Designing a Zero Trust based solution using DDD, Event Storming, CEDAR and Permguard

In this paper, we aim to explore how the Zero Trust model can be applied to software design and development. We believe that Zero Trust aligns well with Domain-Driven Design (DDD), Event Storming, and microservices-based architectures.

What is Zero Trust?

Zero Trust is a security model based on the principle that no user or system, whether inside or outside the organization's network, should be trusted by default. Instead, every request for access to resources must be continuously verified, regardless of where it originates. The key principles of Zero Trust are:

Never trust, always verify: Never trust any interaction implicitly. Always verify the identity and context of users, services, and software processes before granting access or executing operations within the software environment.

Least privilege access: Apply the principle of least privilege to software flows, ensuring that users, services, and processes can access only the specific resources and actions required for their role. Access privileges should always be contextual, dynamically adapting to the current state of the system and the identity of the requester at the moment of the interaction. Privileges must also be time-bound, ensuring they expire or are revoked as soon as they are no longer needed.

Assume breach: Design software systems with the assumption that a breach can occur at any time. Implement safeguards to limit potential damage and prevent unauthorized movement between components or workflows. Ensure strong segmentation and employ continuous monitoring to quickly detect and isolate suspicious activities. Additionally, implement robust auditing mechanisms to track and verify the true identity behind every action, even in cases of delegation or impersonation, ensuring accountability and transparency.

In software systems, access must be governed by strict policies based on the requester's identity, role, and permissions. This ensures that access is both appropriate and secure. Security in software systems is therefore focused on two key components:

2. **Resources**: Identifying and defining what needs to be protected within the system.

3. **Actions**: Controlling and managing which actions can be performed on these resources, and ensuring that each action is authorized according to the requester's identity and permissions.

This layered approach ensures that both the infrastructure and software components collaborate effectively, providing comprehensive and robust security across the system

This authorization model must be carefully crafted as part of the broader system architecture. Its design should align with the same principles used in modeling business domains. But what exactly do we mean by resources, actions, and business domains? These terms may sound familiar because, when abstracted, they map directly to entities in the system. This alignment mirrors the principles of Domain-Driven Design (DDD), a method traditionally used for modeling complex business domains. Interestingly, DDD can also be applied to software security.

By modeling resources and actions within the domain, DDD provides a structured framework for integrating security requirements directly into the system's design. This approach ensures that access controls, authorization rules, and security policies become an integral part of the core system logic, embedding security within the architecture itself rather than treating it as an afterthought.

Key Takeaways:

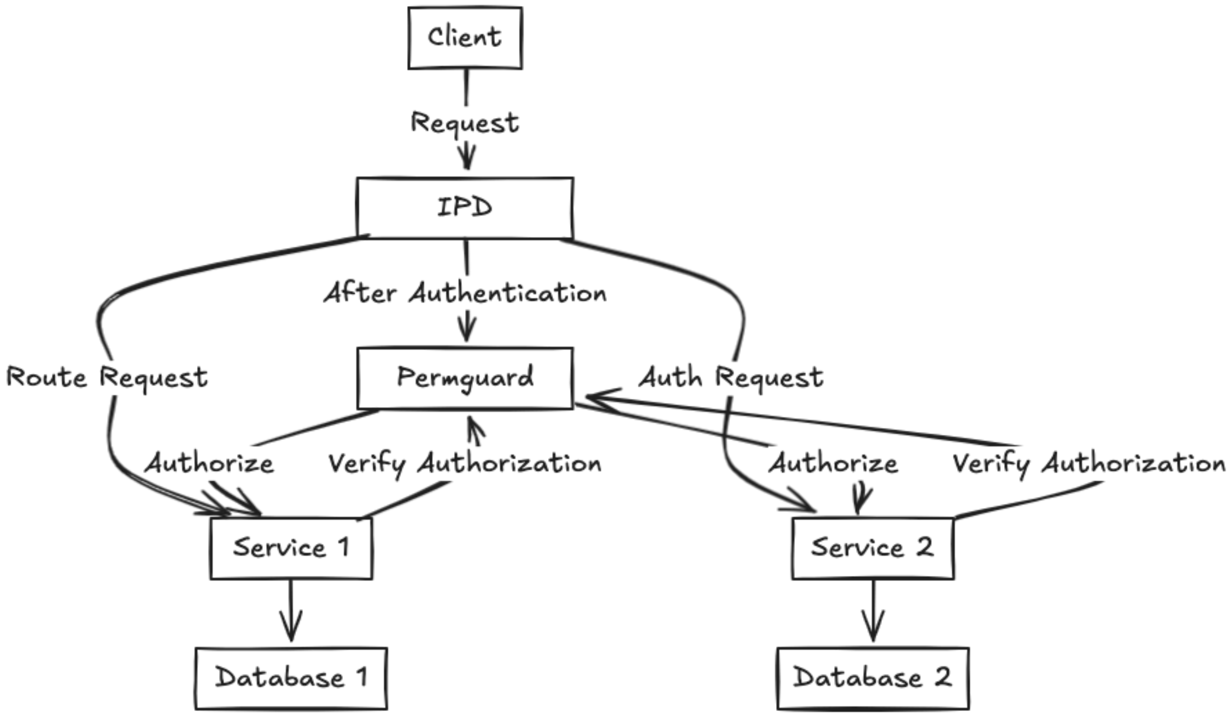
- **Tight Coupling of Resources and Actions**: Resources and actions are inherently tied to the system's architecture, and only the system architect fully grasps their structure. This tight coupling can introduce challenges when scaling or modifying the system.

