Access Control 2

COS 316: Principles of Computer System Design

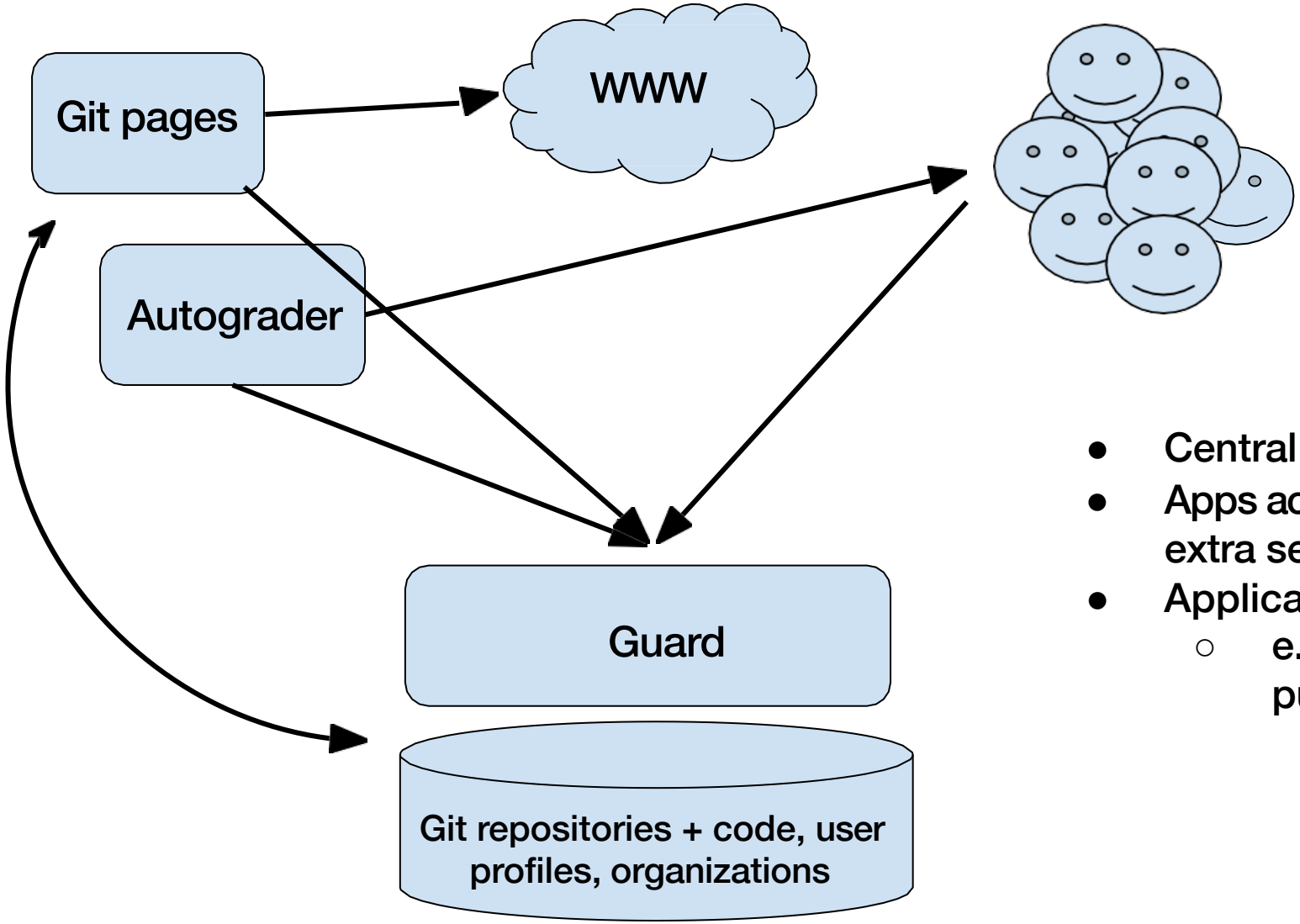
Lecture 15

Wyatt Lloyd & Rob Fish

Access Control

- Restrict access to resources based on the principal trying to access them
  - Canvas:
    - Only Wyatt & Rob can update grades
    - Only you and course staff can see your grades
  - File system on my laptop:
    - Only Wyatt can update or read /Users/wlloyd/.ssh
    - Everyone can read /usr/bin/
  - Facebook:
    - Only I can create posts as me
    - Only the selected audience (global, friends, …) can read the posts

