CapServiceError);

    Ok(())
}
```

# Goal: Capability System

Use this library together with a authorization server (AS)

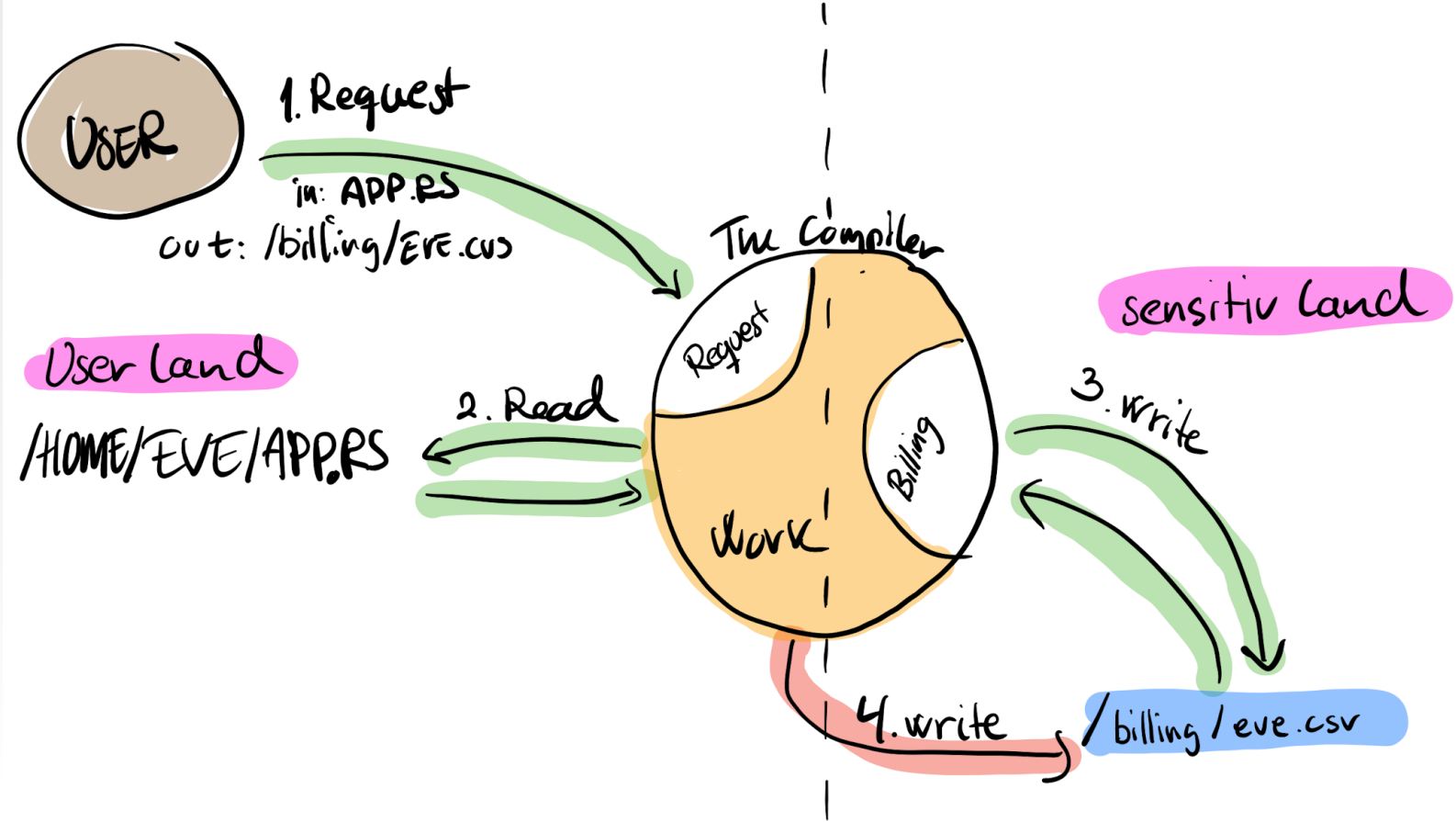
Have a web app using AS to access our API

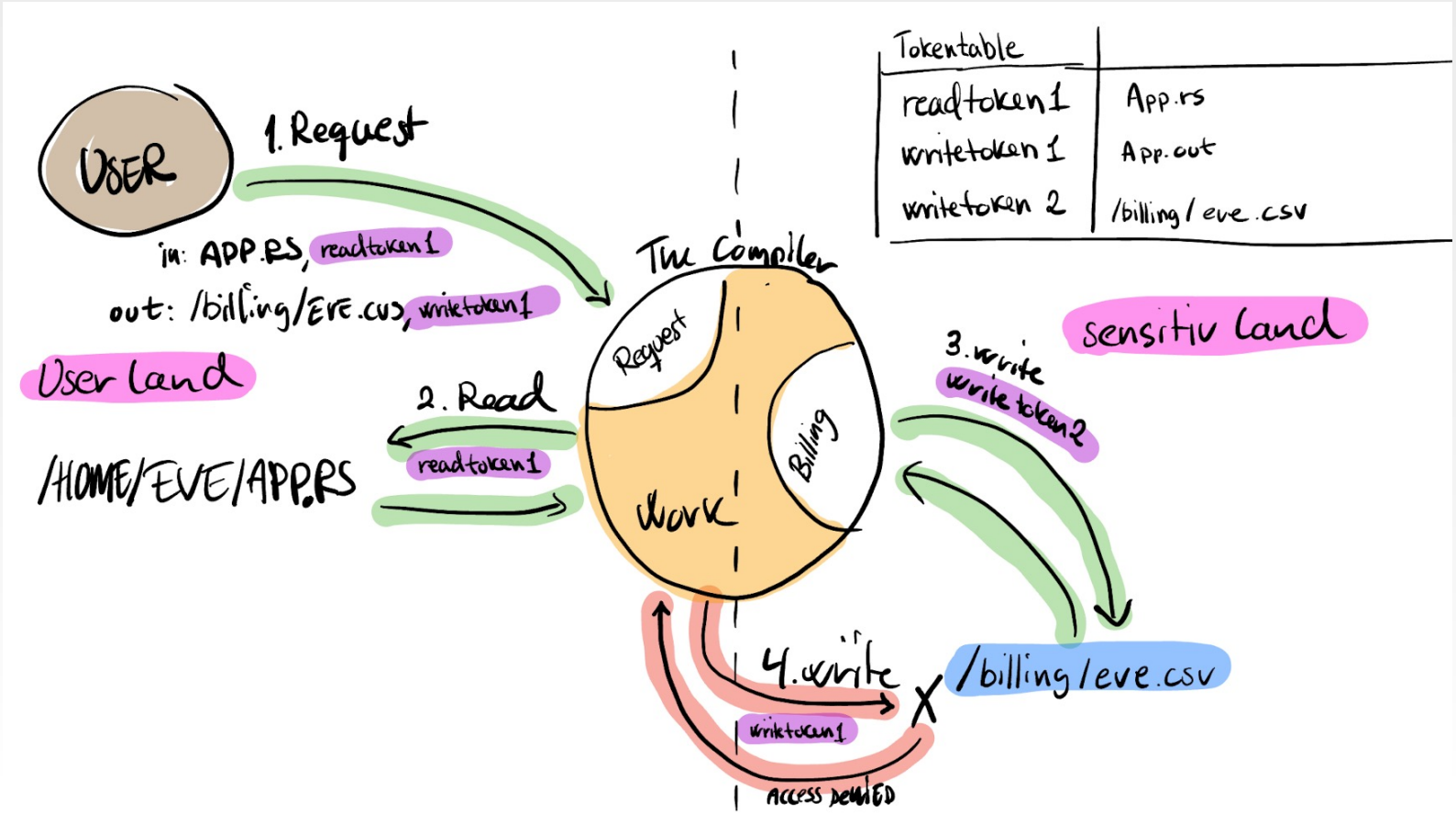


# Confused Deputy & Capabilities

# Confused Deputy

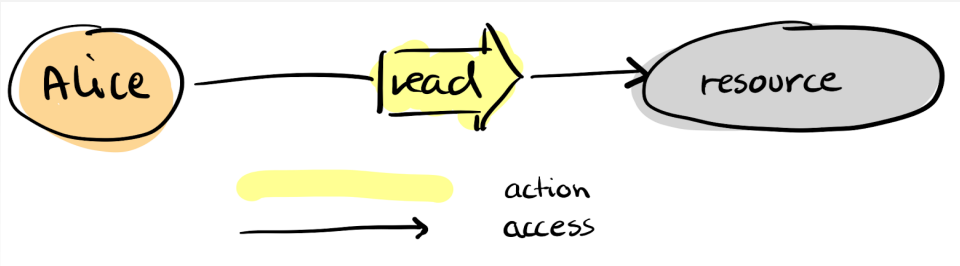
- A privileged process that is tricked
- Principle of Least Privilege (PoLP)
  - Does not solve, can reduce it


USER
1. Request
in: APP.RS
out: /billing/EVE.csv
The Compiler
Request
Billing
Work
User Land
/HOME/EVE/APP.RS
2. Read
3. write
sensitiv Land
4. write
/billing/eve.csv

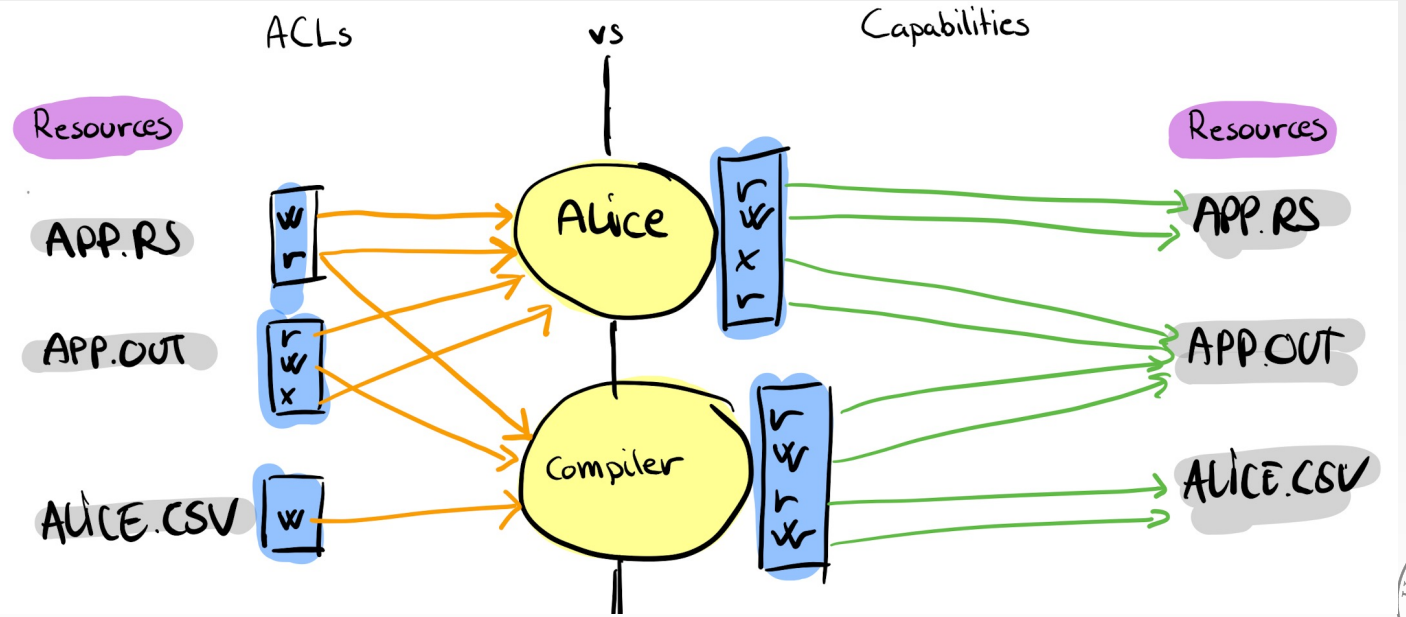

Tokentable
readtoken1
App.rs
writetoken1
App.out
writetoken 2
/billing/eve.csv
USER
1. Request
in: APP.RS, readtoken1
out: /billing/EVE.CSV, writetoken1
User Land
/HOME/EVE/APP.RS
