1 | Understanding Authorization

To be able to model and create intuitive and understandable authorization flows, one must understand the basis of authorization flows and the elements by which they are made successful.

In the most simply basis, authorization is the process by which permissions are assigned to a user. There are a few models by which authorization is done, and we will aim to list a few successful systems and their downfalls.

To begin this discussion, we will aim to describe a few terms:

- `model`: a system of authentication, like the ones discussed below
- `agent`: the software tool in a `model` by which authorization is checked
- `rule`: a statement made available to the `agent` to validate claims made by `users`
- `resource`: a file/page/tool by which the `model` aims to protect
- `action`: what the `agent` grants to do to a `resource`
- `user`: an actor leveraging the `model`'s `agent` to gain perform `actions`

1.1 | UNIX/BSD PAM

The PAM authentications model, manifested in the `/etc/shadows` files on most *nix systems, is one of the most familiar system of authentication to most.

PAM protects individual resources by checking for an octal permission representing whether or not an `action` on a `resource` is accessible to a `user` or a group of users.

There exists 3 `actions`: read, write, and execute. The authorization `rules` are determined on a `resource` level, and `agents` check against `rules` on access time by users. User permissions override group permissions, which override global permissions.

This systems does not have permission dependencies nor exceptions, group grants and are the only batch executor available.

1.2 | AFS

Access control lists first act separately upon directory resources and file resources. Hence, there are two groups of `actions` available: those that apply to directories and those that apply to files.

Apart from these permissions, groups/batch executors are applied via lookup and literal expressions-based batched execution by the `agent`. `rules`, like PAM, are singletons that only grant or remove specific `actions`.

1.3 | Amazon IAM

Because IAM is the backbone of the Amazon AWS infrastructure, every single lockable AWS permission is considered an `action`. `actions` could be grouped into presents named ”roles”, and `rules` could either apply a role (batched `actions`) or a specific `action` to a group.

Inheritance, therefore, could be applied via the creation of groups, applying those permissions to groups, and editing the groups; to prevent conflicts

1.4 | Microsoft Graph Permissions

Microsoft Graph Permissions grants grouped `actions` to specific users. Each `action` have any number (or no) dependencies; hence, when the `agent` applies any `action` in the group to a users, all of the dependents `action` is simultaneously applied.

Similar to PAM, permission groups could be binned and `rules` applied together. However, there are still no user-level dependency.

1.5 | OAuth

The OAuth `model` allows for authorization of third party `resources` without the `users` credentials.

OAuth creates access tokens which are then passed to third party `models` which use the access token to access `resources` hosted by the resource server.

These temporary access tokens then expire, allowing for a third party to be authenticated without ever giving it credentials and vise versa.

Because of the generalization and lack of `model` of OAuth, there is no hard restriction to permission inheritance

2 | New System Proposal

2.1 | compiled features

- PAM is simple!
  - few `actions`
  - groups -> arbitrary user level granularity
  - AFS
    - Top down from `resources`
    - `resource` dependent `actions` // we dont have this!
      - lack of redundant `actions` // but, we do have this!
- IAM
  - roles
    - `actions` are abstracted from entities they apply to
- Microsoft Graphs
  - `actions` can have dependencies on other `actions`
    - simplicity -> ease of use
  - OAuth
    - `actions` are arbitrary
    - authurization is arbitrary
      - auth is abstracted, thus we can protect credentials

2.2 | **New**

- from OAuth: LEGITIMIZE
  - credential server
  - resource server
  - token access in third party
- tree structure of roles, where each node contains
  - { add, sub, connections }
  - given a user at a node, we DFS through the tree and append actions to a list
- user contains:
  - n roles which generate a final action list (through dfs applied above)
- superuser gets root node

[executer] -> [exucture] [editor] -> [[read], [write]]