

Access Control



COS 316: Principles of Computer System Design

Lecture 15

Wyatt Lloyd

Access Control

- Restrict access to resources based on the principal trying to access them
 - Canvas:
 - Only Wyatt & Kayla 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

A (Slightly) Formal Model

- Resources: the things being accessed
 - A file, network socket, satellite imagery of “nuclear facilities,” missile launcher...
- Subjects: an entity that requests access to an resource
 - A process, network endpoint, ...
 - **Principal**: some unique account or role, such as a user

Ad-hoc Access Control

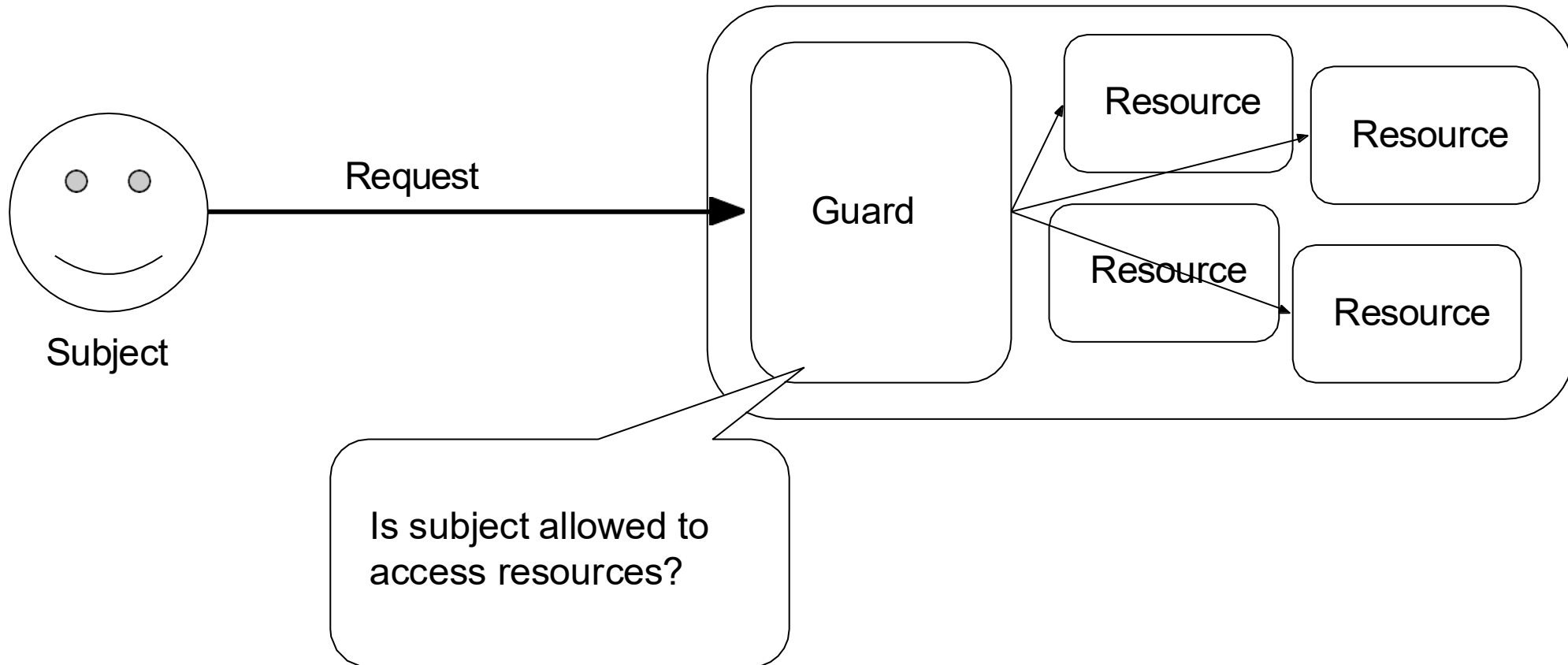
- Access policy enforcement is scattered throughout system

```
fn (profile *Profile) viewProfile(user) (HTML) {  
  if profile.public ||  
    profile.friends.contains(user) {  
    return profile.HTML  
  } else {  
    return HTML.Forbidden  
  }  
}
```

```
fn (profile *Profile) viewFullName(user) (HTML) {  
  if profile.public || user.handle ==  
    "NSA_Backdoor" {  
    return profile.FullName.HTML  
  } else {  
    return HTML.Forbidden  
  }  
}
```

