# Capability Restrictions

## Introduction

The goal is to enable operators to protect a cluster from usage patterns which may be expensive, suboptimal or potentially detrimental to the health of the cluster.

Operations that may be performed by users will be assigned one or more capabilities which are required for execution. Admins will be able to restrict which capabilities users possess via the existing roles subsystem. For example, the ability to use ALLOW FILTERING in a CQL query may require a specific capability. An admin should be able to use the capabilities framework to ensure that only certain groups of users are permitted to run queries with that clause.

Capabilities may be considered as a partitioning of the privilege to perform some action into a set of distinct sub-privileges.

## Definitions

- Resource: An database object in Cassandra to which users may have access. In the authz subsystem, permissions are granted to Roles in order to perform actions using Resources (i.e. to read from a table, execute a function, modify a role etc). Represented in the codebase by IResource, there are several implementations representing Keyspaces, Tables, Functions, Roles, and JMX Objects.
- Role: A signifier of user identity, or group membership. Permissions are granted to Roles and are inherited when one Role is granted to another Role.
- User: An authenticated client session. When authentication is enabled, will be associated with at least one role, but may be granted many.
- Operation:  In the context of this document, we consider an operation to be a user-initiated action, such as executing a CQL statement or performing an RPC call using JMX.

## Assumptions and Requirements

The assumptions made in this proposal are:

- Capabilities are set at the role level, from which follows:
    - Enforcement of any restrictions based on capabilities requires authentication.
    - It is feasible to group capabilities sensibly, to avoid falling back to defining a role per-restriction and the associated administrative overhead.
- Possession of capability is qualified by an IResource, meaning that the role may possess a certain capability for operations on some resources and not on others.

- Any given capability may not necessarily be applicable or relevant to all types of resource.
- An operation may require zero, one or several capabilities in order to be performed.
- The set of required capabilities for an operation can be determined prior to execution.
- The set of resources used in an operation can usually be determined prior to execution.
- Operations may be interactive (e.g. execution of CQL queries, or RPC via JMX) or they may be non-interactive (e.g. activation of probabilistic query tracing).
- Capabilities are complementary to authz. When both features are enabled, performing a given operation on specific resource requires both the necessary permission and capabilities.

The requirements of an implementation are:

- This should be an opt-in feature and like authn/authz should be disabled in the default configuration.
- Unless specifically restricted, a role possesses every defined capability.
- Restrictions on capabilities are inherited transitively with roles.
- There should be CQL syntax for adding, removing and viewing the restrictions on a role's capabilities.
- Where a user's capabilities are insufficient to execute an operation it is not performed. For interactive operations, an error response is returned. For non-interactive operations, a suitable message is logged.
- If the feature is not enabled there should be no performance impact.
- If the feature is enabled any additional overhead should be minimised.
- Related metrics should be exposed to operators (precisely which metrics would be useful is a point for discussion).

# Proposal Overview

The basic proposal is to implement something roughly analogous to POSIX capabilities[1]. This effectively partitions the privileges of the root user into a number of discrete capabilities, which can then be granted or denied to individual processes (or actually, threads). A concrete example from the man page is the CAP_SYS_TIME capability, which is required by any process which wants to change the system clock by calling `settimeofday`, `stime` or `adjtimex`.

While there are conceptual similarities between POSIX capabilities and this proposal, the POSIX model doesn't precisely address our use case. Process capabilities under POSIX interact with capabilities specified at the filesystem level and also with SELinux. Processes are able to temporarily drop privileges, which is not needed for our purposes, so in that regard the implementation in Cassandra should probably be a little more static and straightforward.

[1] http://linux.die.net/man/7/capabilities

However, we do have a requirement to attach capabilities not only to a role, but to further qualify them with the resource they relate to. This is to enable capabilities to be restricted on a per-role, per-resource level so that, for example, a user can be prevented from performing certain operations on a single set of tables.

This proposal models capabilities in C* as a blacklist, or a set of capability restrictions. That is, the default assumption is that a user can perform any operation unless a restriction is specified which denies her the necessary capabilities. Such restrictions are a property of a (role, resource) tuple, just like permissions. Also like permissions, restrictions are inherited when one role is granted to another. So the restrictions enforced upon a particular user are the union of all of the restrictions for all of the roles granted directly or transitively to that user. The presence of a restriction for one role in the user's granted set constitutes a restriction for that user. Being a blacklist, a restriction cannot be overridden or superseded and can only be revoked by removing it completely.

# Proposal Details

As capabilities will be defined as a blacklist, the process of managing them can be thought of in terms of adding and removing restrictions on capabilities for a given combination of role and resource. If a role has a restriction imposed, any user who has that role is not permitted to perform an operation using the named resource which requires the named capability. It should be noted that containment relationships between resources exist. For instance, a keyspace encompasses all tables within it, so any restriction applied at the keyspace level applies to all tables in that keyspace. This is consistent with the way permissions work.