Consider a GitHub-like Ecosystem

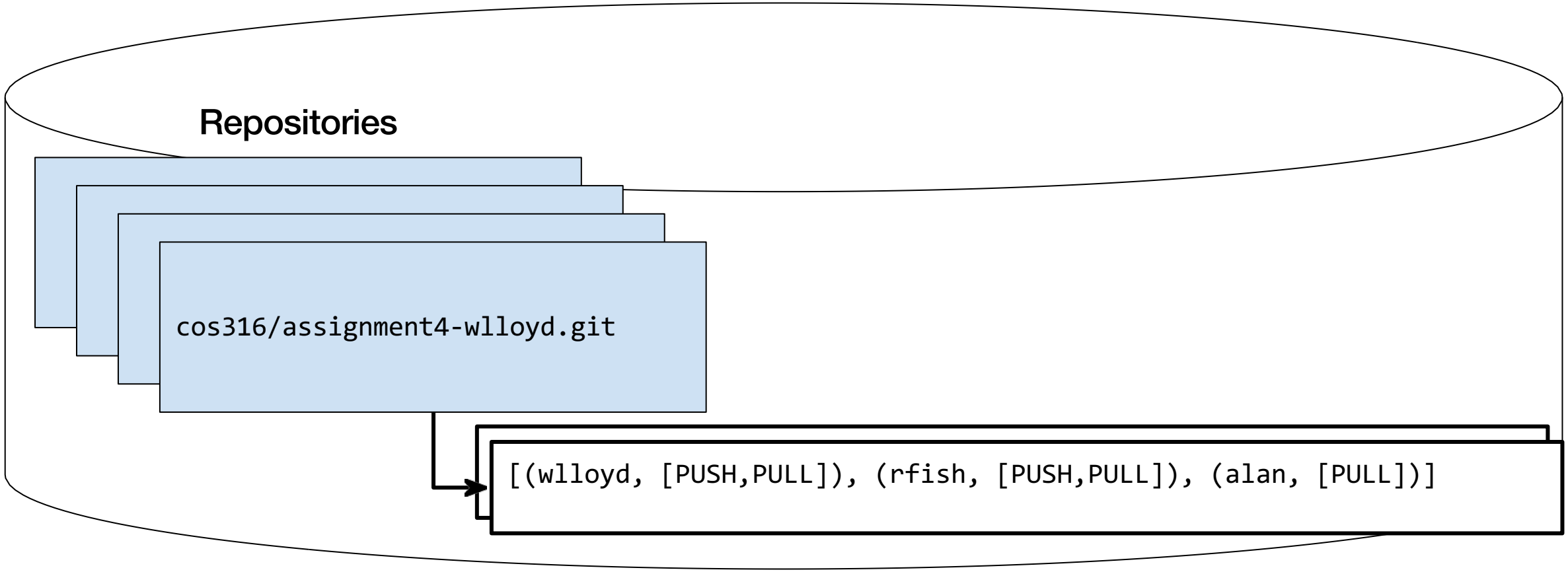


- Central code DB
- Apps access DB resources to provide extra services
- Application access must be restricted:
  - e.g., don't make private repos public

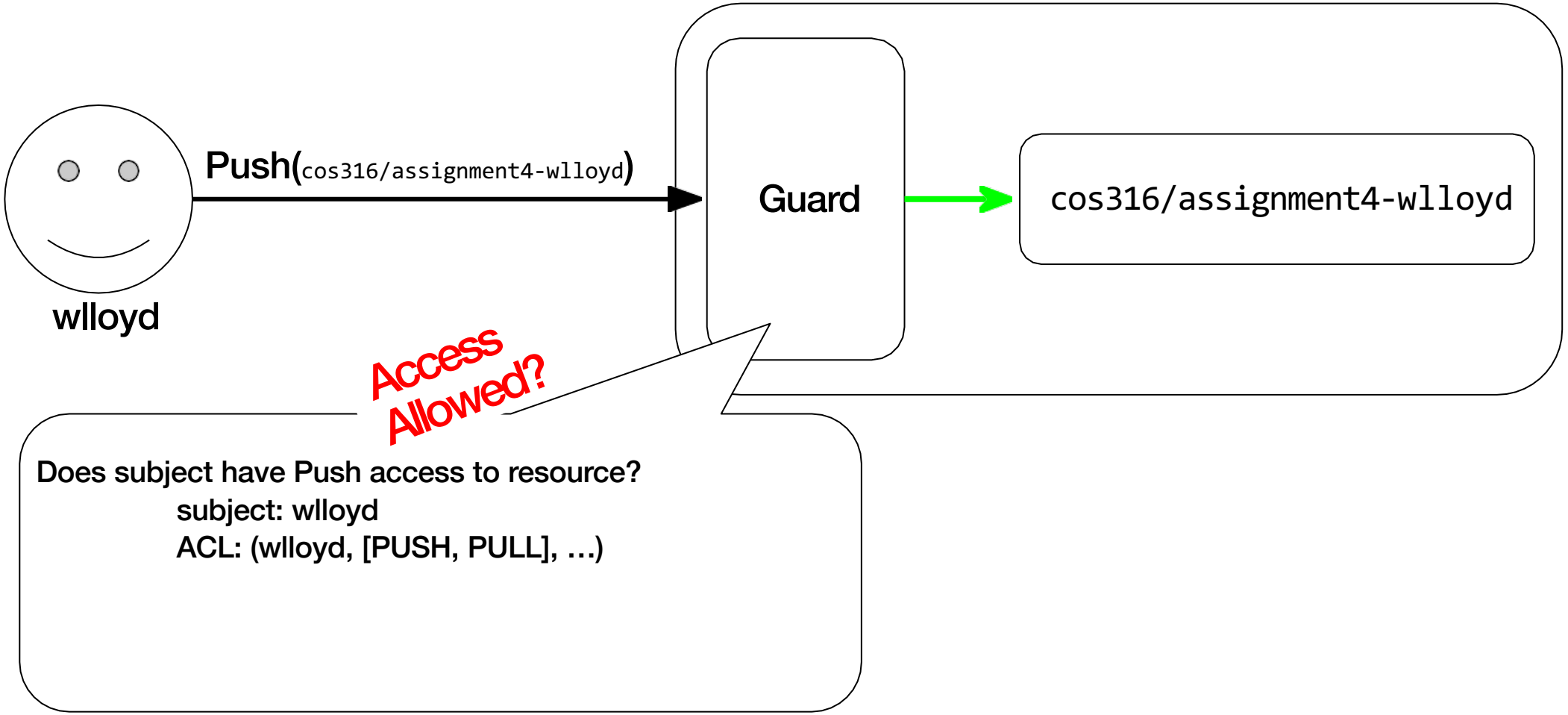
Access Control Lists (ACLs)

