

Formal Specification of Access Control for OneDrive (Personal Edition)

Project Team 31

December 14, 2025

Abstract

This document presents a formal model for the access control system of Microsoft OneDrive (Personal Edition). It begins by analyzing the specific operational characteristics of the system, justifies the selection of a Capability-Based architectural model over traditional alternatives (DAC, RBAC), and concludes with a rigorous mathematical specification of entities, invariants, policy logic, and state transitions.

1 Introduction & Scope

This model formally defines the access control mechanism for Microsoft OneDrive (Personal). Unlike traditional Discretionary Access Control (DAC) systems (e.g., Linux file permissions), OneDrive Personal relies heavily on a **Capability-Based** model augmented by **Identity Constraints**. Access is primarily granted through *Sharing Links* (Tokens) rather than direct Access Control Lists (ACLs), with strict hierarchical inheritance and a high-security partition known as the *Personal Vault*.

2 System Characteristics Analysis

Before defining the formal model, we identify the core characteristics that define the behavior of OneDrive for Personal Use. These observations form the basis for the mathematical constraints defined later.

1. **Ownership & Root Access:** The account owner (u_{owner}) possesses implicit, irrevocable full control over all resources. Unlike enterprise systems, ownership is rooted at the account level; resources cannot be transferred to another user's ownership while remaining in the original drive.
2. **Strict Hierarchical Inheritance:** Resources exist in a strict tree structure. Permissions granted to a folder are automatically inherited by all descendant files and subfolders. If a file is moved out of a shared folder, it immediately loses inherited permissions.
3. **The “Personal Vault” Partition:** A high-security partition exists that enforces a strict boundary. Resources inside the Vault cannot be shared via links, and access requires a specific, elevated authentication context (MFA).
4. **Link-Based Sharing (The Core Mechanism):** Access is primarily granted via *Sharing Links*. A single resource can have multiple active links simultaneously, each with different permissions.
 - **Anonymous:** “Anyone with the link” (Possession of token = Access).

- **Identity-Bound:** “Specific People” (Possession of token + Identity verification).
5. **Permission Granularity:** Personal OneDrive permissions are binary: {view} or {edit}. View allows downloading by default; edit grants download, view, and sharing rights.
 6. **Combined Permission Evaluation:** When multiple permissions apply to a file (e.g., inherited from a folder + direct link), effective access is the union of all permissions. If any permission grants Edit, the effective permission becomes Edit. A file inside a View-shared folder can still become Edit if it receives an Edit link. Conversely, if a folder is shared as Edit, no child file or subfolder can be restricted to View.
 7. **Stateful Constraints:** Access tokens carry their own state, including expiration dates and password requirements. These are attributes of the *link*, not the user.
 8. **External Interaction:** The system supports interaction with users outside the local namespace (Guest users). Anonymous links do not require account creation, relying entirely on the validity of the token.

3 Architectural Justification

We propose a **Link-Mediated Access Model (LMAM)**, which is a derivative of the **Capability-Based Access Control (CapBAC)** model. Below is the justification for this choice and why traditional models are insufficient.

3.1 Why Capability-Based?

In OneDrive, the *Sharing Link* acts exactly as a capability token: it is a communicable, unforgeable reference that grants authority.

- **Token Identity:** The system evaluates the validity of the *Link* first, and the *User* second.
- **Decoupled Access:** CapBAC allows access to be granted without pre-registering the user’s identity. This perfectly models the “Anyone with the link” feature.
- **Revocation:** Revoking access in OneDrive is done by deleting the link (destroying the capability), which affects all holders of that link. This is a hallmark of CapBAC.

3.2 Why other models are infeasible

- **vs. Discretionary Access Control (DAC/ACLs):** DAC relies on a list of users attached to a file. DAC cannot effectively model anonymous access or the scenario where one user holds two different links (e.g., a View link and an Edit link) for the same file.
- **vs. Role-Based Access Control (RBAC):** RBAC assigns permissions to roles (e.g., Manager, Staff). In a personal drive, there are no organizational roles. RBAC lacks the granularity to share a single specific file with a specific external user.
- **vs. Relationship-Based Access Control (ReBAC):** ReBAC (used by social networks) relies on a social graph (Friends). OneDrive allows sharing with strangers who have no prior relationship with the owner, making ReBAC structurally incorrect for this use case.

4 Entity Definitions (Sets)

We define the universe of entities involved in the system.

4.1 Users (\mathcal{U})

The set of all principals.