- *Why is this a bad idea?*

The Guard Model



Examples of the Guard Model

- Kernel
 - File system permissions: as long as resources modeled as files, access checks are centralized
- Networks
 - Firewall
 - Apache HTTP Server's .htaccess rules

The Guard Model

- A mechanism, leaves us with many questions:
 - How do we ensure applications only interact via the guard?
 - What kinds of rules does the guard enforce?
 - Who gets to set or change the rules?
 - What is the granularity of subjects and resources?
 - Who gets to create new principles?
- Answers to these questions help determine the expressivity, performance, and security of the system.

Enforcing the Guard Through Isolation

- Key idea, either:
 - Don't “connect” resources directly to applications, only to guard
 - Ensure (somehow) resources access embed guard rules
 - Some combination
- There are three basic kinds of isolation:
 - Hardware enforced:
 - memory protection
 - put the guard and resources on different machines
 - Language-based isolation: use restrictive language to express applications
 - type-safe languages
 - Static validation: symbolic execution, software fault isolation

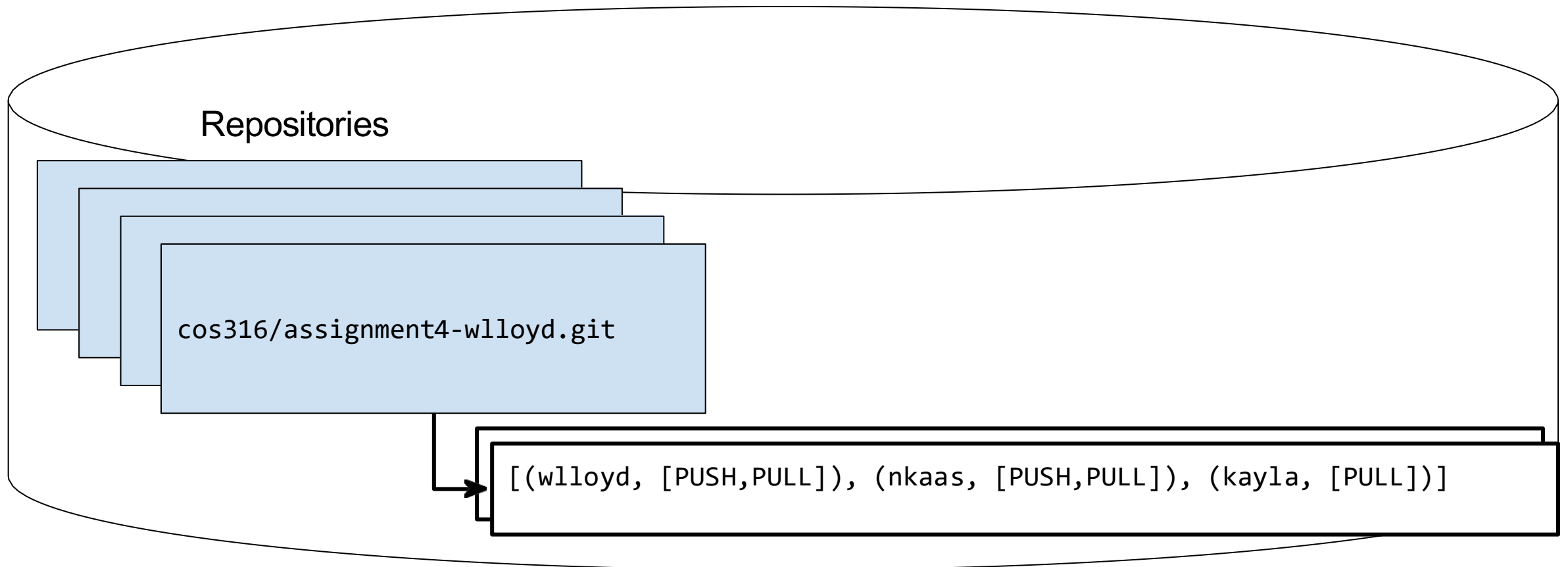
What kinds of rules?

