40. RBAC Policy DSL Compiler with Static Security & Privilege Escalation Detection

Course Name: Compiler Design

Course Code: CS1202

Name: Pinikeshi Meghana

Roll No: 24CSB0A54

Sec: CSE-A

Document Details: Week 7 Deliverables

Week 6 - Semantic Analysis & Symbol Table Construction

1.Introduction

In Week 7, the focus is on implementing the semantic analysis phase of the RBAC Domain-Specific Language (DSL) compiler. After the lexical and syntax analysis phases ensure that the input follows the correct grammar structure, semantic analysis verifies the logical correctness of the policy.

While the parser checks whether the program is syntactically valid, it does not ensure that references are meaningful or consistent. For example, a policy may follow correct grammar but still contain issues such as undefined roles, duplicate declarations, or circular inheritance between roles. These logical inconsistencies are identified during semantic analysis.

To support this phase, symbol tables are constructed for roles, users, and permissions. The symbol tables act as centralized data structures that store declarations and enable efficient validation of references throughout the policy.

The semantic analyzer performs the following key validations:

- Detection of duplicate role and user declarations
- Identification of undefined role references in user assignments or inheritance
- Detection of circular role inheritance

By implementing semantic analysis and symbol table construction, the RBAC DSL compiler ensures that access control policies are not only syntactically correct but also logically consistent and reliable

2.Symbol Table design

In the semantic analysis phase, symbol tables are constructed to store and manage all declared entities in the RBAC policy. These tables enable efficient lookup, validation of references, and detection of semantic errors such as duplicates and undefined identifiers.

To ensure fast access and simplicity, Python dictionaries are used for symbol table implementation. Dictionaries provide constant-time lookup (O(1)), making them suitable for validating references during semantic checks.

1. Role Symbol Table

The role symbol table stores all declared roles in the policy.

Structure:

```
roles = {
   "Admin": {
      "permissions": ["read", "write"],
      "parent": None
   },
   "Dev": {
      "permissions": ["deploy"],
      "parent": "Admin"
   }
}
```

Design Explanation:

- **Key** → Role name
- **Value** → Dictionary containing:
    - permissions: List of permissions assigned to the role
    - parent: Name of the role it extends (if any)

Purpose:

- Detect duplicate role declarations
- Validate role inheritance (extends)
- Detect circular inheritance
- Verify role references in user assignments

2. User Symbol Table

The user symbol table stores all declared users and their assigned roles.

Structure:

```
users = {
   "Alice": {
      "roles": ["Dev"]
   },
   "Bob": {
      "roles": ["Admin"]
```

```
    }
}
```

Design Explanation:

- **Key** → User name
- **Value** → Dictionary containing:
    - roles: List of roles assigned to the user

Purpose:

- Detect duplicate user declarations
- Validate that assigned roles exist in the role symbol table

3. Permission Handling

Permissions are stored within each role instead of maintaining a separate global permission table. This design aligns with the RBAC model, where permissions are associated with roles rather than directly with users.

Reasoning:

- Simplifies lookup during validation
- Keeps role-related information encapsulated
- Avoids unnecessary global structures

Design Decisions

- Python dictionaries are used for efficient lookup and duplicate detection.
- Symbol tables are constructed before performing semantic validation checks.
- Separation of roles and users into distinct tables improves modularity and clarity.

This structured symbol table design enables systematic semantic validation and ensures logical consistency of the RBAC policy.

3.Semantic Validations

After constructing the symbol tables, the semantic analyzer performs logical validation of the RBAC policy. While the parser ensures syntactic correctness, semantic validation ensures that all declarations and references are meaningful and consistent.

The following validations are implemented:

1. Duplicate Role Detection

Each role declaration is checked before insertion into the role symbol table.

Logic:

- When a new role is encountered, check if its name already exists in the roles dictionary.
- If it exists, raise a semantic error.

Example Input:

role Dev {}
role Dev {}

Expected Output:

[SEMANTIC ERROR] Duplicate role 'Dev'

Purpose:
Prevents redeclaration of roles, which could cause ambiguity in access control policies.