- u_{owner} : The single root owner of the OneDrive account.
- u_{auth} : Authenticated Microsoft account holders.
- u_{anon} : Anonymous users (guest access via public links).

4.2 Resources (\mathcal{R})

The set of objects protected by the system.

- \mathcal{R}_{files} : Leaf nodes (documents, photos).
- $\mathcal{R}_{folders}$: Container nodes.
- $\mathcal{R}_{vault} \subset \mathcal{R}$: The “Personal Vault” subset, which requires elevated authentication.
- \perp : The root directory.

4.3 Actions (\mathcal{A})

The set of operations a user can perform.

$$\mathcal{A} = \{\text{view, download, edit, upload, delete, share}\}$$

4.3.1 Action Hierarchy and Required Permission

Let $\mathcal{P} = \{\text{view, edit}\}$ be the permission set carried by links. Define the required-permission function:

$$\begin{aligned} \text{req} : \mathcal{A} &\rightarrow \mathcal{P}, & \text{req(view)} &= \text{view}, & \text{req(download)} &= \text{view}, \\ \text{req(edit)} &= \text{edit}, & \text{req(delete)} &= \text{edit}, & \text{req(upload)} &= \text{edit}, & \text{req(share)} &= \text{edit} \end{aligned}$$

Actions have an implicit hierarchy: `edit` \succ `view` (`edit` subsumes `view`).

This means that possessing the `edit` permission implicitly grants `view` and `download` capabilities. Formally:

$$\text{edit} \in P_l \implies \{\text{view, download}\} \subseteq P_l^{\text{effective}}$$

where $P_l^{\text{effective}}$ is the effective permission set for link l .

Definition (Effective Permission Expansion):

$$l.\text{perms}^{\text{effective}} = \begin{cases} \{\text{view, edit}\} & \text{if } \text{edit} \in l.\text{perms} \\ \{\text{view}\} & \text{if } l.\text{perms} = \{\text{view}\} \end{cases}$$

4.4 Links/Tokens (\mathcal{L})

The set of active sharing artifacts.

- Each link $l \in \mathcal{L}$ is a unique token that can be created by the owner or by any user with `edit` permission on the resource.

5 State & Relations

These relations define the “snapshot” of the system at any time t .

5.1 Ownership Relation

Since this models a Personal OneDrive account, ownership is global to the account holder.

$$\text{owns}(u, r) \iff u = u_{\text{owner}}$$

This implies that for every resource r in the system, the account owner is the sole owner.

5.2 Resource Hierarchy

$$\text{parent} : \mathcal{R} \setminus \{\perp\} \rightarrow \mathcal{R}_{\text{folders}}$$

Every resource (except root) has exactly one parent.

$$\text{is_ancestor}(p, c) : \text{True if } p \text{ is a parent (recursive) of } c.$$

5.3 Link Attributes

Every link $l \in \mathcal{L}$ is a tuple defined by the function $\text{attr}(l)$:

$$\text{attr}(l) = \langle \text{target}, \text{scope}, \text{perms}, \text{constraints}, \text{key} \rangle$$

1. **target** (r_l): The resource $r \in \mathcal{R}$ the link points to.
2. **scope** (S_l): The target audience.
 - **ANYONE**: Public link. Access is granted solely based on possession of the link token; no identity verification is required.
 - **SPECIFIC**: Restricted to a defined set of recipients. Identity verification is required.
3. **perms** (P_l): The allowed actions ($\{\text{view}\}$ or $\{\text{view}, \text{edit}\}$).
4. **constraints** (C_l): A set of restriction flags.

$$C_l \subseteq \{\text{expiry}(t_{\text{exp}}), \text{password}(pw)\}$$

5. **key** (K_l): The secret token string (URL) possessed by the user.

5.4 Access State

- $\text{recipients} : \mathcal{L} \rightarrow \mathcal{P}(\mathcal{U})$
 - For **SPECIFIC** links, maps to the set of allowed users.
 - For **ANYONE** links, this relation is not used (identity is not checked).
- $\text{holding} : \mathcal{U} \times \mathcal{L}$
 - Relation indicating user u possesses the link URL for l .

6 System Invariants

The following properties must hold at all times in any valid system state:

1. Ownership Uniqueness:

$$\forall r \in \mathcal{R}, \exists! u_{owner} : owns(u_{owner}, r)$$

Each resource has exactly one owner.

2. Acyclic Hierarchy:

$$\neg \exists r \in \mathcal{R} : is_ancestor(r, r)$$

The resource hierarchy forms a tree with no cycles.