- **Request Context and External Dependencies**: To extract the necessary information for authorization, you may need to build a request context, which could require access to external systems. These external dependencies can introduce delays, especially when querying external services or databases.

- **Performance Impact**: When external systems are involved in authorization checks, high latency or frequent requests can lead to significant performance degradation. This can become a bottleneck, particularly in large-scale systems that rely on frequent authorization evaluations.

- **1-to-1 Mapping of Resources and Actions**: Directly mapping resources and actions to data resources can result in slow and inefficient systems. This approach quickly becomes impractical as the complexity of real-world systems grows, failing to scale effectively.

- **Strong Coupling Between Policies and Resources**: Tight integration between security policies and system resources creates a complex and fragile environment. This complexity makes the system harder to maintain, slowing down development and making it more difficult to evolve with changing requirements, ultimately leading to increased technical debt.



These principles can be effectively addressed using a **Policy as Code** approach, with technologies like **Permguard**. The **ZTAuth** specification provides a comprehensive solution to tackle these challenges when working on complex software projects. Currently, the best open-source solutions available to address these challenges are **CEDAR**, a language for defining security policies, and **Permguard**, which supports CEDAR as a Policy as Code language, offering robust support for policy enforcement and management.

permguard

Use Case

We will use it as domain sample Loans. Sectors related to the granting of loans, including personal loans, business loans, mortgage loans, and credit cards. We will simplify our domain because in a real world scenario complexity is much bigger.

Examples of Sectors:

- Personal loans (e.g., peer-to-peer loans)
- Mortgage loans
- Business financing
- Microcredit

In a software system that manages a platform for credit and loans (such as personal, business, mortgage loans, and microcredit), authorization features are crucial to ensure that only authorized users can access and interact with certain operations or data. These features must be designed carefully to ensure data protection, regulatory compliance, and the proper functioning of operations. Here is an overview of the main authorization features I would foresee in such a system:

1. Access Management and Authentication

- **User authentication:** A robust authentication system is necessary to ensure that only legitimate users can access the system. This could include:
  - Authentication via username and password.

    - Two-factor authentication (2FA) for higher security, especially for accessing sensitive features such as loan approvals.
    - Single Sign-On (SSO) authentication for business users accessing via centralized systems.
    - Authentication via digital certificates or biometrics (particularly for sensitive operations).
- **Role-Based Access Control (RBAC):** Users must be assigned to specific roles (e.g., administrators, employees, financial advisors, officers, customers), and each role will have specific permissions to access certain areas of the system.

2. Role and Permission Management

- **Administrators:**
    - Have full access to all functionalities, including user management, loan approval and management, report generation, and system configuration.
    - Can view and modify all customer financial data.
- **Analysts or Financial Advisors:**
    - Can access information related to credit assessments and loans and can approve them.
    - Can view and analyze financial data but cannot modify loan statuses.
- **Customers (Borrowers and Lenders):**
    - Can only access information related to their own loan (loan status, interest rates, due dates) and perform operations such as making payments or requesting changes to loans (e.g., loan increase requests).
    - Cannot view or modify other customers' data.

3. Granular Access Control
Each resource or module in the system (such as financial data, loans, customer requests) should have an access control list defining who can view, modify, approve, delete, or create specific resources.

Example: An administrator may have full access to a customer's data, while a financial advisor may only view certain information such as the loan balance and payment status.

Access must be defined for each type of operation:

- **Viewing data:** Read-only permissions for users who only need to view data.
- **Creating or modifying data:** Permissions for users who can create, modify, or approve loans or modify customer information.
- **Deleting data:** Permissions for users who need to delete or archive data (this should be highly restricted).

4. Loan and Request Management Permissions

- **Loan approval requests:**
  - Only certain administrative or advisory roles can approve or reject loan requests.
  - Permissions should be based on a risk assessment, such as the credit score, repayment ability, etc.
  - Approvers should be able to see necessary information (e.g., credit score reports) but should not have access to unrelated sensitive data.