Let's Start with User Permissions

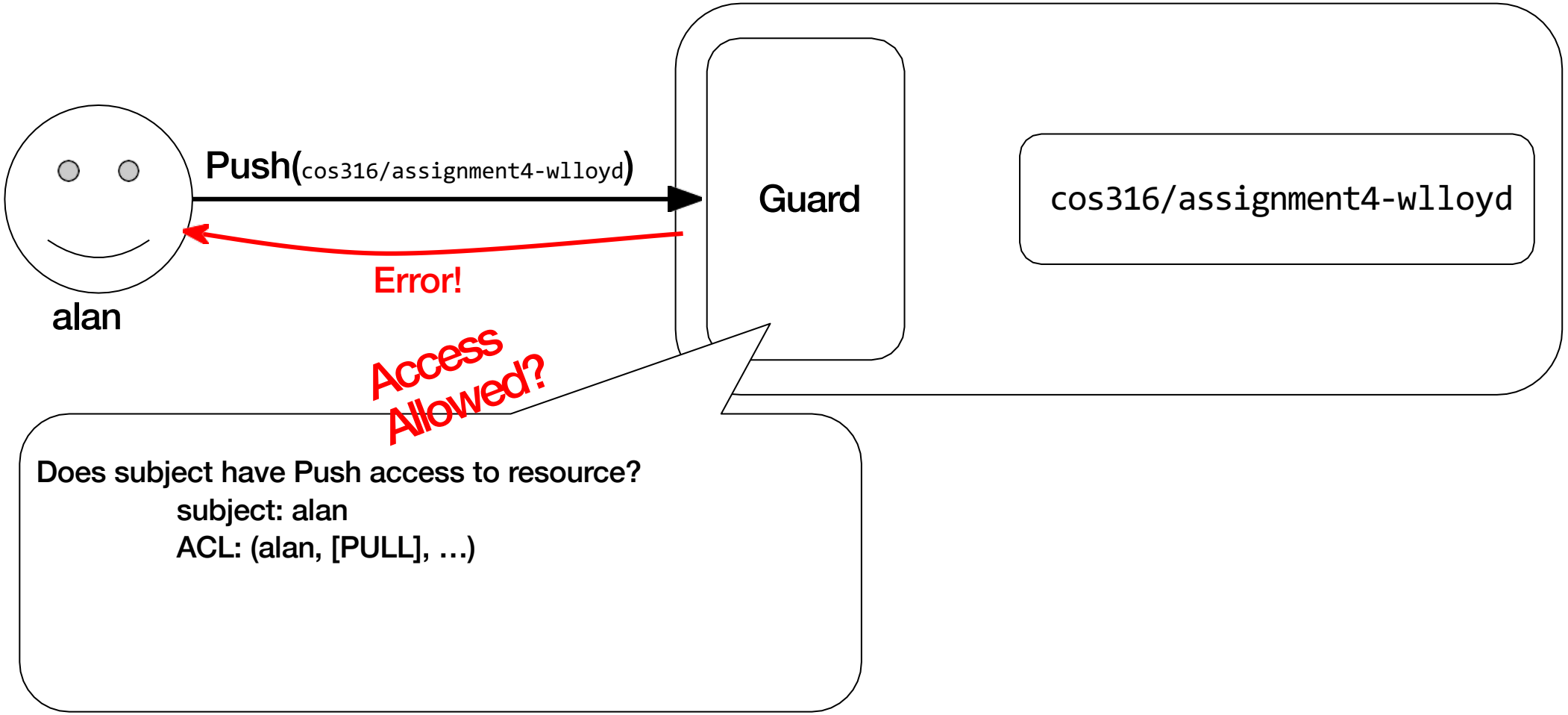
- Associate a list of (user, permissions) with each resource