3. Vault Non-Shareability:

$$\forall l \in \mathcal{L}, r \in \mathcal{R}_{vault} : l.target \neq r$$

Files in the Personal Vault cannot be shared via links.

4. Vault Location Constraint:

$$\forall r \in \mathcal{R}_{vault} : (parent(r) = \perp) \vee (parent(r) \in \mathcal{R}_{vault})$$

Resources classified as Vault items must either be at the root (the Vault folder itself) or contained within other Vault folders. They cannot be children of standard shared folders.

5. Link Management Consistency:

$$\forall l \in \mathcal{L}, u \in \mathcal{U} : \text{User } u \text{ modifies } l \implies \text{CanManage}(u, l.target, \Gamma)$$

Only the owner or users with effective edit permissions on the target resource may alter its sharing state.

6. Vault MFA Requirement:

$$\forall u, r \in \mathcal{R}_{vault}, \Gamma : \text{VaultAccess}(u, r, \Gamma) = \text{True} \implies \Gamma.auth_level = mfa$$

Access to Vault resources requires multi-factor authentication.

7. Link-State Consistency:

Every sharing link in the system must be well-formed. Its target must refer to a valid, non-Vault resource, and the holding relation may only reference currently existing links.

$$\forall l \in \mathcal{L} : \text{attr}(l).target \in \mathcal{R} \setminus \mathcal{R}_{vault}$$

$$\forall (u, l) \in \text{holding} : u \in \mathcal{U} \wedge l \in \mathcal{L}$$

This invariant ensures that (i) no link ever points to a deleted or Vault-protected resource, and (ii) no user can “hold” or possess a link that does not exist in the current system state.

8. **Key Uniqueness:** Two different links never share the same key.

This fits the capability intuition (each URL is a distinct capability):

$$\forall l_1, l_2 \in L, l_1 \neq l_2 \Rightarrow \text{attr}(l_1) \cdot \text{key} \neq \text{attr}(l_2) \cdot \text{key}$$

It cleanly enforces that revoking one key never accidentally revokes another.

9. Scope-Recipients Consistency:

$$\forall l \in L : \begin{cases} \text{attr}(l).\text{scope} = \text{ANYONE} \Rightarrow \text{recipients}(l) = \emptyset \\ \text{attr}(l).\text{scope} = \text{SPECIFIC} \Rightarrow \text{recipients}(l) \subseteq U_{\text{auth}} \end{cases}$$

This invariant ensures that (i) public links carry no identity restrictions, and (ii) Specific-People links list only authenticated Microsoft account users.

10. Constraint-Scope Consistency:

$$\forall l \in L : (\text{attr}(l).\text{constraints} \neq \emptyset) \Rightarrow \text{attr}(l).\text{scope} = \text{ANYONE}$$

This invariant encodes the Personal-edition restriction that password protection and expiration dates apply only to public (ANYONE) links.

7 Policy Logic (Predicates)

The core logic used by the Authorization Oracle to permit or deny actions.

7.1 Context Definition

Decisions depend on the request context Γ :

$$\Gamma = \langle t_{\text{now}}, \text{auth_level}, \text{provided_password} \rangle$$

where $\text{auth_level} \in \{\text{none}, \text{standard}, \text{mfa}\}$ (MFA required for Vault).

The context Γ provides dynamic request information (current time, authentication strength, and supplied password) needed to evaluate link expiration, password checks, and Vault access.

7.2 Link Validity Predicate

A link l is **valid** for user u performing action a in context Γ if and only if:

$$\text{ValidLink}(l, u, a, \Gamma) \iff$$

1. **Possession:** $(u, l) \in \text{holding} \wedge$
2. **Identity:** $((l.\text{scope} = \text{ANYONE}) \vee (u \in \text{recipients}(l))) \wedge$
3. **Expiration:** $((\text{expiry} \in l.\text{constraints}) \implies (\Gamma.t_{\text{now}} < l.t_{\text{exp}})) \wedge$
4. **Password:** $((\text{password} \in l.\text{constraints}) \implies (\Gamma.\text{provided_pw} = l.\text{pw})) \wedge$
5. **Permissions:** $(\text{req}(a) \in l.\text{perms}^{\text{effective}})$

where $l.\text{perms}^{\text{effective}}$ includes implicit permissions based on the action hierarchy.

ValidLink checks whether a link can be used in the current request by verifying possession, identity requirements, expiration, password constraints, and the requested permission.