5. Monitoring and Auditing

- **Tracking actions:** The system must log every user action, especially those related to loan management, financial data modifications, and payment operations.
- **Access and action logs:** All activities must be monitored and stored in an audit log that can be used for control and analysis.
- **Security alerts:** Alerts must be sent to administrators in case of suspicious access, failed login attempts, or unauthorized modifications.

6. Legal and Regulatory Compliance

- **KYC (Know Your Customer) and AML (Anti-Money Laundering) checks:**
  - The system should include features for customer identity verification (KYC) and reporting of suspicious activities (AML).

    - These features should be accessible only to specific roles (e.g., compliance officers or administrators) and must be implemented before approving a loan.

- **Privacy management:** Access to personal and sensitive data should be managed with data minimization principles, so that only users with justified needs can view sensitive data such as credit history and banking details.

A software system for managing credit and loans must implement a robust authorization model that provides detailed and secure access management for sensitive data and financial operations. The authorization features must ensure that only users with the appropriate access level can perform specific operations, ensuring legal compliance, data security, and operational integrity.

Domain-Driven Design (DDD) Document for Credit and Loans Management System

1. Introduction

This document outlines the Domain-Driven Design (DDD) approach for the development of a software system managing credit and loan operations, including personal loans, business loans, mortgage loans, and microcredit. It specifies the domain model, aggregates, entities, value objects, and services required to design the system effectively. The system should ensure secure access to sensitive data, comply with regulatory requirements, and provide role-based authorization for users.

---

2. Domain Model Overview

Bounded Contexts:

- **Loan Management Context**: Handles the management of loan applications, approvals, and repayments. It includes features such as loan types (personal, business, mortgage), loan term modifications, and payment schedules.

- **User and Access Management Context**: Manages user authentication, roles, and permissions. Includes features such as login, authentication (via username, password, 2FA), role-based access, and secure data management.

- **Loan Request Context**: Handles regulatory features like KYC (Know Your Customer) and AML (Anti-Money Laundering). Manages customer identity verification and suspicious activity reporting.

---

3. Key Domain Entities

1. **Loan**
   - **Attributes**: Loan ID, loan type (personal, business, mortgage, microcredit), loan amount, interest rate, term, borrower (Customer), status (approved, pending, rejected).
   - **Methods**:
     - `calculateInterest()`: Calculate loan interest based on type and loan term.
     - `updateLoanStatus()`: Update the loan status (approved, rejected).
     - `adjustTerms()`: Modify loan conditions (amount, interest rate).
2. **Customer**
   - **Attributes**: Customer ID, name, contact information, financial status, credit score.
   - **Methods**:
     - `submitLoanRequest()`: Request a loan.
     - `updatePersonalInformation()`: Update contact or financial information.
     - `makePayment()`: Make payment for a loan.
3. **Payment**
   - **Attributes**: Payment ID, payment amount, payment date, loan ID, payment method.
   - **Methods**:

        - `processPayment()`: Process a payment against the loan balance.

4. **User**
    - **Attributes**: User ID, username, password, role, last login.
    - **Methods**:
        - `authenticate()`: Authenticate user.
        - `assignRole()`: Assign roles and permissions to users.
5. **Role**
    - **Attributes**: Role ID, role name (e.g., Administrator, Financial Advisor, Customer).
    - **Methods**:
        - `assignPermission()`: Assign specific permissions to a role.
        - `modifyPermissions()`: Modify role permissions.
6. **Compliance**
    - **Attributes**: Customer ID, KYC status, AML flag, verification documents.
    - **Methods**:
        - `verifyCustomerIdentity()`: Verify customer identity (KYC).
        - `flagSuspiciousActivity()`: Report potential AML concerns.

---

4. Value Objects

1. **LoanTerms**
    - **Attributes**: Interest rate, loan duration, repayment schedule.
    - **Methods**:

        - `calculateMonthlyPayment()`: Calculate the monthly repayment amount based on loan amount, interest, and term.

2. **PaymentInformation**
    - **Attributes**: Payment method, bank details, payment amount.
    - **Methods**:
        - `validatePaymentMethod()`: Ensure that the payment method is valid and authorized.

---

5. Aggregates

1. **LoanAggregate**
    - Contains: Loan, Customer, Payment, LoanTerms.
    - Responsibilities:
        - Manage the lifecycle of a loan, including creation, approval, modification, and repayment.
        - Ensure that payments are processed correctly and adjust loan terms if required.
        - Keep track of the loan status (approved, pending, or rejected).
2. **UserAggregate**
    - Contains: User, Role, Permissions.
    - Responsibilities:
        - Manage user authentication, roles, and permissions.
        - Assign appropriate roles and ensure that users can access only the resources they are authorized for.
3. **ComplianceAggregate**

- Contains: Customer, Compliance (KYC, AML).
- Responsibilities:
    - Ensure that the customer's KYC status is verified before approving loans.
    - Flag any suspicious activities based on the AML guidelines.