- There are many “policy languages”
 - Access control lists: which subjects can read/write which resources
 - Capabilities: unforgeable tokens that encode specific rules on resources
 - Subjects unnamed
 - Information flow: the relationship between data sources and data sinks
 - Neither subjects nor resources named

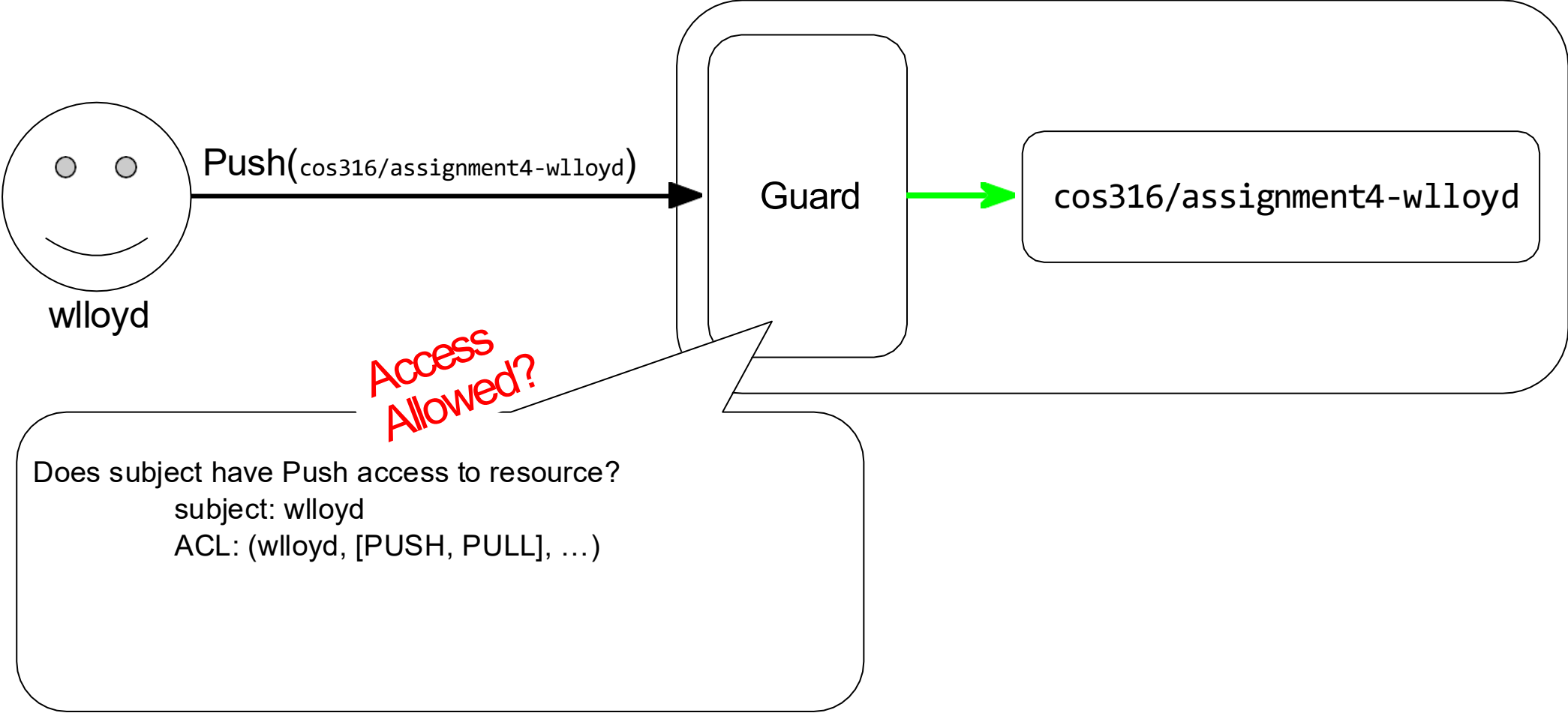
Access Control Lists (ACLs)