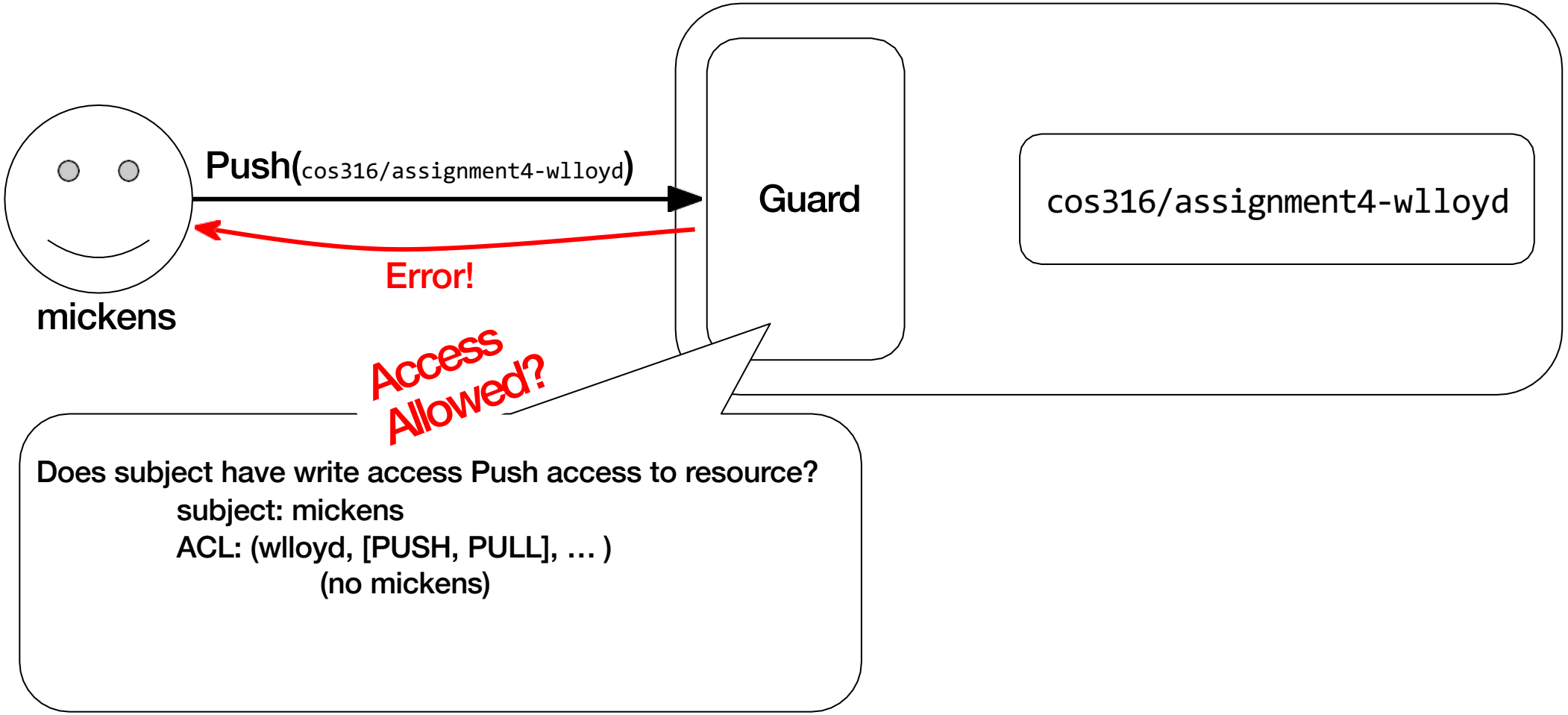
ACLs in Action



ACLs in Action



ACLs in Action



ACLs in Action Q & A

- How do we know subject?
  - Authenticate use username/password, ssh key, …

Extending ACLs to Apps: a-la UNIX

- Applications act on behalf of users
- When an application makes a request, it uses a particular user's credentials
  - Either one user per application
  - Or different users for different requests
- Works great for:
  - Alternative UIs, e.g., the `git` client vs. the GitHub Web UI both act on behalf of users

Extending ACLs to Apps: Special Principals

- Create a unique principal for each app
  - E.g., the “autograder” principal
  - Acts just like a regular user
- When applications make request, they use their own, unique, credentials
- Add application principals to resource ACLs as desired

- Works when
  - Applications need to operate with more than one user's access
    - e.g., the autograder needs to access private repositories owned by different students
  - and less than any one user's access (e.g., less than mine)
    - e.g., the autograder shouldn't be able to access non COS316 repositories