The Compiler
Request
Billing
Work
2. Read
readtoken1
3. write
write token2
sensitiv Land
4. write
writetoken1
ACCESS DENIED
/billing/eve.csv

# Capability Myths

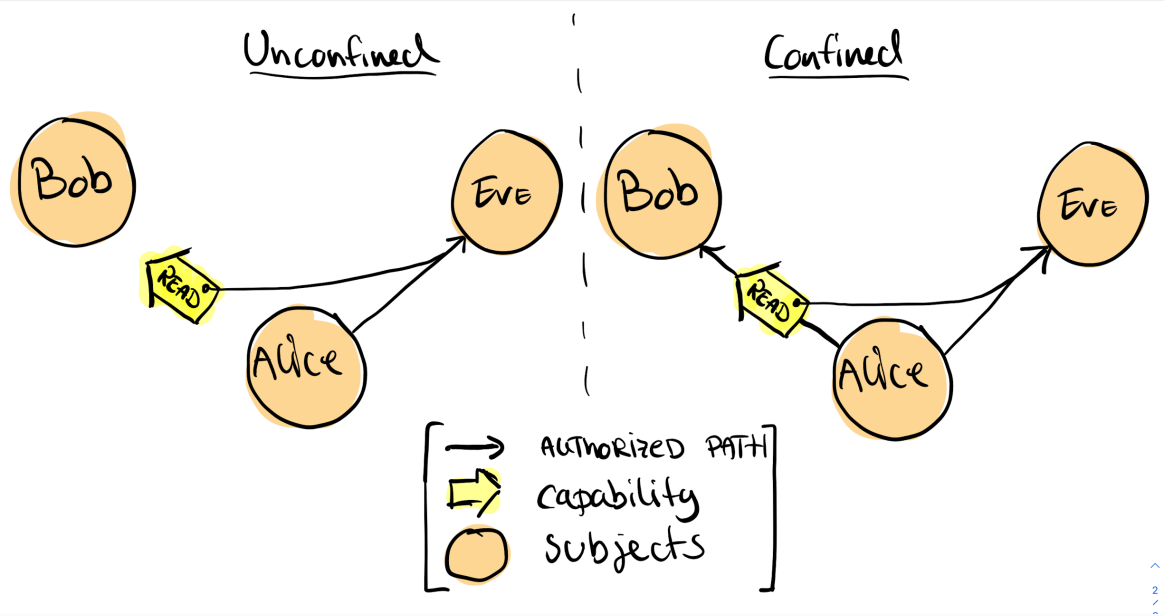
- Why are not capabilities used that much?
- What are capabilities?
  - Keys «Token»
  - Delegate
  - Revokable
  - Composable



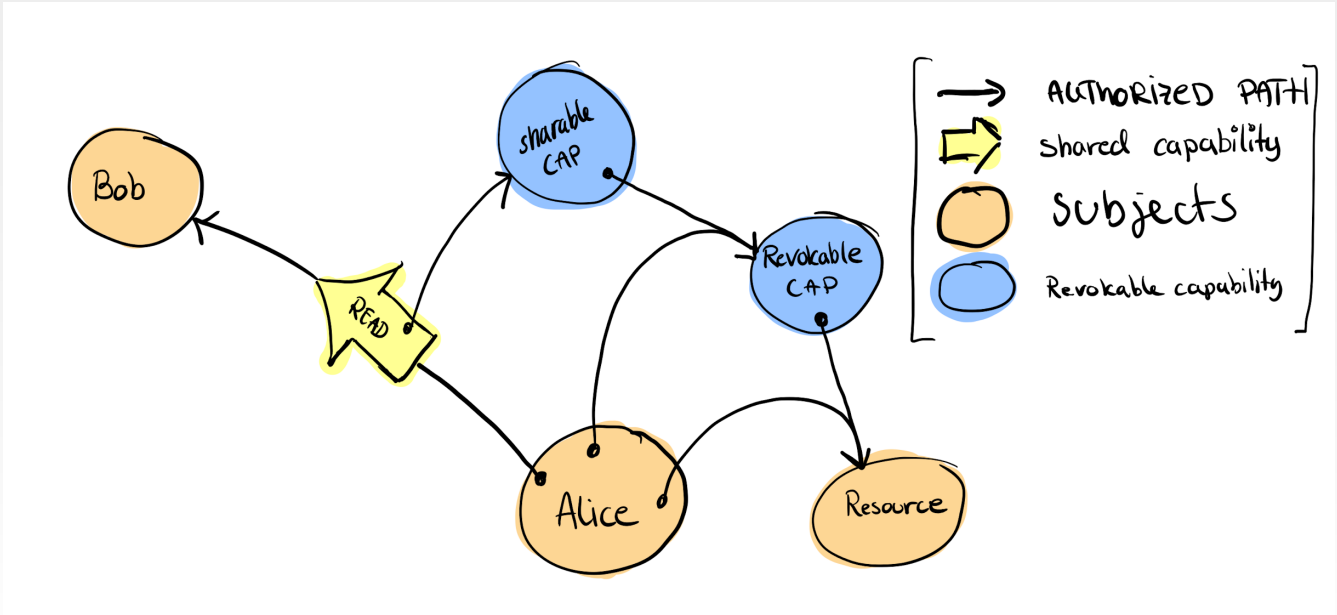
# ACL vs Capabilities



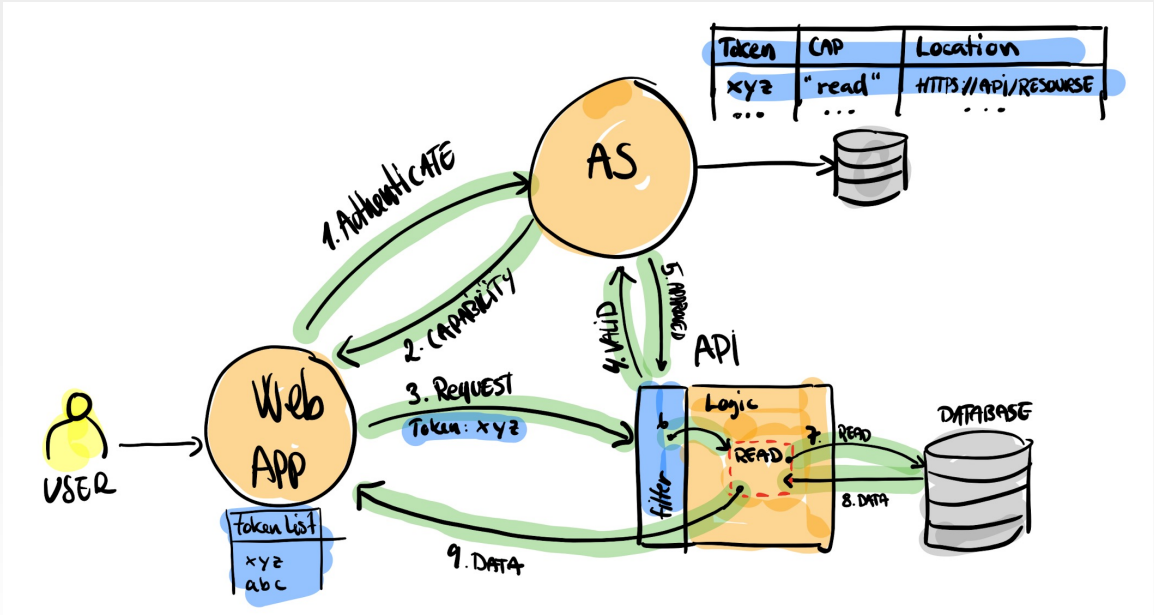
# Confinement Myth



# Irrevocability Myth

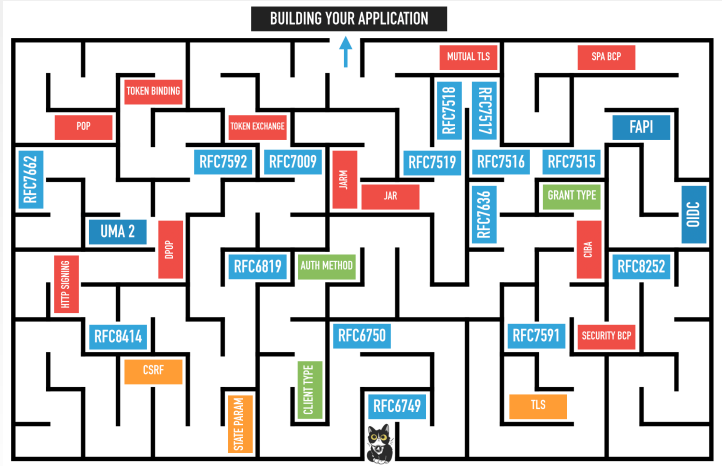


# Capability System



# GNAP instead of OAuth

## OAuth




## GNAP

- draft-ietf-gnap-core-protocol
- draft-ietf-gnap-resource-servers

Source: https://aaronparecki.com/2019/12/12/21/its-time-for-oauth-2-dot-1

# Experiments

# Experiments 1a-c

- Generic Type Parameters and Trait Bounds