---

6. Services

1. **LoanService**
    - **Methods**:
        - `applyForLoan()`: Allow a customer to apply for a loan.
        - `approveLoan()`: Approve or reject a loan application based on creditworthiness.
        - `modifyLoan()`: Modify loan terms after approval.
2. **PaymentService**
    - **Methods**:
        - `processPayment()`: Process a customer's loan payment.
        - `adjustBalance()`: Adjust the loan balance after a successful payment.
3. **AuthenticationService**
    - **Methods**:
        - `authenticateUser()`: Authenticate users based on their credentials.
        - `perform2FA()`: Perform two-factor authentication for secure access.
4. **ComplianceService**

- **Methods**:
    - `verifyKYC()`: Verify the customer's KYC data.
    - `reportAML()`: Report any suspicious financial activity for AML purposes.

5. **ReportingService**
    - **Methods**:
        - `generateLoanReport()`: Generate reports related to loan status, customer creditworthiness, and financial metrics.
        - `generateComplianceReport()`: Generate reports for regulatory compliance, including KYC and AML statuses.

---

7. Repositories

1. **LoanRepository**
    - **Methods**:
        - `findByLoanId()`: Retrieve loan details by loan ID.
        - `save()`: Persist loan data.
        - `update()`: Update loan status or terms.
2. **UserRepository**
    - **Methods**:
        - `findByUserId()`: Retrieve user data by user ID.
        - `save()`: Persist user data.
        - `update()`: Update user roles and permissions.
3. **ComplianceRepository**

        - **Methods**:
            - `findByCustomerId()`: Retrieve compliance information (KYC/AML status).
            - `save()`: Persist compliance verification data.
4. **PaymentRepository**
    - **Methods**:
        - `findByPaymentId()`: Retrieve payment transaction details.
        - `save()`: Persist payment records.

---

8. Domain Events

1. **LoanApprovedEvent**
    - Triggered when a loan is approved, notifying the system that the loan status has changed to "approved."
2. **PaymentProcessedEvent**
    - Triggered after a payment is processed, updating the loan balance.
3. **UserAuthenticatedEvent**
    - Triggered when a user is successfully authenticated.
4. **KYCVerifiedEvent**
    - Triggered when a customer's KYC verification is completed successfully.

---

9. Authorization and Security

- **Role-Policy-Based Access Control (PBAC)**: Users will be assigned specific roles that define their access to different areas of the system. Policies will govern each key security checkpoint within the system, where decisions and actions are influenced by both specific commands and system-generated events.

- **Two-Factor Authentication (2FA)**: Sensitive operations, such as loan approval or financial data access, will require 2FA.
- **Access Control Lists (ACL)**: For each resource, an ACL will define what actions (view, modify, approve, delete) can be performed based on user roles.

This design document provides a blueprint for developing a credit and loans management system. The approach focuses on secure access, role-based permissions, financial operations, and regulatory compliance, while leveraging a robust domain model to structure the system efficiently. The design ensures that users interact with the system in a way that aligns with their role, provides proper authorization, and guarantees legal and regulatory compliance.

Domain Event AND Zero Trust

First of all, let's give some definitions.

Domain events are events that represent something that has occurred in the system, typically in the business domain. They are often used in **event-driven architecture**, where actions or changes in the system are communicated by firing events.

Incorporating domain events into **authorization decisions** can provide a more dynamic and context-sensitive way of controlling access, especially in a more complex or distributed system. Here's how domain events could enhance authorization:

- **Dynamic Context**: Domain events can carry rich contextual information that can be used in authorization decisions. For instance, a domain event could signal that a user's role has changed or that a sensitive action has occurred (e.g., a financial transaction). The system could use this event to update the authorization policies dynamically and adjust access controls in real-time.
- **Event-Driven Authorization**: As events like "user created", "user role changed", or "sensitive data accessed" are triggered, they can inform the PDP of changes in the security context. This allows policies to be evaluated more accurately in real-time, ensuring that authorization decisions are based on the most up-to-date information.

Policy Decision Point (PDP)

A **Policy Decision Point (PDP)** is a component in a security architecture that is responsible for making **authorization decisions**. When a user or system attempts to perform an action that requires access to a resource, the PDP evaluates the request against security policies (such as access control lists, roles, or rules) to decide whether to allow or deny the action.

- **Function**: The PDP checks the context (e.g., user roles, requested actions, resource sensitivity, etc.) against the defined security policies and makes a decision on whether access should be granted or denied.