Access Control Lists

Advantages

- Simple to implement
- Simple to administer
- Easy to revoke access

Drawbacks

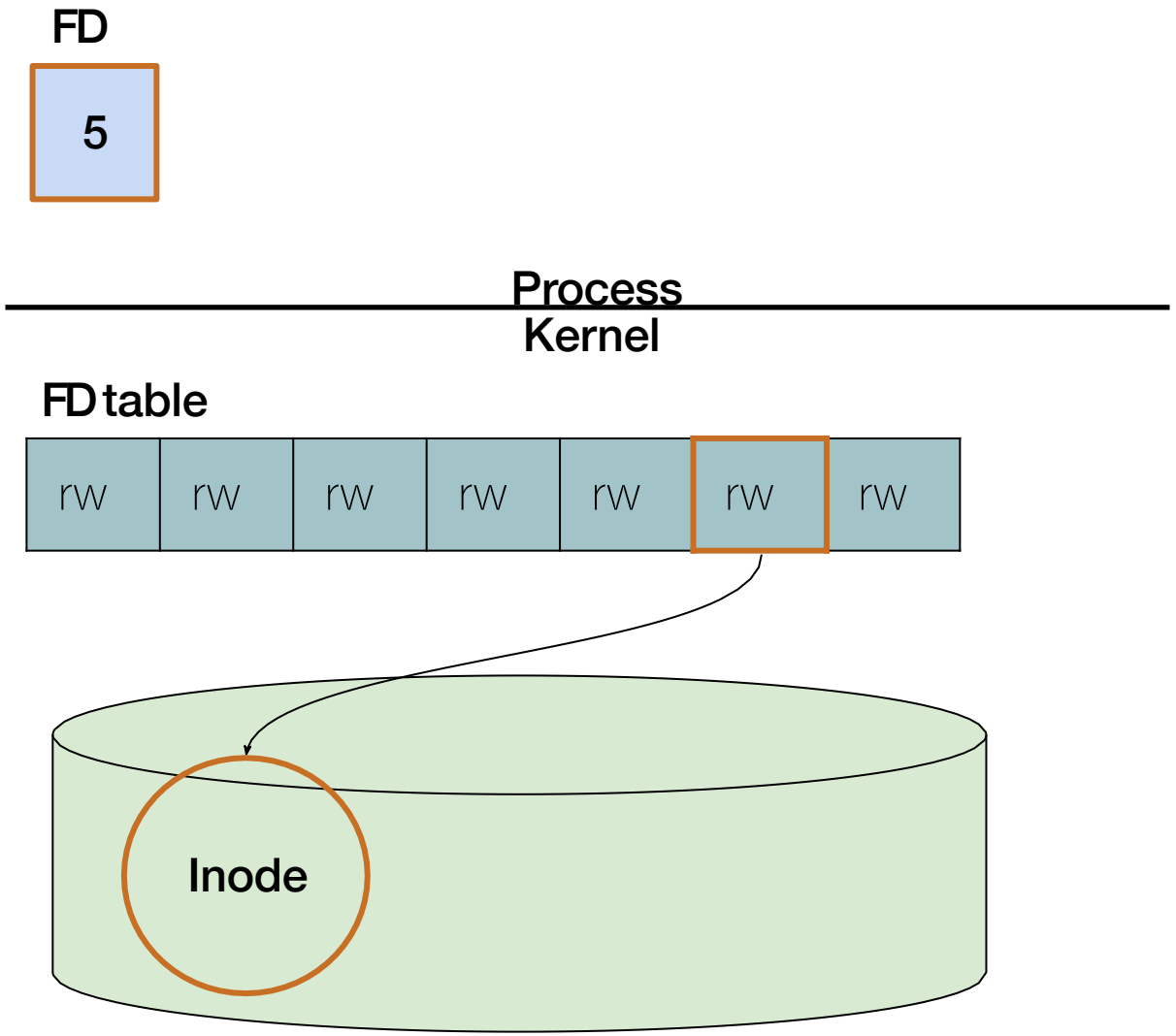
- Tradeoff granularity for simplicity
  - More granular permissions require more complex rules in the guard
- Doesn't scale well
  - e.g., need up to Users * Repos * Access Right entries in ACL table

An Alternative - Capabilities

“[A] token, ticket, or key that gives the possessor permission to access an entity or object in a computer system.” - Capability-Based Computer Systems