- Capability field inspired from TypeState
- Using macros in Rust

# Example: CanChangeAndDeleteUserData

```
capability!(CanChangeAndDeleteUserData for SQLite,
            composing { Save<User>,   User, DBError },
                      { Update<User>, User,   DBError },
                      { Delete<User>, (),   DBError });

trait CanChangeAndDeleteUserData:
    Capability<Save<User>, Data = User, Error = DBError>
    + Capability<Update<User>, Data = User, Error = DBError>
    + Capability<Delete<User>, Data = (), Error = DBError {}

impl CanChangeAndDeleteUserData for SQLite {}
```

# Experiments 2a-b

- Creating a library that eases the developers work
- Putting it together to a capability system

# Library Use Example

```rust
#[capabilities(Delete, id = "id")]
pub struct Orders {
    id: i32,
    name: String,
}

#[service(SqliteDb, name = "db")]
#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let connection_string = "sqlite::memory:".to_string();
    let _pool = CapService::build(connection_string)
        .await
        .expect("Failed to create database");
    Ok(())
}

#[capability(Delete, Orders)]
fn delete_order(order: Orders) -> Result<(), CapServiceError> {
    let res = sqlx::query!(r#"DELETE FROM orders WHERE id = $1"#, order.id)
        .execute(&self.db)
        .await
        .map_err(CapServiceError);

    Ok(())
}
```

# Library: Code generation

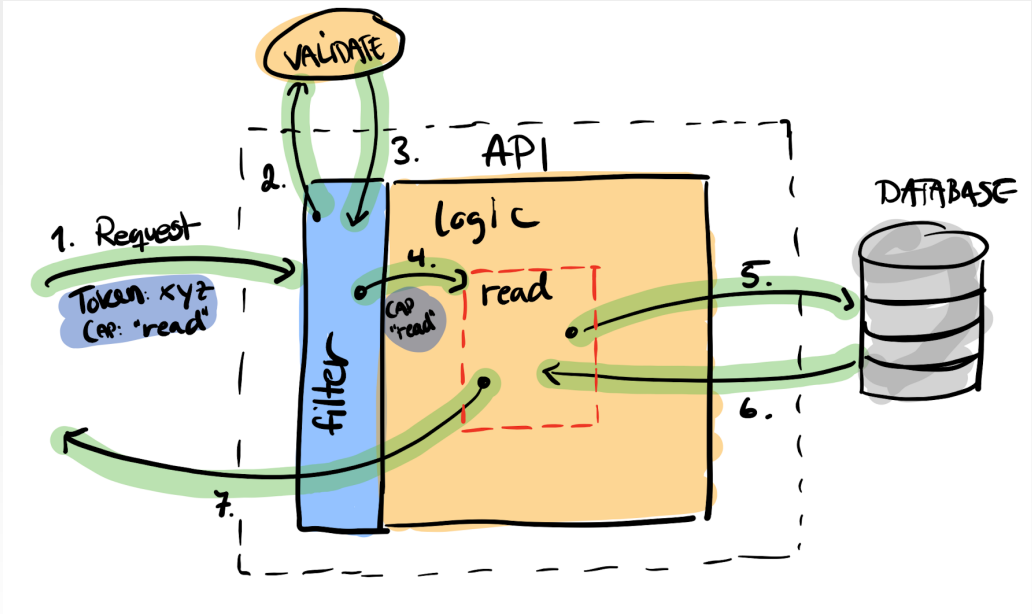
```rust
#[capability(Create, Bowl)]
pub fn create_db_bowl(bowl: Bowl) -> Result<Bowl, CapServiceError> {
    let _res = sqlx::query!(r#"INSERT INTO bowls (name) VALUES ($1)"#, bowl.name)
        .execute(&self.db)
        .await
        .expect("unable to create bowl");