Let's Start with User Permissions

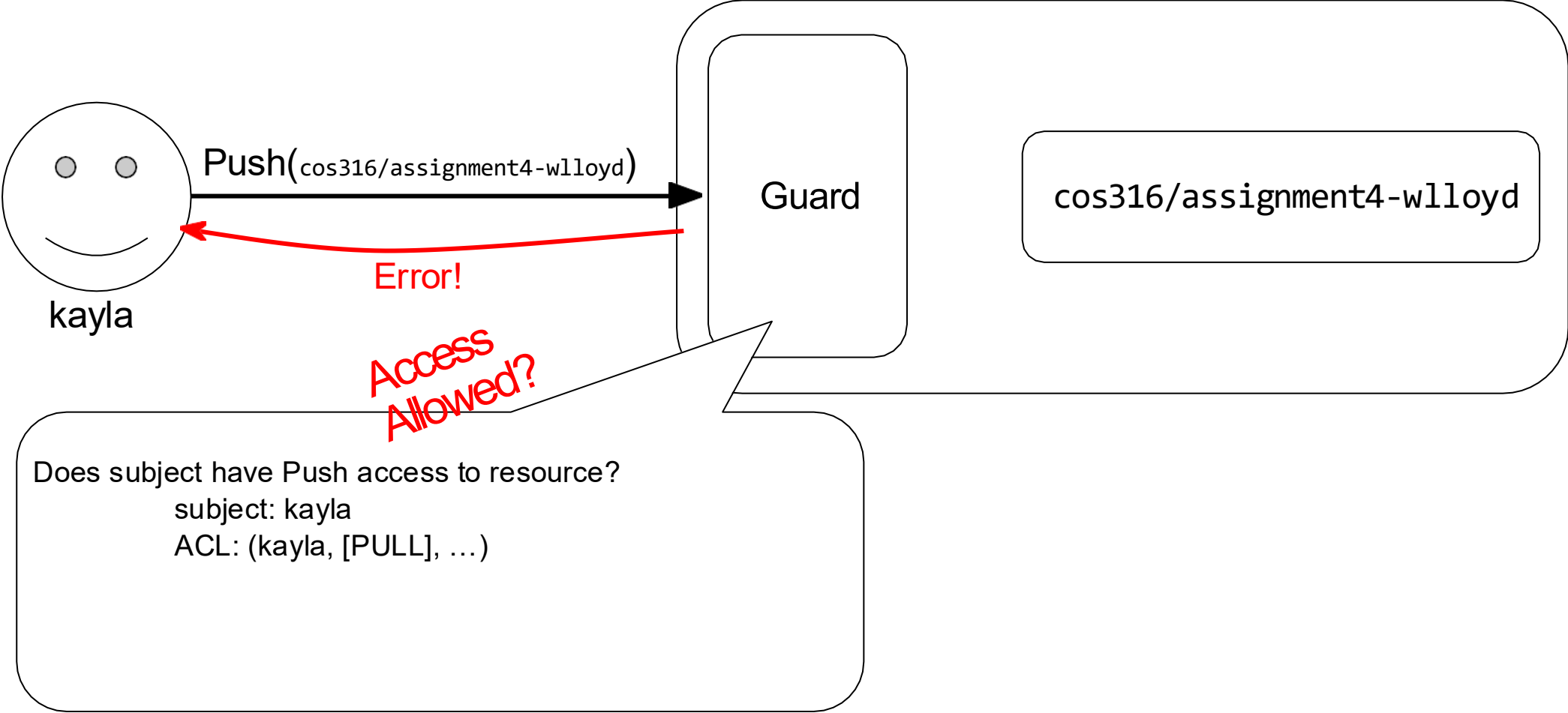
- Associate a list of (user, permissions) with each resource