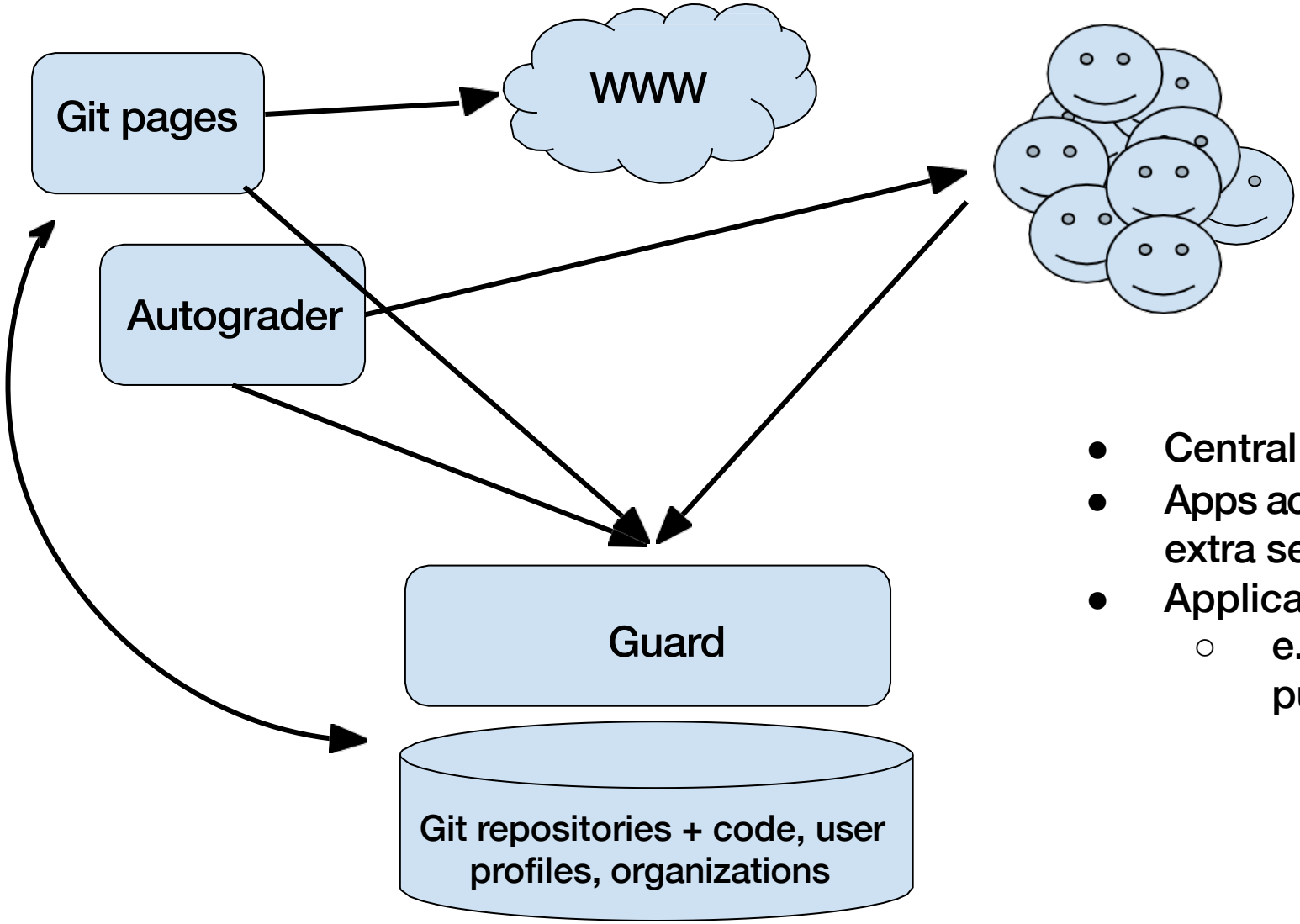
- Self-describing
  - Contains both object name and permitted operations
- Globally meaningful
  - Object and operation names are not subject-specific
- Transferrable
  - A subject can pass a capability to another (e.g., a sub-process, via IPC, a third-party app)
  - Ideally can delegate subset of capabilities
- Unforgeable
  - Subjects cannot create capabilities with arbitrary permissions

File Descriptors as Proto-Capabilities



- Unforgeable ✓
  - Process-level fd is just an index in a kernel structure
- Self-describing ✓
  - Kernel fd contains reference to inode + permissions
- Globally meaningful ✗
  - Fds are process-specific
- Transferrable ✓/✗
  - Via IPC sendmsg/recvmsg

Consider a GitHub-like Ecosystem



- Central code DB
- Apps access DB resources to provide extra services
- Application access must be restricted:
  - e.g., don't make private repos public