## Internal API

The API for management of capabilities will defined in a new interface, `ICapabilityManager`. As with roles, authn and authz, a default implementation which uses C* tables to persist the capabilities data will be included. The data model for the base capabilities data is described below.

`ICapabilityManager` is an extension point for third parties to plug in their own implementations. It will define methods to manage restrictions for a role (add, remove, remove all, list), to list all restrictions across all roles, as well as a method to verify whether a set of capabilities required to perform an operation using a specific resource are unrestricted for a particular role.

The semantics of this verification method are as follows:

- An operation on resource T1 requires capabilities C1 & C2.
- The primary role, R1, of a connected client (i.e. the 'logged in user') has been granted roles R2 and R3.

- R2 has also been granted R4 and R5, so the transitive set of roles is {R1, R2, R3, R4, R5}
- The operation is permitted if **none** of these roles has a restriction on **either** C1 or C2 which relates to T1.
- A restriction relates to T1 if it is qualified with T1 or any parent resource of T1
- Conversely, if **any** of the roles does have a defined restriction on C1 **or** C2 that applies to T1, execution is not permitted and an authz error should be raised.

This capabilities check should be performed before any execution of the operation begins. For CQL statements, this implies making it part of the `checkAccess` implementation.

## CQL Syntax

To add a new restriction:

```
CREATE RESTRICTION IF NOT EXISTS ON <role> USING <cap> WITH
<resource>;
```

To remove a restriction:

```
DROP RESTRICTION IF EXISTS ON <role> USING <cap> WITH <resource>;
```

With the optional `IF EXISTS/IF NOT EXISTS`, attempting creation of an existing restriction or removal of a non-existent restriction is handled silently as a no-op. Without the clause, an invalid request error should be returned. If an invalid role or capability name is supplied, an invalid request error is returned. Part of the validation should also be to verify that the specified capability is applicable for the named resource.  This is mentioned in more detail in the section Initial Set of Capabilities.

List restrictions; to provide query functionality consistent with the equivalent `LIST PERMISSIONS` & `LIST ROLES` statements, this has a slightly more complex grammar:

```
LIST RESTRICTIONS;
LIST RESTRICTIONS ON ANY ROLE USING ANY CAPABILITY WITH <resource>;
LIST RESTRICTIONS ON ANY ROLE USING <cap>;
LIST RESTRICTIONS ON ANY ROLE USING <cap> WITH <resource>
```

Without a qualifying `ON ROLE` clause, elevated privileges should be required. In terms of precedent, listing all permissions requires superuser privileges (with the default IAuthorizer) whilst listing all roles requires DESCRIBE permission on the root RoleResource. Neither of these fit 100% here, but initial preference is to follow the Roles model and have unqualified LIST RESTRICTIONS require DESCRIBE on ALL ROLES. The reasoning for this is to avoid (ab)use of superuser and hence potential security holes wherever possible.

```
LIST RESTRICTIONS ON <role>;
LIST RESTRICTIONS ON <role> USING <cap>;
LIST RESTRICTIONS ON <role> USING <cap> WITH <resource>;
LIST RESTRICTIONS ON <role> USING ANY CAPABILITY WITH <resource>;

LIST RESTRICTIONS
   (ON <role>)?
   (USING <cap>)?
   (WITH <resource>)?
   (norecursive)?
```

The presence of the optional `norecursive` clause indicates that where qualifying `ON ROLE` clause is also present, only restrictions created directly against that role should be included. Otherwise, include any restrictions which apply to that role. Without a qualifying `ON ROLE` clause, no recursive has no effect.

## CQL Protocol

If it is acceptable to use the existing `2100 Unauthorized` error response when operations are denied due to restrictions, no protocol level changes are required. If a dedicated response code is required it will need to be added in protocol V5 (and so this feature will become dependent on the protocol version bump).

## Extensibility

Third party code which extends C* along extension points such as `Index`, `IRoleManager` etc may find it useful to define their own capabilities so that admins can restrict aspects of custom functionality. To support that, the set of defined capabilities in the system should be extensible, i.e. not implemented as an enum. Clearly, non-standard capabilities can only be used by custom code, so there shouldn't be any need to provide a declarative mechanism of defining them (i.e. no CREATE CAPABILITY statement in CQL). However, they must be supported in CREATE/DROP RESTRICTION statements, so the grammar should not preclude that. Additionally, there must be a means to check at validation time whether a capability referenced in a CREATE/DROP statement is valid in the particular context. Code to validate standard capabilities for existence and for applicability to the named resource will be provided in C* and the checking for non-standard capabilities will be delegated to the `ICapabilityManager`.