7.3 The Vault Barrier (High Security)

The Personal Vault introduces a mandatory barrier that overrides standard link logic.

$$\text{VaultAccess}(u, r, \Gamma) \iff \begin{cases} \text{True} & \text{if } r \notin \mathcal{R}_{\text{vault}} \\ \text{True} & \text{if } r \in \mathcal{R}_{\text{vault}} \wedge u = u_{\text{owner}} \wedge \Gamma.\text{auth_level} = \text{mfa} \\ \text{False} & \text{otherwise} \end{cases}$$

VaultAccess enforces the Personal Vault's stricter policy by requiring the requester to be the owner and authenticated with MFA before any Vault resource may be accessed.

7.4 Management Authority

To determine if a user has the right to modify access control (create, revoke, or delete links) for a specific resource r :

$$\text{CanManage}(u, r, \Gamma) \iff (u = u_{\text{owner}} \vee \text{edit} \in \text{TotalPerms}(u, r, \Gamma))$$

7.5 Effective Permissions (Union Logic)

Permissions in OneDrive are additive. A user's effective permission on a resource r is the union of all permissions granted by valid links on r and its ancestors.

$$\text{DirectPerms}(u, r, \Gamma) = \bigcup \{l.\text{perms} \mid l.\text{target} = r \wedge \text{ValidLink}(l, u, \text{view}, \Gamma)\}$$

$$\text{InheritedPerms}(u, r, \Gamma) = \bigcup \{l.\text{perms} \mid \text{is_ancestor}(l.\text{target}, r) \wedge \text{ValidLink}(l, u, \text{view}, \Gamma)\}$$

$$\text{TotalPerms}(u, r, \Gamma) = \text{DirectPerms}(u, r, \Gamma) \cup \text{InheritedPerms}(u, r, \Gamma)$$

Note: This union model ensures that permissions cannot be downgraded on child items. If a folder grants `edit`, a file inside inherits `edit`, and a direct `view` link on the file does not remove the inherited `edit` capability.

8 Authorization Decision Function

The Oracle function `decide_access` returns `GRANT` or `DENY`.

$$\text{decide_access}(u, r, a, \Gamma) = \begin{cases} \text{DENY} & \text{if } \neg \text{VaultAccess}(u, r, \Gamma) \\ \text{GRANT} & \text{if } u = u_{\text{owner}} \\ \text{GRANT} & \text{if } \text{req}(a) \in \text{TotalPerms}(u, r, \Gamma) \\ \text{DENY} & \text{otherwise} \end{cases}$$

The decision function evaluates access by first enforcing the Vault's MFA restriction, then granting full authority to the owner, and finally checking whether the requested action is included in the user's effective permissions. All other cases are denied.

9 Operational Semantics (State Transitions)

To model the system completely, we must define how the state changes through administrative operations.

9.1 Create Link (Share Resource)

A sharing link can be created by the owner or by any user with edit permission on the resource.

- **Operation:** $CreateLink(u, target, scope, perms, constraints)$
- **Precondition:** $(u = u_{owner} \vee \text{edit} \in \text{TotalPerms}(u, target, \Gamma)) \wedge target \notin \mathcal{R}_{vault}$
- **Effect:**
 - A new link l_{new} is created.
 - $\mathcal{L}' = \mathcal{L} \cup \{l_{new}\}$
 - $\text{attr}(l_{new}) = \langle target, scope, perms, constraints, \text{generate_key}() \rangle$

9.2 Add User to Link

Adds a new recipient to an existing “Specific People” link and grants them possession of the token.

- **Operation:** $AddUser(u, link_id, new_user)$
- **Precondition:**

$$\begin{aligned} & \text{CanManage}(u, l.target, \Gamma) \wedge \\ & (l.scope = \text{SPECIFIC}) \wedge \\ & (new_user \in \mathcal{U}_{auth}) \end{aligned}$$
- **Effect:**

$$\begin{aligned} recipients'(l) &= recipients(l) \cup \{new_user\} \\ holding' &= holding \cup \{(new_user, l)\} \end{aligned}$$

9.3 Revoke User (Specific People)

When the owner or an authorized editor removes a specific user from a shared resource, the link l is *mutated*, not deleted.

- **Operation:** $RemoveUser(u, link_id, target_user)$
- **Precondition:** $\text{CanManage}(u, l.target, \Gamma) \wedge (l.scope = \text{SPECIFIC})$
- **Effect:** $recipients'(l) = recipients(l) \setminus \{target_user\}$
- **Result:** Other users on the same link retain access.