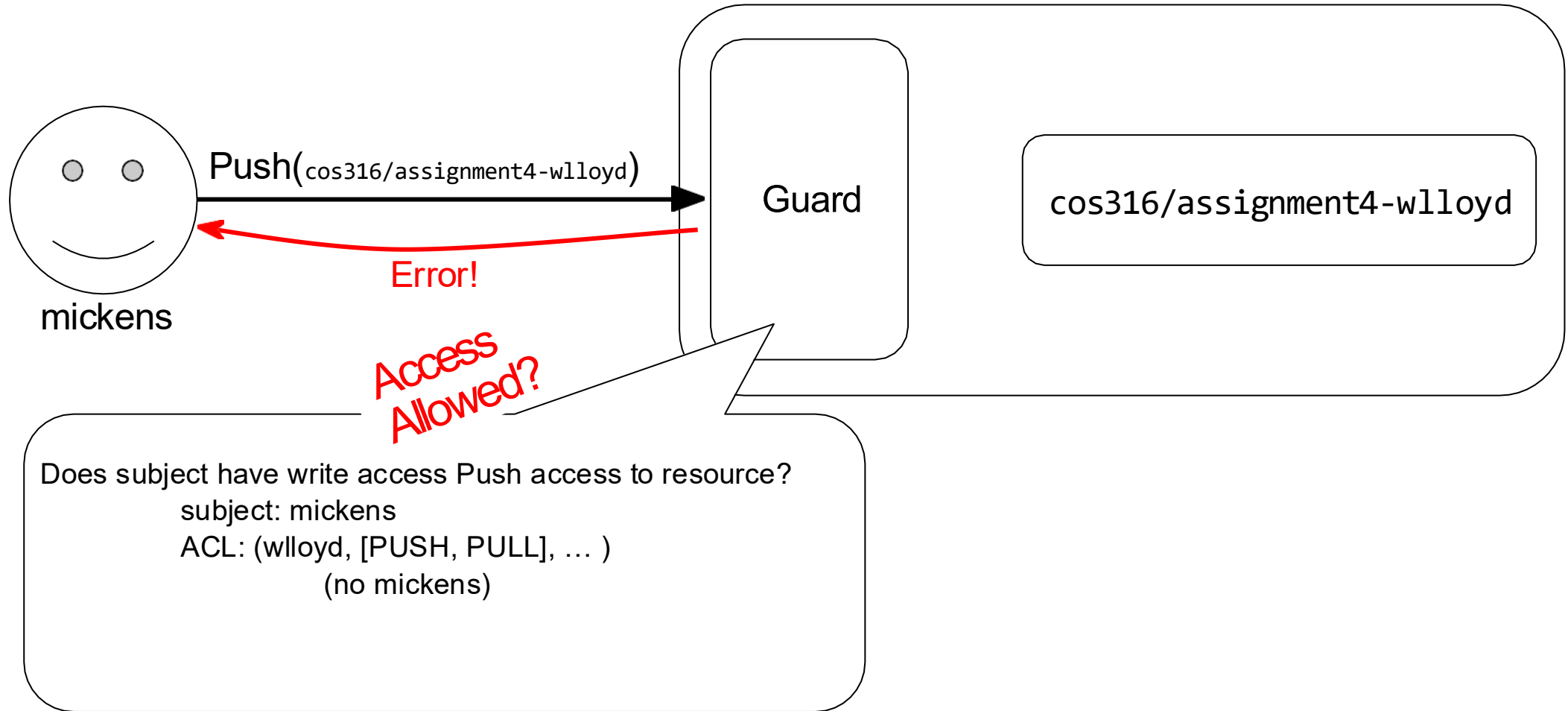
ACLs in Action



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, etc)
 - Ideally can delegate subset of capabilities
- Unforgeable
 - Subjects cannot create capabilities with arbitrary permissions