User Permissions using Capabilities

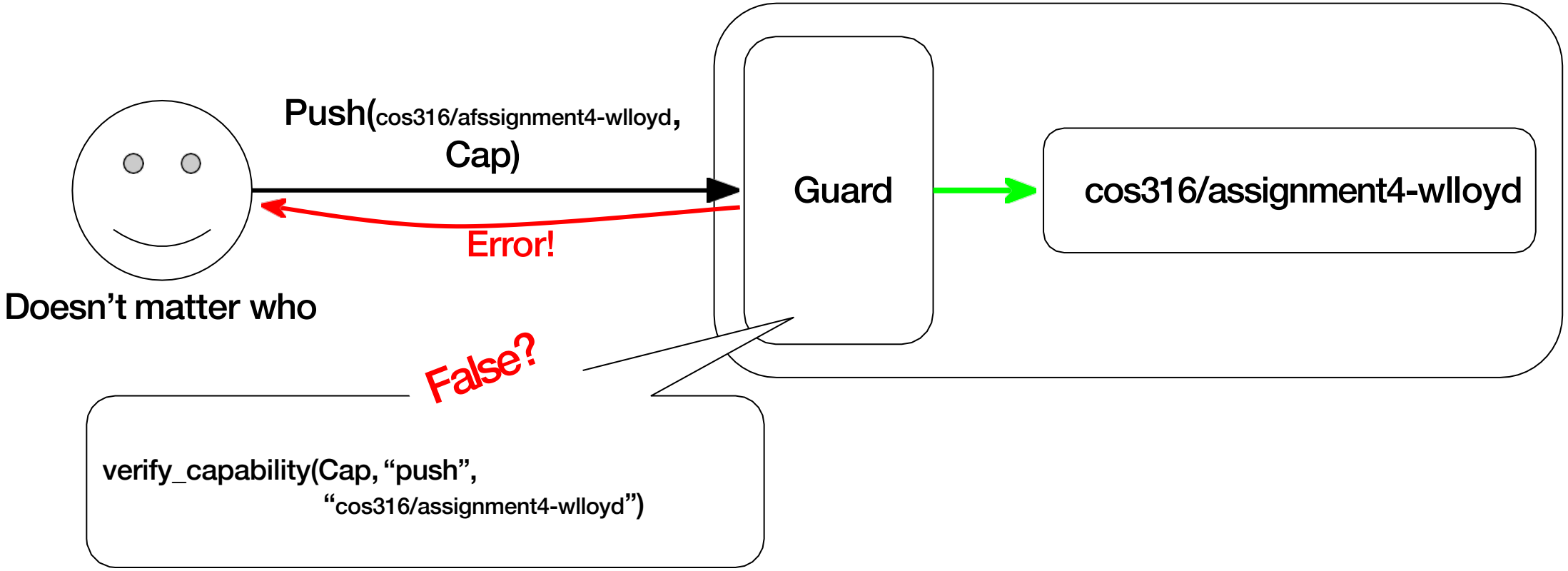
- Hand out communicable, unforgeable tokens encoding:
  - Object
  - Access right
- Users store capabilities, not the database
  - e.g.,
    - “push(cos316/assignment4-wlloyd)”
    - “pull(cos316/assignment4-wlloyd)”

Implementing Capabilities with HMAC

- HMAC - a keyed-hash function: hmac(secret_key, data) hash of data

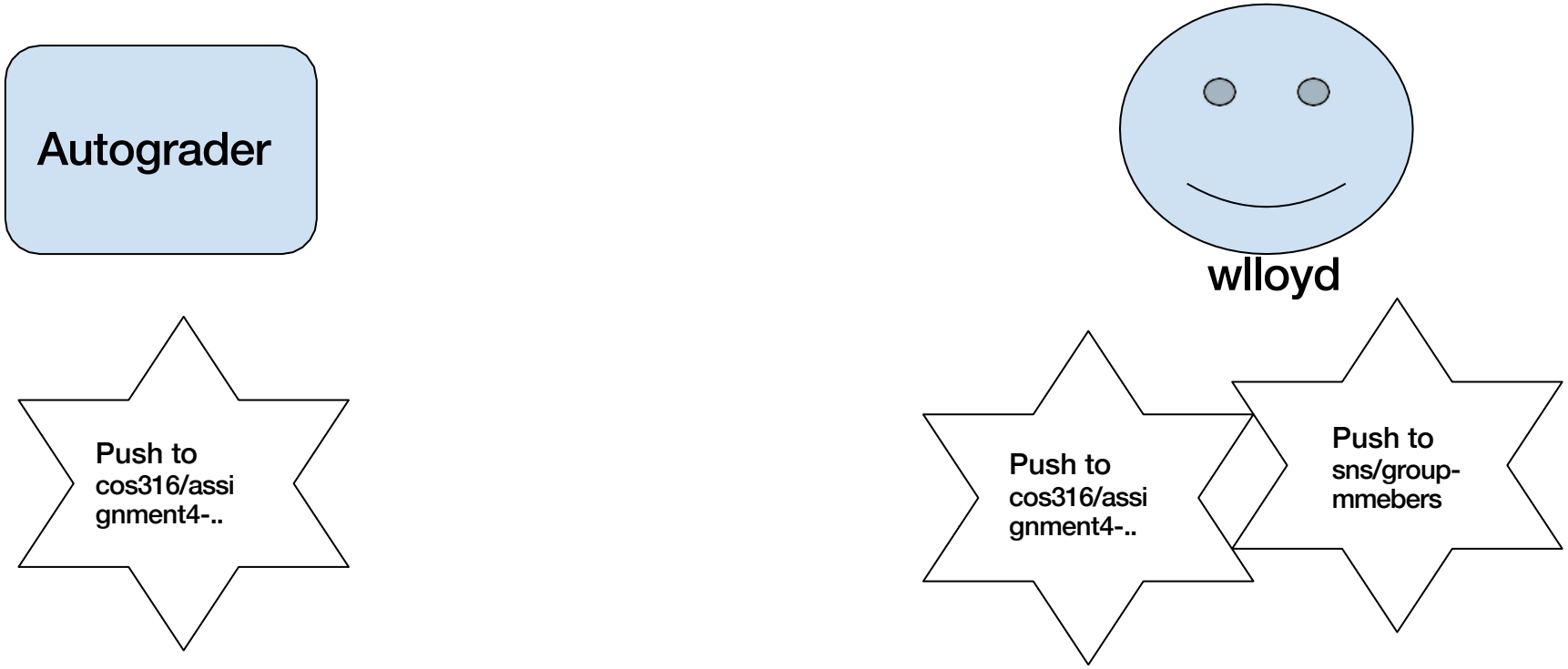
```
fn gen_capability(op, repo) {
     hmac(db_secret, fmt.Sprintf(“%s(%s)”, op, repo))
}


fn verify_capability(cap, op, repo) {
     cap == hmac(db_secret, fmt.Sprintf(“%s(%s)”, op, repo))
}
```