- **Example**: If a user tries to access a file, the PDP will check if the user's role and permissions allow access to that file.

This part of the system could be demanded to a solution as Permguard that already implement completely a Zero Trust based approach for software development named ZAuth*.

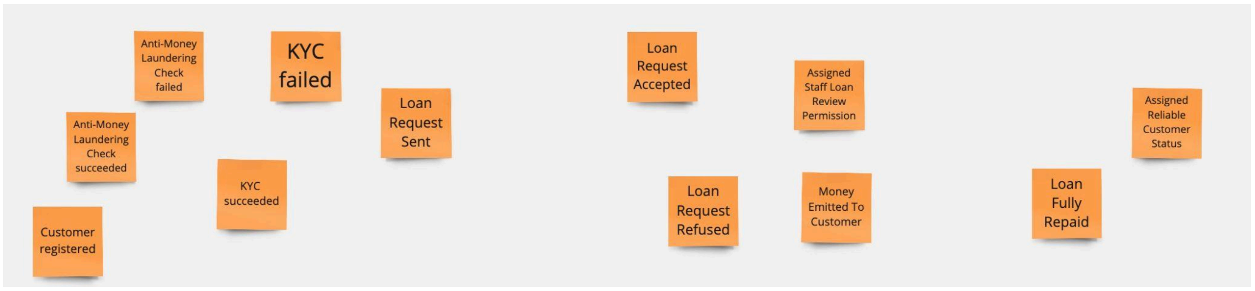
Policy Enforcement Point (PEP)

A **Policy Enforcement Point (PEP)** is a component that **enforces the decision made by the PDP**. It intercepts the action request and either allows or denies the action based on the PDP's decision.

- **Function**: The PEP is the system or service that actually prevents or allows access to resources. It doesn't make decisions but ensures that the policies set by the PDP are actually implemented by restricting or granting access.

- **Example**: In a web application, when a user requests access to a page, the PEP will ensure the user can only access the page if the PDP has authorized it.

This part of an authorization system is usually implemented in code.


Anti-Money Laundering Check failed
KYC failed
Loan Request Sent
Anti-Money Laundering Check succeeded
KYC succeeded
Customer registered
Loan Request Accepted
Assigned Staff Loan Review Permission
Loan Request Refused
Money Emitted To Customer
Loan Fully Repaid
Assigned Reliable Customer Status

Incorporating **Zero Trust** with PDP, PEP, and domain events can increase security in the following ways:

- **Continuous Authentication and Authorization**: Zero Trust assumes that trust is never implicit. Therefore, every request or action, even from an internal user or system, must go through authentication and authorization checks. Domain events can trigger re-evaluations of access rights, ensuring that security policies are continuously enforced and updated based on changing conditions.

- **Dynamic Policy Enforcement**: In Zero Trust, policies are not static. Domain events can provide real-time insights that help to dynamically adjust the security posture. For example, if a domain event triggers the detection of an unusual behavior (e.g., accessing resources at an odd hour), the system could tighten security measures and enforce stricter access controls, even requiring re-authentication. We identified also that every command could contain a step where one or more policies should be verified and enforced.

- **Granular Access Control**: Zero Trust architecture requires granular, policy-based access control (PBAC). This means decisions are made based on multiple factors such as the user's identity, the device they are using, the location, the time of access, etc. Domain events can serve as real-time context for these decisions. For instance, an event might signal that a user is attempting to access a critical resource, and the PDP could consider additional factors (such as if the device is trusted) before granting access.

Let's imagine a scenario in a corporate environment where an employee requests access to a financial report stored in a database:

1. **Request Initiation**: The employee makes a request to access a sensitive financial report. This is a command in our Event Storming diagram.

2. **PEP**: The request first hits the Policy Enforcement Point, which intercepts the request and forwards it to the Policy Decision Point for evaluation.

3. **PDP Decision**: The PDP evaluates the request by checking the employee's identity, role, and any domain events related to the user. For example:

    - Is the employee authorized to view financial reports based on their role?

    - Is the user's device considered trusted (for Zero Trust)?