## Data Model for Default Implementation

The base restriction data closely resembles the existing `system_auth.role_permissions` table. Here, each row contains all restrictions for a given role on a given resource.

```
CREATE TABLE system_auth.capability_restrictions (
    role text,
    resource text,
    restrictions set<text>,
    PRIMARY KEY (role, resource)
)
```

To allow all restrictions to be invalidated when the role or resource they refer to is dropped, an inverted index table is also maintained. Again, this mirrors the permissions data model.

```
CREATE TABLE system_auth.resource_capability_restrictions_index (
    resource text,
    role text,
    PRIMARY KEY (resource, role)
)
```

## Caching

Because the expectation is that the number of roles in a typical system is very low, coupled with the fact that restrictions define a blacklist, it should be possible to aggressively cache restriction data to minimize the number of read operations. Additionally, the rate of change is expected to be very low, with reads vastly outnumbering writes. Given these characteristics, it may be worth employing a generational caching strategy for the default implementation rather than the more generic `AuthCache` used for permissions, credentials and roles.

`AuthCache` is really just a thin wrapper around Guava's `LoadingCache` (currently at least, CASSANDRA-10855 proposes to switch to Caffeine). The config for an cache includes validity and refresh periods as they invalidate by expiry while performing periodic refreshes on a background executor. Instead of doing that, the restrictions cache could load the entire dataset during initialization and invalidate/reload it all when a relevant change is performed. To do this, a separate generations table will be updated any time a restriction is added or removed for any role. The cache will read the generations table periodically and whenever a change is detected, reload the entire dataset. The dataset should be small enough for this to put insignificant load on the system and with a reasonable RF remain suitably available and responsive. The anticipated benefit of using a generational cache is that `AuthCache` requires at least 1 db read for every key accessed during a given validity period regardless of whether the underlying data has changed. Granted, this isn't a huge burden but given the very low expected rate of change in this dataset, this would be reduced (in practical terms) to a single read (of the generations table) per validity period.

The generational cache could be enabled by default but we should allow fallback to an `AuthCache` using either system properties or a flag in yaml. Updates to the generations table

would be performed using LWT whenever a CREATE RESTRICTION or DROP RESTRICTION statement is executed. There may be contention when such modifications are made concurrently, but as long as the generation increases monotonically some delay shouldn't matter. This bumping of the generation will be triggered using an event listener via a new `IAuthSubscriber` interface. Execution of relevant DCL statements will notify any subscribed listeners, which the capabilities cache will provide. It may be possible in a follow up ticket to take a similar approach to improving the caches for credentials, permissions and roles and leave `AuthCache` as a general purpose fallback for custom implementations (or remove it altogether).

## Permissions

Although the overlap with the permissions system is minimal, creating and removing restrictions clearly requires certain privileges. The initial proposal is that in order to add or remove a restriction, the executing user must have AUTHORIZE permission on the target role, which is consistent with GRANT/REVOKE ROLE. In the usual way, this permission may be held directly or transitively.

## Initial Set of Capabilities

Of course, this is wide open for discussion but an initial candidate set of capabilities that would meet all the use cases discussed on CASSANDRA-8303 would include:

- TRUNCATE
- FILTERING
- NATIVE_INDEX
- CUSTOM_INDEX
- UNLOGGED_BATCH
- LOGGED_BATCH
- LWT
- QUERY_TRACING
- UNPREPARED_STMT
- MULTI_PARTITION_READ
- PARTITION_RANGE_READ
- MULTI_PARTITION_AGGREGATION
- CL_ANY_WRITE
- CL_ONE_READ
- CL_ONE_WRITE
- CL_LOCAL_ONE_READ
- CL_LOCAL_ONE_WRITE
- CL_TWO_READ
- CL_TWO_WRITE
- CL_THREE_READ

- CL_THREE_WRITE
- CL_QUORUM_READ
- CL_QUORUM_WRITE
- CL_LOCAL_QUORUM_READ
- CL_LOCAL_QUORUM_WRITE
- CL_EACH_QUORUM_READ
- CL_EACH_QUORUM_WRITE
- CL_ALL_READ
- CL_ALL_WRITE
- CL_SERIAL_READ
- CL_LOCAL_SERIAL_READ

While these are mostly self explanatory, there is a special case worth mentioning. The operations which would require TRACING may start before the set of resources is known. Furthermore, custom tracing implementations may begin persisting session data before the point at which those resources are known, giving implementers a headache over cleaning up aborted sessions. For that reason, restrictions on that capability should only be allowed on **all keyspaces**, at least in v1. Restricting the TRACING capability for a particular keyspace or table should not be allowed.

Tracing is also a non-interactive operation. As alluded to previously, should tracing be enabled for a request (either explicitly or as a consequence of trace probability) and the authenticated user has this restriction in place the tracing session id should be removed from the `QueryState` before query execution so that no session is created. The statement execution should then proceed as normal but push a notification back to the client warning that the trace directive was ignored.

On a more general note, all of these initial capabilities are only applicable to one kind of resource, `DataResource` which represents keyspaces and tables. The expectation then would be that in version 1, attempting to execute a CREATE RESTRICTION statement which references a `RoleResource`, `FunctionResource` or `JMXResource` would receive an invalid request response (or if we want to be consistent with GRANT PERMISSIONS, an invalid syntax exception, even if that is slightly semantically wonky). In future, additional capabilities which may apply to other resource types could be added. This could be used, for instance, to exercise more fine grained control over what types of operations a UDF is permitted to perform when executed by certain users.