Product Specification

ROLE-BASED ACCESS CONTROL (RBAC)

Current Version	1.0
File Name	Role-based access control (RBAC)
Requirement unique ID	Sec_Req 2
Responsible / Approver	Richard Ben Aleya
Classification	Public

Document Control

Version	Description	Date	Editor
1.0	Initial Version	26/07/2024	Richard Ben Aleya

Table of contents

1. Introduction

1.1. Introduction

The Vauban project wants the product to implement Role-Based Access Control (RBAC).

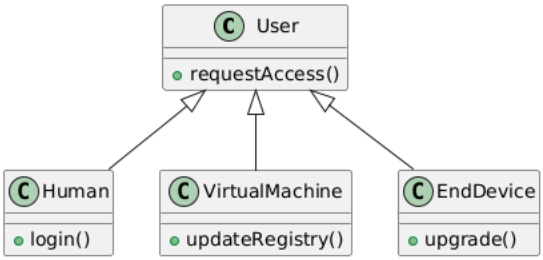
2. RBAC overview

Establish specific roles and permissions to determine who can access the bastion and servers and what actions they are allowed to perform. This ensures that access is granted based on the principle of least privilege, minimizing potential security risks.

3. RBAC key components and definitions

3.1. Users

A user is any entity requesting access. This request can be either proactive or automatic, such as when a user logs in. Additionally, users are not always human; in Role-Based Access Control (RBAC), services and computing entities like virtual machines or end-devices can also be users. For instance, if a device attempts to update its registry contents as part of an upgrade, it is considered a user requiring RBAC-approved privileges.

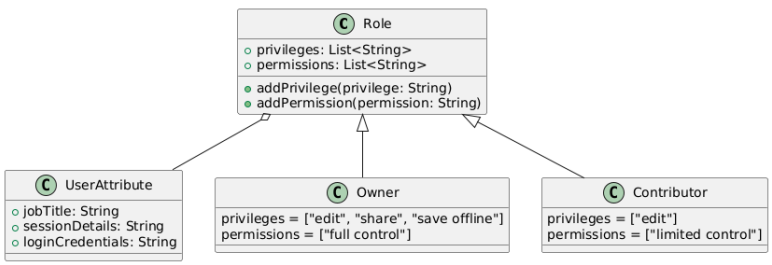



```
@startuml
class User {
    + requestAccess()
}
class Human {
    + login()
}
class VirtualMachine {
    + updateRegistry()
}
class EndDevice {
    + upgrade()
}
User <|-- Human
User <|-- VirtualMachine
User <|-- EndDevice
@enduml
```

3.2. Roles

Roles define which privileges and permissions can be assigned to a user. Roles are often organized in hierarchies, where higher-level roles possess more privileges than lower-level ones. For instance, a document owner might have full control over a document (edit, share, save offline), while a contributor might only be able to edit it. In this case, the owner role is superior to the contributor role.

In RBAC, roles are an aggregation of various user attributes—such as their job title, session details like the device used, and login credentials. RBAC systems often come with predefined roles and support the creation of custom roles.

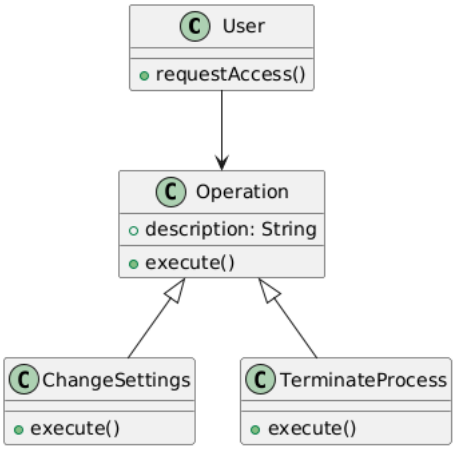



```
@startuml
class Role {
    + privileges: List<String>
    + permissions: List<String>
    + addPrivilege(privilege: String)
    + addPermission(permission: String)
}
class UserAttribute {
    + jobTitle: String
    + sessionDetails: String
    + loginCredentials: String
}
Role <|-- Owner
Role <|-- Contributor
Owner : privileges = ["edit", "share", "save offline"]
Owner : permissions = ["full control"]
Contributor : privileges = ["edit"]
Contributor : permissions = ["limited control"]
Role o-- UserAttribute
@enduml
```

3.3. Operations

Users can request access to perform operations or access objects.