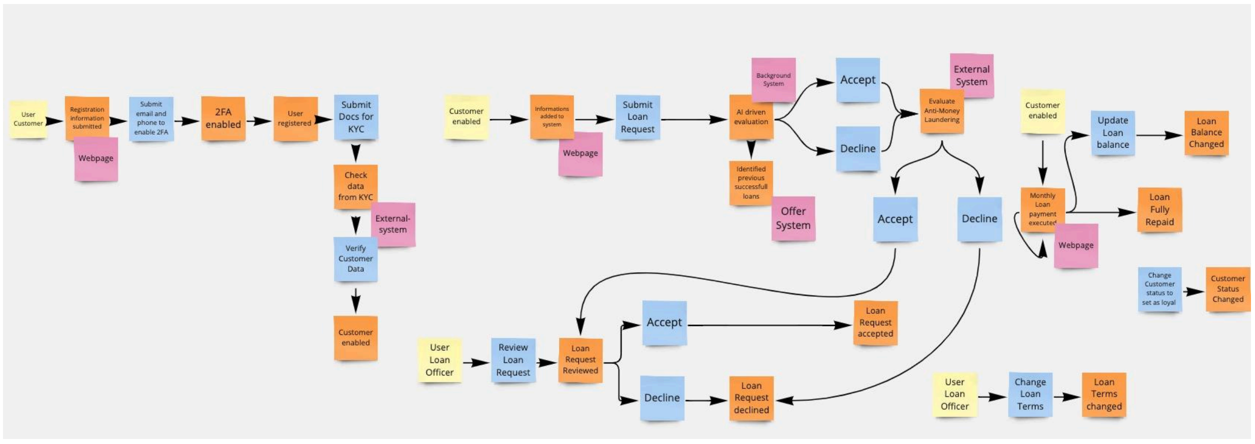
    - Is there an event indicating the employee has recently changed departments, potentially altering their permissions?

4. **Domain Event Handling**: Let's say there's an event that the employee's department has changed. This event could trigger a re-evaluation of their access rights and ensure that

the correct permissions are enforced before granting access.

5. **Authorization and Enforcement**: Based on the PDP's decision, the PEP either grants or denies the request. If denied, the system could initiate an alert or log the event for further monitoring.

6. **Zero Trust Model**: At any point, if any unusual behavior is detected (e.g., the request is coming from an untrusted device or location), the Zero Trust model would trigger additional security measures like multi-factor authentication (MFA) before access is granted.



The **Zero Trust** model is based on the principle of "never trust, always verify." This means that every request (whether internal or external) must be authenticated and authorized before access is granted, regardless of where the request originates. Zero Trust architecture involves continuous validation of users, devices, and actions.

In our vision as said we identified possible security joint authorization points where commands are executed and events happened.

Policies are not static, and can change during application lifetime. Also are influenced by two primary factors:

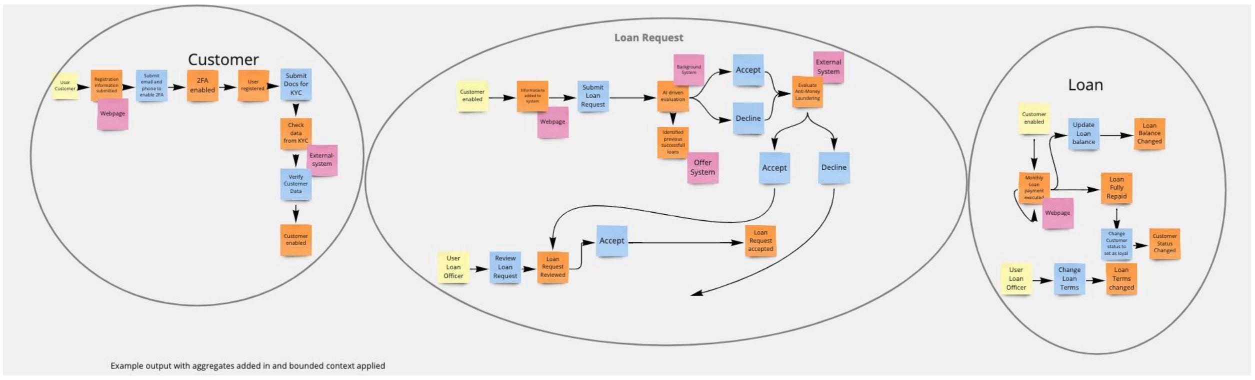
1. **Commands**: These are explicit actions or requests made by users or systems. For instance, a user requesting access to a specific resource or a system triggering an action (such as updating user data or executing a transaction) would be considered a command. Security policies will assess these actions in real-time to ensure that they comply with the required security protocols and authorization rules.

2. **Events**: These are changes or occurrences in the system that may not always be directly tied to user input but are nonetheless critical to security. Events could include things like a user's role being changed, a device accessing the system from an unusual

location, or even an automatic update in the system that triggers a security review. Policies must adapt to these events to continuously monitor and adjust access control decisions based on the evolving state of the system.

We can identify commands and events as potential security authorization checkpoints. At these points, we should invoke **PermGuard** to enforce security policies. This will ensure that commands are only executed by users or applications who have the appropriate permissions to do so. Additionally, it will verify that any event that triggers a computation is a legitimate and authorized message, with the proper permissions to initiate any computational process.