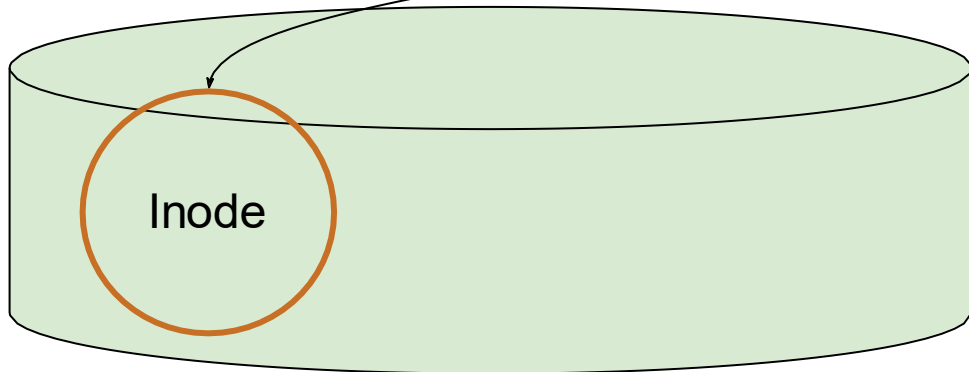
File Descriptors as Proto-Capabilities

FD

5

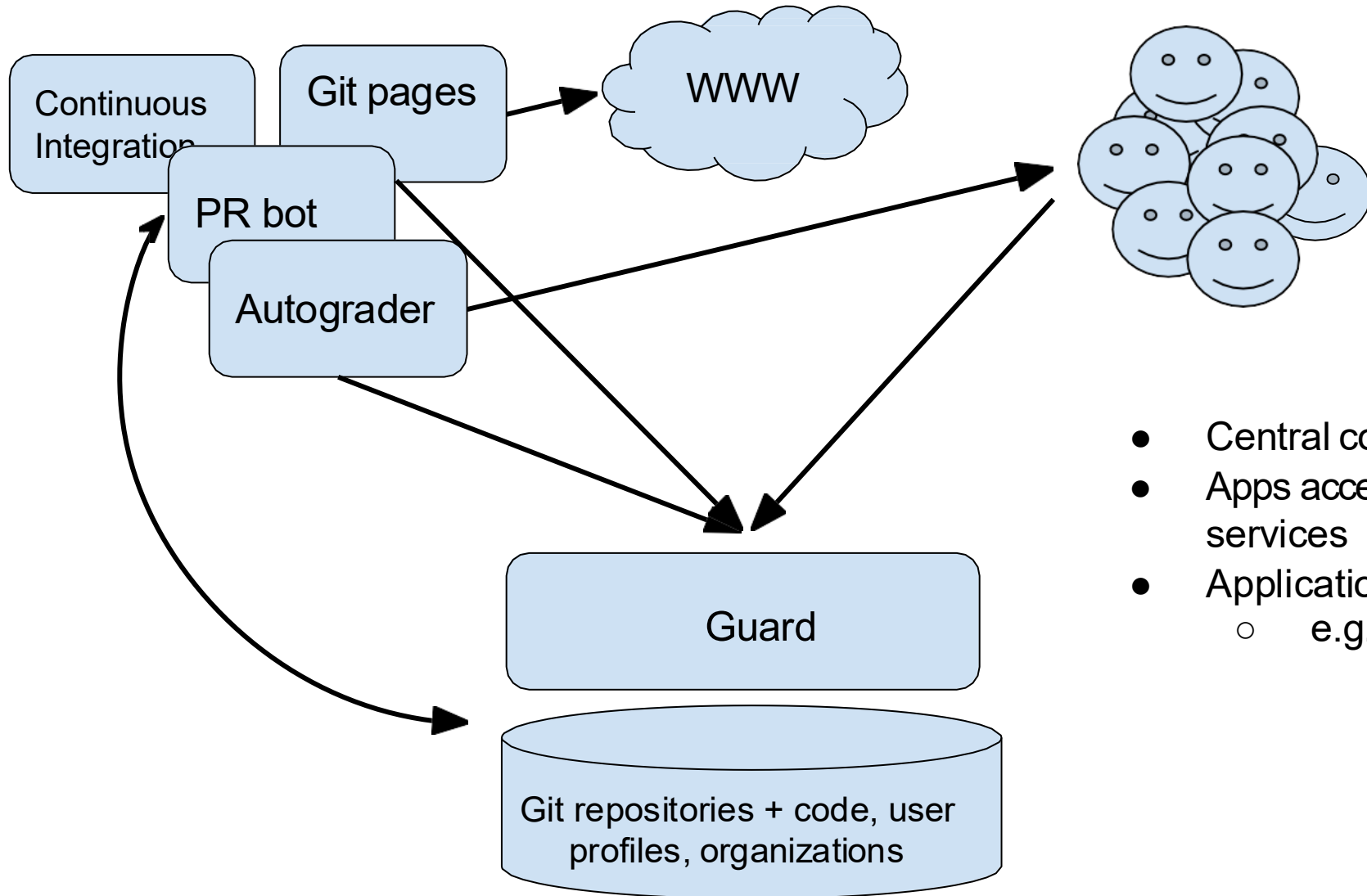
Process
Kernel

FDtable



- 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/recvmmsg

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)”