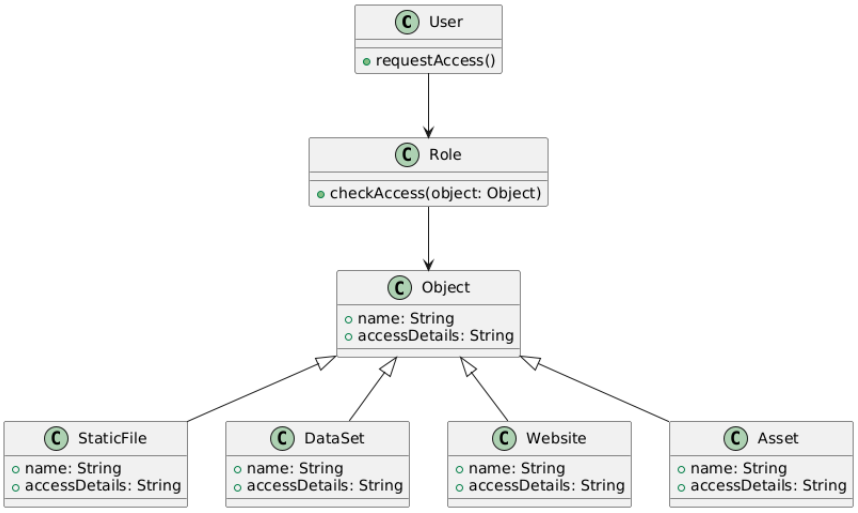
Operations refer to any activity or process within a computing environment. Examples include changing system settings or terminating active processes. Depending on the IT environment, operations can range in complexity and may require robust security measures to protect against potential risks.




```
@startuml
class User {
    + requestAccess()
}
class Operation {
    + description: String
    + execute()
}
class ChangeSettings {
    + execute()
}
class TerminateProcess {
    + execute()
}
User --> Operation
Operation <|-- ChangeSettings
Operation <|-- TerminateProcess
@enduml
```

3.4. Objects

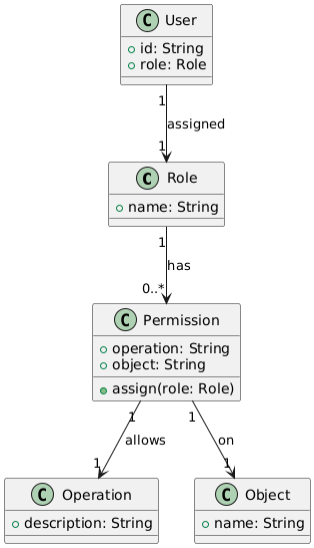
Users can also request access to objects, which can be static files, data sets, websites, or other assets. Unlike operations, accessing an object does not alter the system state, making unauthorized access harder to detect. This is where RBAC is crucial—it ensures the user's role is appropriate for the object and authorizes the access. RBAC also logs access details, including date and time.




```
@startuml
class User {
    + requestAccess()
}
class Role {
    + checkAccess(object: Object)
}
class Object {
    + name: String
    + accessDetails: String
}
class StaticFile {
    + name: String
    + accessDetails: String
}
class DataSet {
    + name: String
    + accessDetails: String
}
class Website {
    + name: String
    + accessDetails: String
}
class Asset {
    + name: String
    + accessDetails: String
}
User --> Role
Role --> Object
Object <|-- StaticFile
Object <|-- DataSet
Object <|-- Website
Object <|-- Asset
@enduml
```