In this way, the security framework dynamically evaluates both user-driven actions (commands) and system-driven occurrences (events), ensuring that policies are enforced comprehensively across the system's operations. This creates a more adaptable and responsive security environment that can enforce rules in real-time, mitigating potential risks and ensuring secure system behavior.



Security as Part of Aggregate Boundaries

In DDD, aggregates enforce **consistency rules** and **transaction boundaries**. When security policies are integrated into the design, they govern what actions can be performed on an aggregate based on the **role** of the user, **context** (such as location or device), and **permissions**. Every security policy affects one or more aggregate.

For example, in the **Loan** aggregate:

- A borrower may only be allowed to update their personal information or make payments on the loan.

- An admin or loan officer might be allowed to approve or reject loan applications, change loan terms, or write off a loan.
- A customer support agent may only have access to view loan details, but they cannot make changes.
- An administrator could assign permissions or roles to system users.

Security policy: Only users with the `LoanOfficer` role can approve loans or modify loan terms.

Impact on aggregates: The aggregate will contain methods that restrict who can perform specific operations on it. For example, only authorized users can approve a loan application, while unauthorized users will be blocked from taking such actions.

2. Authorization Decisions within Aggregates

Security policies are enforced before changes are made to the aggregate. When a user attempts to perform an action (e.g., approving a loan), the system checks whether the user has the necessary permissions.

For example:

- **LoanApplication Aggregate**: A user might try to approve or reject a loan application. The aggregate would have a method like `approveLoan()` that checks whether the user has the correct role (e.g., `LoanOfficer`).
- **Security policy**: A `LoanOfficer` can approve a loan if the loan application is in the "Pending" state. Any user without this role will be denied access to this method.

```
public class LoanApplication

{

    public void ApproveLoan(User user)

    {

        if (user.Role != Role.LoanOfficer)

        {
            throw new UnauthorizedAccessException("Only Loan Officers can
approve loans.");

        }



        // Business logic for approving loan

    }

}
```

In this example, the security policy is embedded in the aggregate method (`ApproveLoan`), ensuring that only authorized users can execute specific actions on the aggregate.

This approach requires rewriting the code whenever there are changes in authorization logic, which means you'll need to recompile and redeploy your application. In the case of a significant security issue, this process could take hours or even days to resolve.

On the other hand, using an external, centralized provider with eventual consistency can significantly enhance scalability, flexibility, and speed. It also improves governance, making it easier to manage security policies and ensure compliance.

In a real-world scenario, it's not enough to just verify the loan officer's user role when approving data. Additional checks are also necessary, such as:

- Passing Anti-Money Laundering (AML) and Fraud Prevention checks
- Ensuring AI-driven risk assessments are passed
- Validating loan terms and conditions
- Confirming eligibility criteria are met
- Completing audit and compliance checks

After modeling the requirements using Event Storming and applying Domain-Driven Design (DDD) practices, it's time to focus on creating our CEDAR schema to define our entities, types, and actions.

A CEDAR schema serves as a valuable tool to assist in policy writing and code validation. It's important to note that, during runtime evaluation, having a CEDAR or CEDAR JSON schema is not required.

Our schema is a simplified version of what you might typically use in a production environment. However, it provides a useful example of how we can model our data effectively.

```
namespace LoanPlatform {

   entity Group;

   entity Customer {

       KYC: Bool,

       emailValidate: Bool,

       phoneNumberValidate: Bool,

       enabled: Bool,

       username: String,

       owner: String

   };

   entity LoanRequest{

       owner: String,

       officer: String

   };

   entity User in [Group]{

       enabled: Bool,

       username: String,
       roles: Set<String>

   };

   entity Document  = {

       isPrivate: Bool,

       owner: User,

       publicAccess: Bool,

   };

   entity AIAgent = {

       identity: String,

   };

   entity Loan = {

       customer: String,

       officer: String,

       viewers: Set<User>,

       editors: Set<User>,

   };

   action Enable, KYCready appliesTo {

       principal: [Customer, User],

       resource: [Customer],

       context: {
           }

   };

   action DeleteGroup, ModifyGroup appliesTo {

       principal: [User],

       resource: [Group],

       context: {}

   };


   action SubmitLoanRequest appliesTo {

       principal: [Customer],

       resource: [LoanRequest],

       context: {}

   };


   action SetLoyalty appliesTo {

       principal: [User],

       resource: [Customer],

       context: {}

   };


   action ValidationAIBased appliesTo {
       principal: [AIAgent],

       resource: [LoanRequest],

       context: {}

   };


   action ChangeLoanRequestStatus appliesTo {

       principal: [User],

       resource: [LoanRequest],

       context: {

           "currentTime": Long,

           "sourceIp": ipaddr

       }

   };


   action Create

       appliesTo {

           principal: [Customer],

           resource: [LoanRequest],

           context: {}

   };
```

```
    action Pay

        appliesTo {

            principal: [Customer],

            resource: [Loan],

            context: {}

    };

}
```

In an organization, each bounded context within a microservices architecture can define its own schema, allowing entities to manage and validate information specific to that context. This is crucial for maintaining clarity and autonomy within the system.