Capabilities in Action

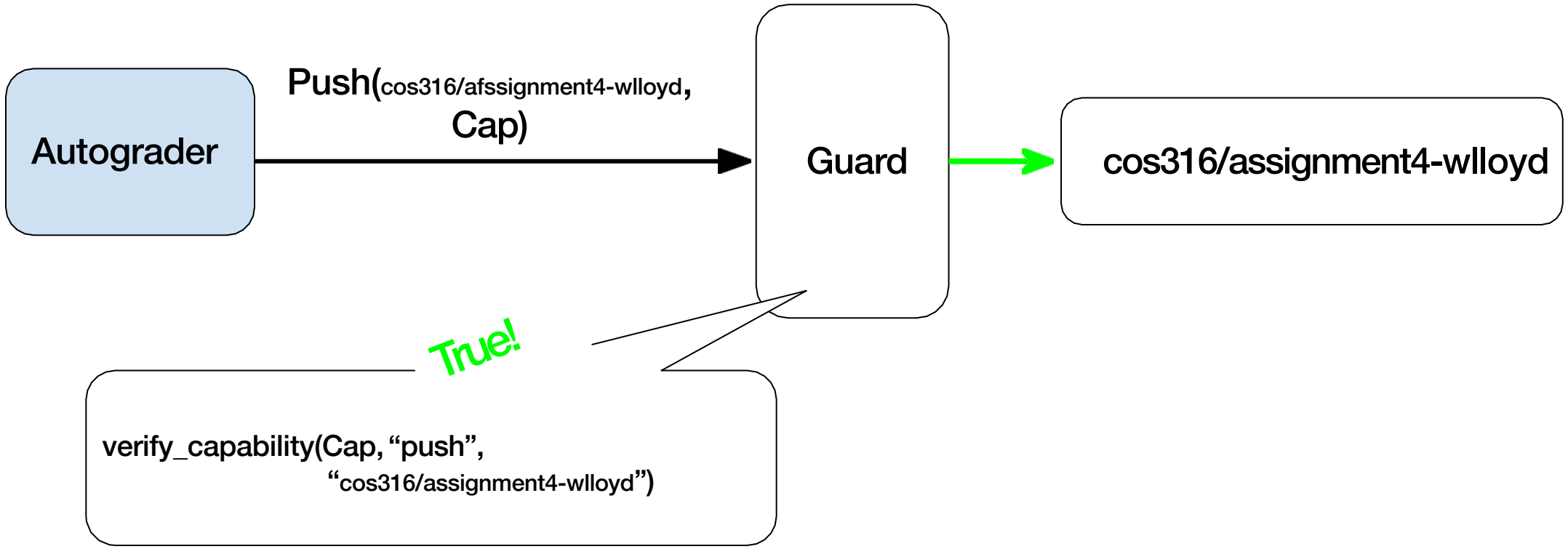


Extending Capabilities to Applications

- Users can simply give applications a subset of their capabilities



Extending Capabilities to Applications



Capabilities

- Advantages
  - Decentralized access control
    - Anyone can “pass” anyone a capability
  - Scales well
  - Granular permissions are simple to check
- Drawbacks
  - How do you revoke a capability?
  - Moves complexity to users/clients
    - Users must manage their capabilities now

Capabilities In The Wild

- Operating Systems
  - History of industry and research operating systems
  - seL4
  - FreeBSD's Capsicum
  - Fuschia OS

- Web
  - S3 Signed URLs
    - URL to private resources, contain signature, expiration, permitted HTTP methods, etc
  - CDN-hosted images/videos (FB, Instagram, YouTube)
    - Browsing via Web page/app is protected by login+cookie, but media typically fetched unauthenticated

We Still Have a Problem

- The autograder is allowed to:
  - read all cos316/ repositories
  - comment on all cos316/ repositories

- Can code from a private repository end up in a comment on a public repository?

- *Solution: Information Flow Control Systems*

Summary

- Access control reflects some real-world policy
  - Design with care
- Ad-hoc access control is very common, but problematic, so prefer systems
- The guard model separates security enforcement from other functionality
- Behavior of a security system is determined by:
  - Isolation mechanism
  - Policy rules
  - Granularity of subjects/resources
- Access Control Lists:
  - Common, but some limitations…
- Capabilities:
  - More scalable, granular, but more complex for users…