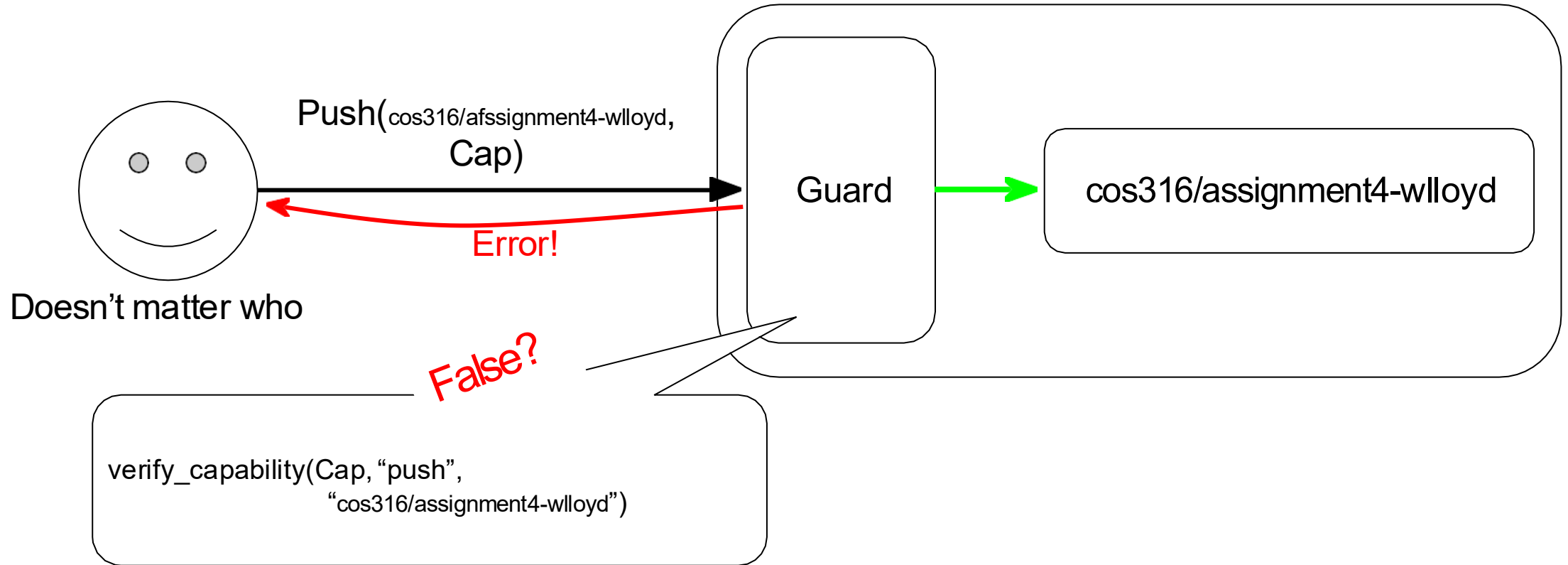
Implementing Capabilities with HMAC

- 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))  
}
```

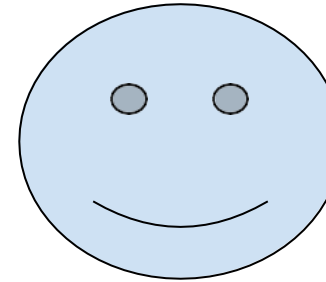
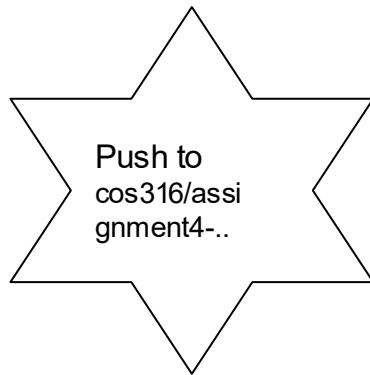
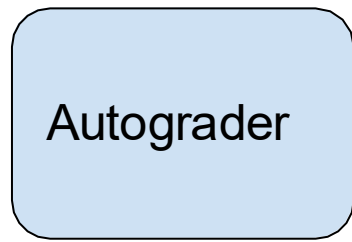
```
fn verify_capability(cap, op, repo) {  
    cap == hmac(db_secret, fmt.Sprintf("%s(%s)", op, repo))  
}
```