    let b = sqlx::query_as!(Bowl, r#"SELECT * FROM bowls WHERE name = $1"#, bowl.name)
        .fetch_one(&self.db)
        .await
        .expect("Didn't find any bowls");
    Ok(b)
}
```

```rust
pub async fn create_db_bowl<Service>(
    service: &Service,
    param: Bowl,
    cap: ::capabilities::Capability,
) -> Result<Bowl, CapServiceError>
where
    Service: CapCreateBowl,
{
    let valid = ::capabilities::Create { data: param };
    if valid.into_enum().eq(&cap) {
        service.perform(valid).await
    } else {
        Err(CapServiceError)
    }
}
impl CapabilityTrait<Create<Bowl>> for CapService {
    type Data = Bowl;
    type Error = CapServiceError;
    fn perform<'life0, 'async_trait>(&'life0 self, action: Create<Bowl>,
    ) -> ::core::pin::Pin<
        Box<dyn ::core::future::Future<Output = Result<Self::Data, Self::Error>>
                + ::core::marker::Send
                + 'async_trait,>,>
    where 'life0: 'async_trait, Self: 'async_trait,
    {
        Box::pin(async move {
            if let ::core::option::Option::Some(__ret) =
                ::core::option::Option::None::<Result<Self::Data, Self::Error>>
            {
                return __ret;
            }
            let __self = self;
            let action = action;
            let __ret: Result<Self::Data, Self::Error> = {
                let bowl: Bowl = action.data;
                {
                    // injected function block specified by the developer
                }
            };
            #[allow(unreachable_code)]
            __ret
        })
    }
}
```

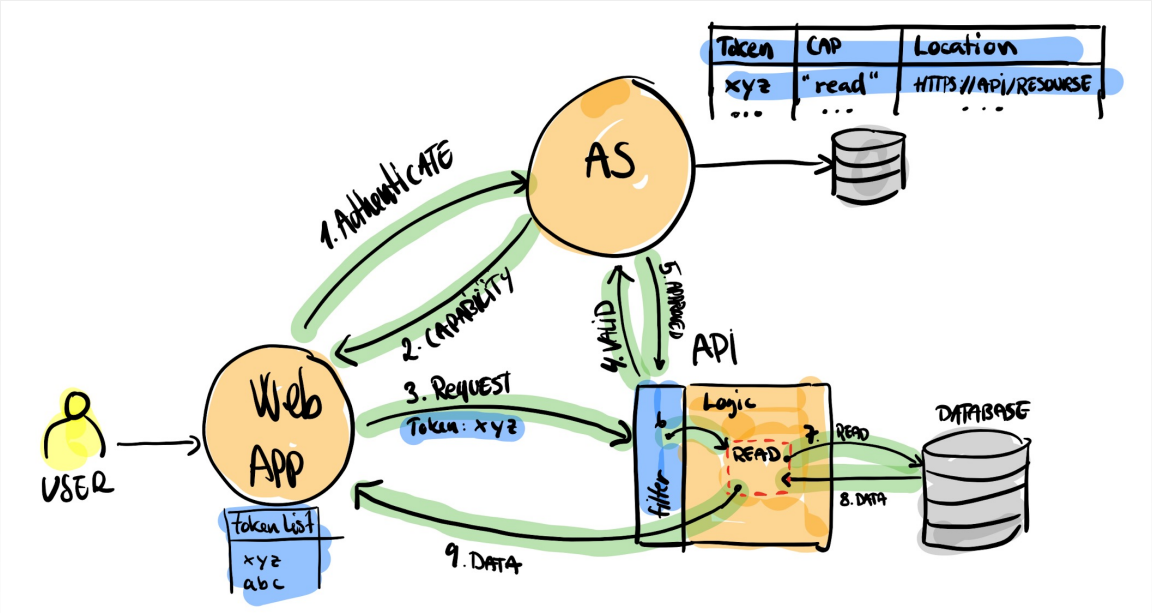
# API Endpoint

```rust
#[post("/bowls/")]
pub async fn create_new_bowl(
    json: web::Json<BowlsDTO>,
    svc: web::Data<CapService>,
    cap: Capability,
) -> impl Responder {
    let svc = svc.get_ref();
    let newbowl: Bowl = Bowl {
        id: 0,
        name: json.name.to_owned(),
    };

    println!("{:#?}", newbowl);
    println!("Cap: {:#?}", cap);
    match create_db_bowl(svc, newbowl, cap).await {
        Ok(bowl) => HttpResponse::Ok().json(bowl),
        _ => HttpResponse::BadRequest().json("{ \"request\": \"bad request\" }"),
    }
}
```

# How it works


VALIDATE
2.
3.
API
logic
1. Request
Token: xyz
Cap: "read"
filter
4.
cap "read"
read
5.
DATABASE
6.
7.

# Capability System



# Demo

# Contribution

# Results

- Built the basis of a capability-based system
- Created a library for developer
- The confinement and delegation problem is not solved
- Database


uib.no