9.4 Delete Link (Global Revocation)

- **Operation:** $DeleteLink(u, link_id)$
- **Precondition:** $\text{CanManage}(u, l.target, \Gamma)$
- **Effect:** $\mathcal{L}' = \mathcal{L} \setminus \{l\}$
- **Result:** Access is lost for **all** users holding this specific capability token.

9.5 Modify Link Constraints

Allows updating expiration or passwords without recreating the link.

- **Operation:** $UpdateConstraints(u, link_id, new_constraints)$
- **Precondition:** $\text{CanManage}(u, l.\text{target}, \Gamma)$
- **Effect:** $l.constraints' = new_constraints$

9.6 Change Permissions (Modify Link)

- **Operation:** $ChangePermissions(u, link_id, new_perms)$
- **Precondition:** $\text{CanManage}(u, l.\text{target}, \Gamma) \wedge new_perms \in \{\{\text{view}\}, \{\text{view, edit}\}\}$
- **Effect:** $l.perms' = new_perms$
- **Constraint:** This operation only affects the specific link l . It does not modify inherited permissions on descendant resources.

9.7 Copy Resource

Creates a new resource with separate identity but identical content.

- **Operation:** $CopyResource(u, source, destination_folder)$
- **Precondition:**
 $(\text{view} \in \text{TotalPerms}(u, source, \Gamma)) \wedge (\text{edit} \in \text{TotalPerms}(u, destination_folder, \Gamma))$
- **Effect:**
 - $\mathcal{R}' = \mathcal{R} \cup \{r_{copy}\}$
 - $\text{parent}(r_{copy}) = destination_folder$
 - $\text{owns}(u_{owner}, r_{copy})$
- **Result:** r_{copy} inherits permissions from $destination_folder$. It does **not** retain direct links from $source$.

9.8 Move Resource (Hierarchy Change)

- **Operation:** $MoveResource(u, resource, new_parent)$
- **Precondition:**

$$\underbrace{\text{CanManage}(u, resource, \Gamma)}_{\text{Source Authority}} \wedge \underbrace{(\text{edit} \in \text{TotalPerms}(u, new_parent, \Gamma))}_{\text{Destination Authority}}$$
- **Effect:** $\text{parent}'(resource) = new_parent$
- **Result:** The resource loses inherited access from the old parent and gains inherited access from the new parent. Direct links to the resource remain valid.

9.9 Delete Resource

A resource may be deleted by its owner, or by an editor who possesses write access to the *parent* container of the resource. This condition prevents a user from deleting the root item of a share (as they lack permissions on the owner's private parent folder), while allowing them to delete files inside a folder they have been granted edit access to.

- **Operation:** $DeleteResource(u, resource)$

- **Precondition:**

$$(u = u_{owner}) \vee (\text{edit} \in \text{TotalPerms}(u, \text{parent}(resource), \Gamma))$$

- **Helper Definition:** Let $\text{Descendants}(r)$ be the set of all resources d such that $\text{is_ancestor}(r, d)$ is true.

- **Effect:**

$$\begin{aligned}\mathcal{R}' &= \mathcal{R} \setminus (\{\text{resource}\} \cup \text{Descendants}(\text{resource})) \\ \mathcal{L}' &= \{l \in \mathcal{L} \mid l.\text{target} \notin (\{\text{resource}\} \cup \text{Descendants}(\text{resource}))\}\end{aligned}$$

- **Result:** The resource is removed from the hierarchy. Note: This operation fails if the user only holds a direct edit link to the resource itself, as that capability does not grant authority over the resource's parent container.

9.10 Create Resource (Upload New File)

- **Operation:** $CreateResource(u, new_file, parent_folder)$

- **Precondition:** $(u = u_{owner} \vee \text{edit} \in \text{TotalPerms}(u, parent_folder, \Gamma))$

- **Effect:**

- $\mathcal{R}' = \mathcal{R} \cup \{new_file\}$
- $\text{parent}'(new_file) = parent_folder$
- $\text{owns}(u_{owner}, new_file)$

10 Conclusion

This formal specification provides a complete mathematical model of the access control system for OneDrive Personal Use. It captures the capability-based link sharing mechanism, hierarchical inheritance with additive permission union, Personal Vault security, and all relevant constraints and state transitions. The model accurately reflects the behavior of Microsoft OneDrive for Personal accounts, excluding Business-specific features such as block download.