Capabilities in Action

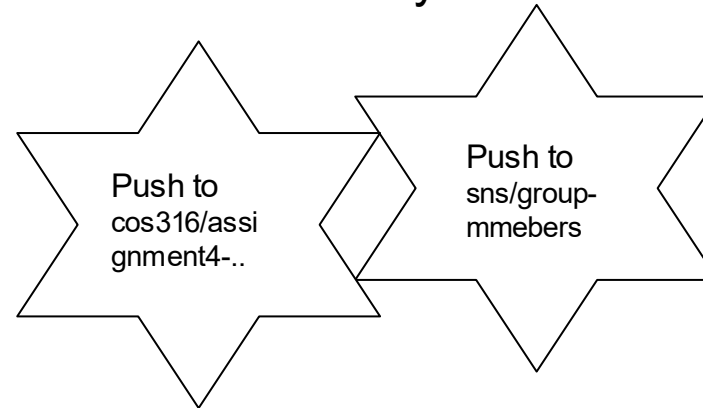


Extending Capabilities to Applications

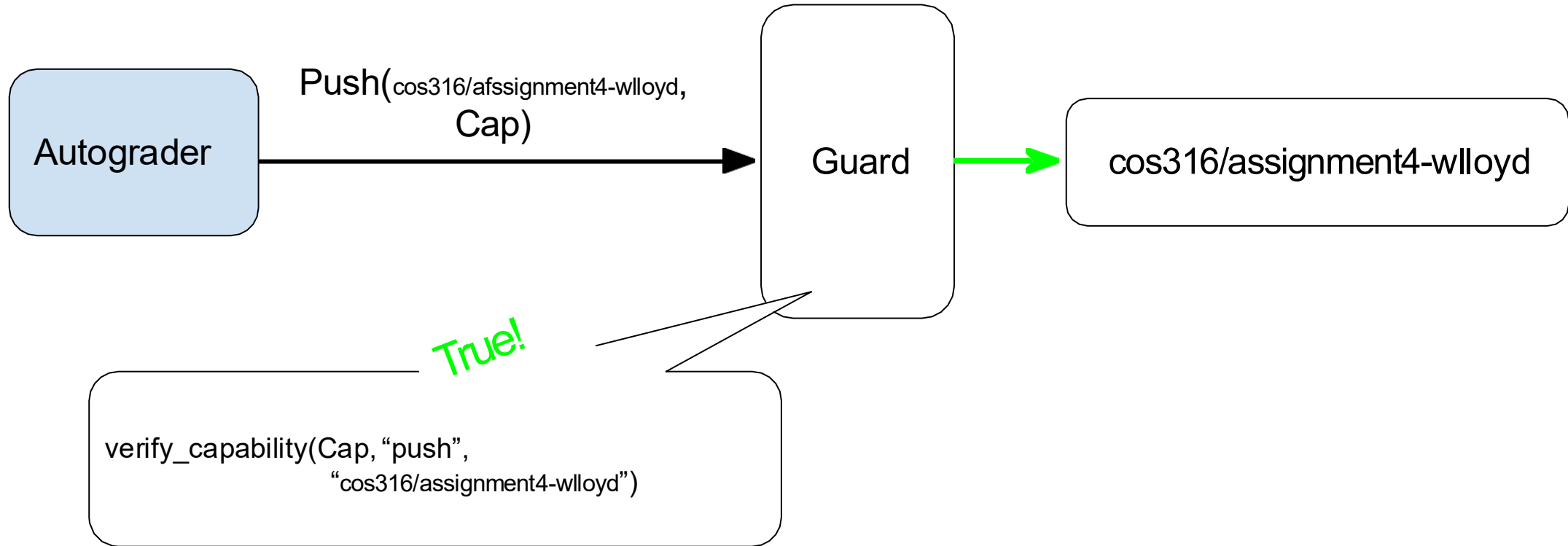
- Users can simply give applications a subset of their capabilities



wlloyd



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

Summary

- Access control is a reflection of 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...