2. Duplicate User Detection

Similar to roles, user declarations are validated before insertion into the user symbol table.

Logic:

- If a user name already exists in the users dictionary, report an error.

Example Input:

user Alice {}
user Alice {}

Expected Output:

[SEMANTIC ERROR] Duplicate user 'Alice'

Purpose:
Ensures unique identification of users in the policy.

3. Undefined Role Reference

Users may be assigned roles, and roles may extend other roles. Each referenced role must exist in the role symbol table.

Logic:

- While processing user role assignments, verify that each assigned role exists.

- While processing inheritance (extends), verify that the parent role exists.

Example 1 – Undefined Role in User:

```
user Bob {
   roles = [Manager]
}
```

If Manager is not declared:

[SEMANTIC ERROR] Undefined role 'Manager'

Example 2 – Undefined Parent Role:

role Dev extends Admin {}

If Admin is not declared:

[SEMANTIC ERROR] Undefined parent role 'Admin'

Purpose:
Prevents invalid references that could break policy logic.

4. Circular Role Inheritance

The system checks whether role inheritance forms a cycle.

Logic:

- Perform a traversal (e.g., Depth-First Search).
- Track visited roles in the inheritance chain.
- If a role is revisited during traversal, a circular dependency exists.

Example Input:

```
role A extends B {}
role B extends A {}
```

Expected Output:

[SEMANTIC ERROR] Circular inheritance detected between roles

Purpose:
Prevents infinite inheritance loops and ensures a valid role hierarchy

4. Algorithm Overview

The semantic analysis phase follows a structured sequence to ensure correctness and consistency of the RBAC policy. The overall workflow is as follows:

- **Step 1:** Receive **AST from Parser**
  The Abstract Syntax Tree (AST) generated during parsing is passed to the semantic analyzer.
- **Step 2: Initialize Symbol Tables**
  Create empty dictionaries for:
    - Roles
    - Users
- **Step 3: Insert Roles into Role Symbol Table**
    - Traverse the AST.
    - For each role declaration:
        - Check if the role name already exists in the role symbol table.
        - If it exists → report duplicate role error.
        - Otherwise, insert the role with its permissions and parent (if any).
- **Step 4: Insert Users into User Symbol Table**
    - For each user declaration:
        - Check if the user name already exists.
        - If it exists → report duplicate user error.
        - Otherwise, insert the user with assigned roles.
- **Step 5: Validate References**
    - For each user:
        - Check if all assigned roles exist in the role symbol table.
    - For each role with inheritance:
        - Check if the parent role exists.
    - If not → report undefined reference error.
- **Step 6: Detect Circular Role Inheritance**
    - For each role:

        - Perform a traversal of its parent chain.
        - Maintain a visited set.
        - If a role is encountered again during traversal → circular dependency detected.
    - Report circular inheritance error.
- **Step 7: Report Errors**
    - If semantic errors are found, display all collected errors.
    - If no errors are found, confirm successful semantic validation.

This systematic process ensures that all semantic constraints are verified after syntax analysis.

5. Design Decisions

Several design decisions were made to ensure efficiency, modularity, and clarity in implementation:

- **Use of Python Dictionaries**
  Symbol tables are implemented using dictionaries to achieve constant-time lookup (O(1)). This allows efficient duplicate detection and reference validation.
- **Separation of Syntax and Semantic Phases**
  Parsing and semantic validation are implemented as separate modules.
    - The parser ensures structural correctness.
    - The semantic analyzer ensures logical correctness.
      This modular design follows standard compiler architecture principles.
- **Error Collection Instead of Immediate Termination**
  Instead of stopping execution at the first semantic error, all errors are collected and displayed together.
  This improves usability and makes debugging easier.
- **Hierarchical Validation Approach**
  Role insertion is performed before user validation to ensure that all role definitions are available during reference checking.
- **Cycle Detection Using Traversal**
  Circular inheritance is detected using a traversal mechanism (e.g., DFS with a visited set), ensuring correctness of the role hierarchy.

These design decisions align with standard compiler construction practices and ensure that the RBAC semantic analysis phase is efficient, modular, and robust.