3.5. Permissions

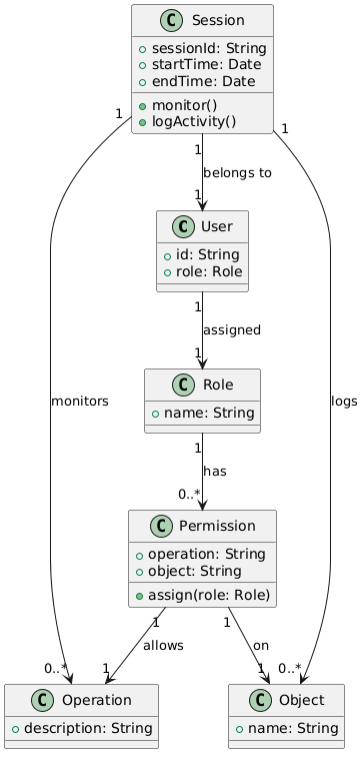
Permissions are fundamental to RBAC, defining what operations and objects a role can access. For example, if an employee with ID A12 has the role of a contributor, permissions specify what actions this role allows. As a contributor, they might be able to edit a document (an object) but not delete it (an operation) or access embedded links (objects). In essence, permissions establish the relationship between a role and the corresponding operations and objects.




```
@startuml
class Role {
    + name: String
}
class Permission {
    + operation: String
    + object: String
    + assign(role: Role)
}
class User {
    + id: String
    + role: Role
}
class Operation {
    + description: String
}
class Object {
    + name: String
}
Role "1" --> "0..*" Permission : has
User "1" --> "1" Role : assigned
Permission "1" --> "1" Operation : allows
Permission "1" --> "1" Object : on
@enduml
```

3.6. Sessions

Sessions refer to the period during which a role interacts with operations and objects. RBAC is activated at the start of a session and remains in effect until it ends. For instance, when a user opens a browser on the company network to access an intranet page, the session begins. The RBAC system verifies the user's role, grants access based on permissions, monitors accessed operations and objects, and logs the session activity until the browser is closed. This entire interaction period is defined as a session.




```
@startuml
class User {
    + id: String
    + role: Role
}
class Role {
    + name: String
}
class Permission {
    + operation: String
    + object: String
    + assign(role: Role)
}
class Operation {
    + description: String
}
class Object {
    + name: String
}
class Session {
    + sessionId: String
    + startTime: Date
    + endTime: Date
    + monitor()
    + logActivity()
}
User "1" --> "1" Role : assigned
Role "1" --> "0..*" Permission : has
Permission "1" --> "1" Operation : allows
Permission "1" --> "1" Object : on
Session "1" --> "1" User : belongs to
Session "1" --> "0..*" Operation : monitors
Session "1" --> "0..*" Object : logs
@enduml
```

4. Conclusion

In conclusion, implementing a bastion host software with Role-Based Access Control (RBAC) involves a comprehensive framework that ensures secure and controlled access to critical resources. The RBAC model incorporates six essential components:

1. **Users**: This encompasses all entities requesting access, whether human, virtual machines, or end-devices. Properly identifying and managing users is crucial for maintaining security and access control integrity.

2. **Roles**: These define the specific permissions and privileges associated with each user, often organized hierarchically. Effective role management allows for streamlined and scalable permission allocation, aligning with user attributes such as job title and session details.

3. **Operations**: These represent the various activities that users can perform within the computing environment. Defining and controlling operations ensure that only authorized activities are executed, safeguarding the system against unauthorized changes.

4. **Objects**: These are the resources users interact with, such as files, data sets, and websites. Access to objects must be carefully controlled and monitored to prevent unauthorized access, which RBAC efficiently manages through role-specific permissions.

5. **Permissions**: These are the core of RBAC, delineating which operations can be performed on which objects by each role. This clear definition prevents unauthorized actions and ensures that users have appropriate access levels based on their roles.

6. **Sessions**: These encapsulate the duration of user interactions with the system, starting from login to logout. By monitoring and logging session activities, the RBAC system can enforce permissions in real-time, ensuring continuous compliance and providing audit trails for security reviews.

Implementing RBAC within bastion host software not only strengthens security by enforcing strict access controls but also enhances manageability by providing a structured and scalable approach to user permissions. This robust framework is vital for protecting sensitive resources and maintaining the integrity of the computing environment.

*** End of specification document ***