One of the core principles in microservices is autonomy — each service should be independent and capable of evolving without unintended side effects on others. When dealing with large systems, it's important to avoid creating shared, common code that lacks clear ownership. Shared code introduces the risk of creating dependencies between services, which can lead to unforeseen issues during the system's evolution.

For instance, when new attributes are added to one service, other services that are unaware of these changes may encounter broken validation logic or policy violations. This is why it's essential to define clear boundaries and schemas for each bounded context, ensuring that each service remains self-contained and can evolve independently without causing disruptions to the rest of the system.

Also you could properly use the context to improve your policy evaluation code, in Cedar, the context object allows you to pass additional information to your authorization policies at evaluation time. This is useful when you need to make authorization decisions based on dynamic data that isn't part of the principal, action, or resource entities. Context is declared in the when or unless clauses of your Cedar policies:

```
permit (
  principal,
  action == "Action::Update",
  resource
)
when {
  context.time_of_day.hour >= 9 && context.time_of_day.hour < 17
```

};
When calling the authorization API (like IsAuthorized), you pass the context as a JSON object:

```
{
  "time_of_day": {
    "hour": 14,
    "minute": 30
  }
}
```

3. Common Context Use Cases

- **Time-based access: Restrict access to certain hours**
- **Location-based access: Check IP addresses or geolocation**
- **Request metadata: Device type, MFA status, etc.**
- **Temporary attributes: Session-specific data**

4. Context Structure

Context is structured as a map of attributes where:

- **Values can be primitive types (string, bool, long, etc.)**
- **Or nested records (objects)**
- **Or sets of these types**

We prefer to organize our policies by dividing them by aggregate. In our example we identified Customer, Loan Rest and Loan Aggregate. Those are the policies that we could apply when we are managing a Loan Request:

```
@id("submit-loan-request")

permit (

   principal is LoanPlatform::Customer,

   action == LoanPlatform::Action::"SubmitLoanRequest",

   resource is LoanPlatform::LoanRequest

   )

when {

   principal.KYC == true &&

   principal.emailValidate == true &&
   principal.phoneNumberValidate == true &&

   principal.enabled == true

};



@id("submit-loan-request-document-submit")

permit (

   principal is LoanPlatform::Customer,

   action == LoanPlatform::Action::"SubmitLoanRequest",

   resource is LoanPlatform::LoanRequest

)

when {

   principal.KYC == true &&

   principal.emailValidate == true &&

   principal.phoneNumberValidate == true &&

   principal.enabled == true &&

   resource.owner == principal.username

};



@id("loan-request-ai-validate")

permit (

   principal is LoanPlatform::AIAgent,
   action == LoanPlatform::Action::"ValidationAIBased",

   resource is LoanPlatform::LoanRequest

)

when{

   principal.identity == "ai-job"

};


@id("loan-request-change-status")

permit (

   principal is LoanPlatform::User,

   action == LoanPlatform::Action::"ChangeLoanRequestStatus",

   resource is LoanPlatform::LoanRequest

)

when {

   principal.enabled == true &&

   resource.officer == principal.username &&

   (context.sourceIp.isInRange(ip("10.0.0.0/8")) ||

   context.sourceIp.isInRange(ip("172.16.0.0/12"))) &&

   (context.currentTime >= 9 &&

       context.currentTime < 17) &&

       principal.roles.contains("loan-officer")
```

```
};
```

In our architecture we assume you also have Authentication in place. Using ZAuth* and Permguard we saw how really we could improve security in our microservice oriented architecture. You can refer to Permguard SDK and Playground to see how you could integrate CEDAR and use it with the programming language that you prefer.

If you want complete CEDAR example that we analyzed you can find it here:

https://github.com/antrad1978/loan-domain-cedar

More info here:

https://www.permguard.com/

https://www.permguard.com/docs/0.0.x/getting-started/permguard-paradigm/

https://medium.com/@antonio.radesca/building-secure-applications-with-permguard-and-fastapi-a5837dcacbb2

https://medium.com/@antonio.radesca/securing-your-net-core-applications-with-permguard-sdk-for-dot-net-core-6b7a74bd8df4

https://medium.com/@antonio.radesca/enhancing-authorization-in-django-with-permguard-a-zero-trust-approach-b85b813d252d

https://www.permguard.com/docs/0.0.x/getting-started/zero-trust-ready/