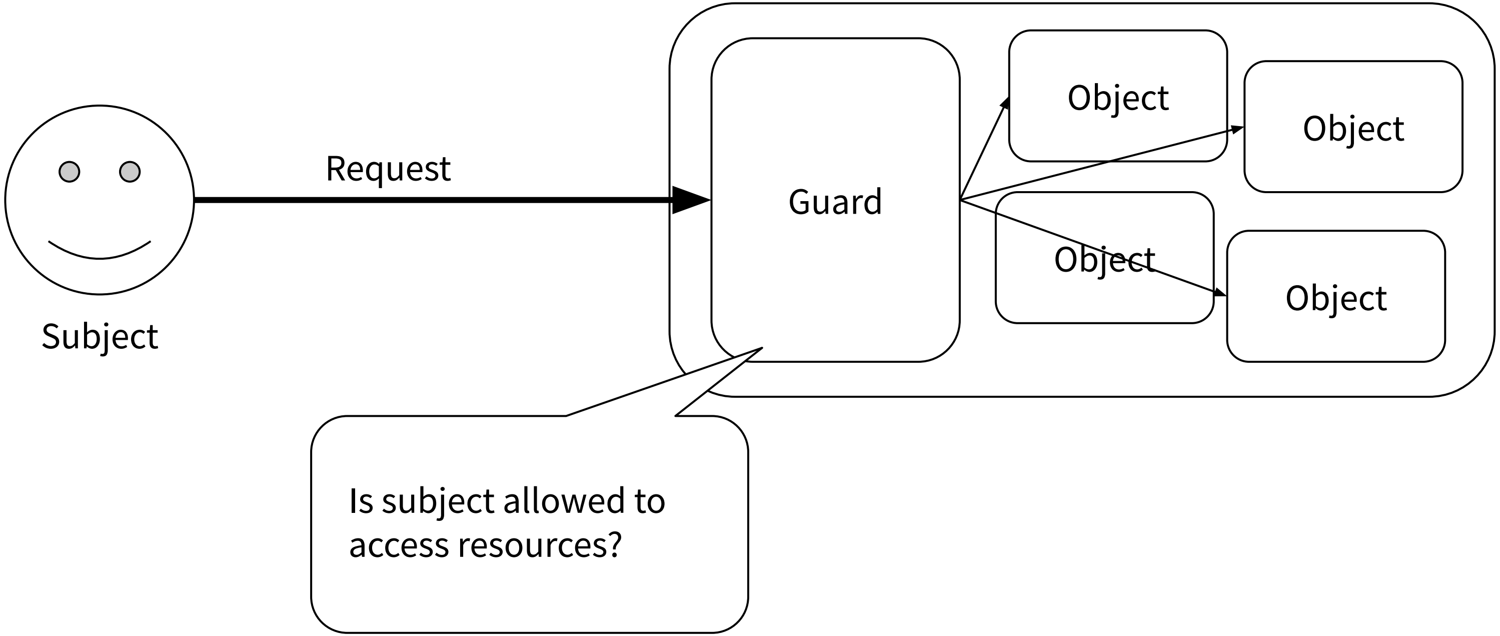
Discretionary Access Control

COS 316

The Guard Model



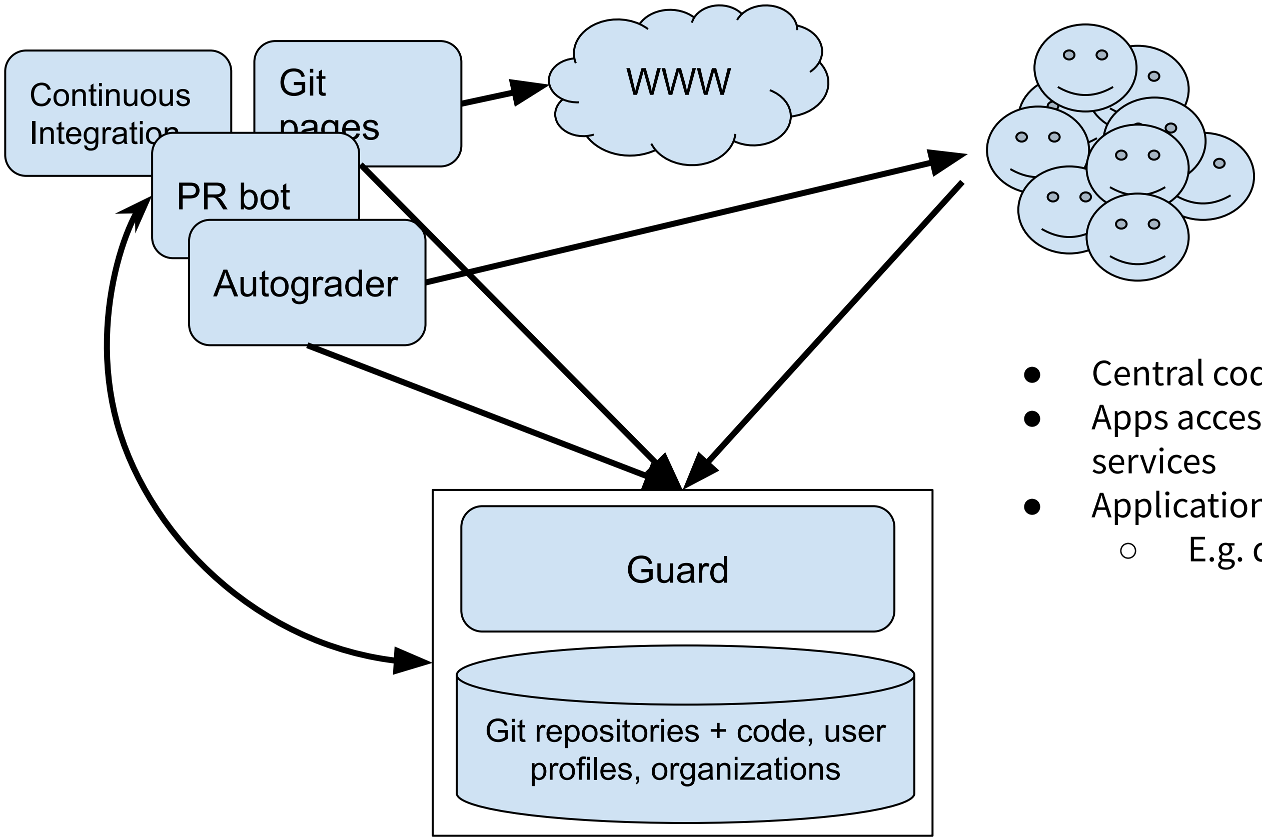
The Guard Model

A mechanism, leaves us with many questions:

- What kinds of rules does the guard enforce?
- Who gets to set or change the rules?
- What is the granularity of subjects and objects?
- Who gets to create new principals?

Answers to these questions help determine the expressivity, performance, and security of the system.

Consider a GitHub-like Ecosystem

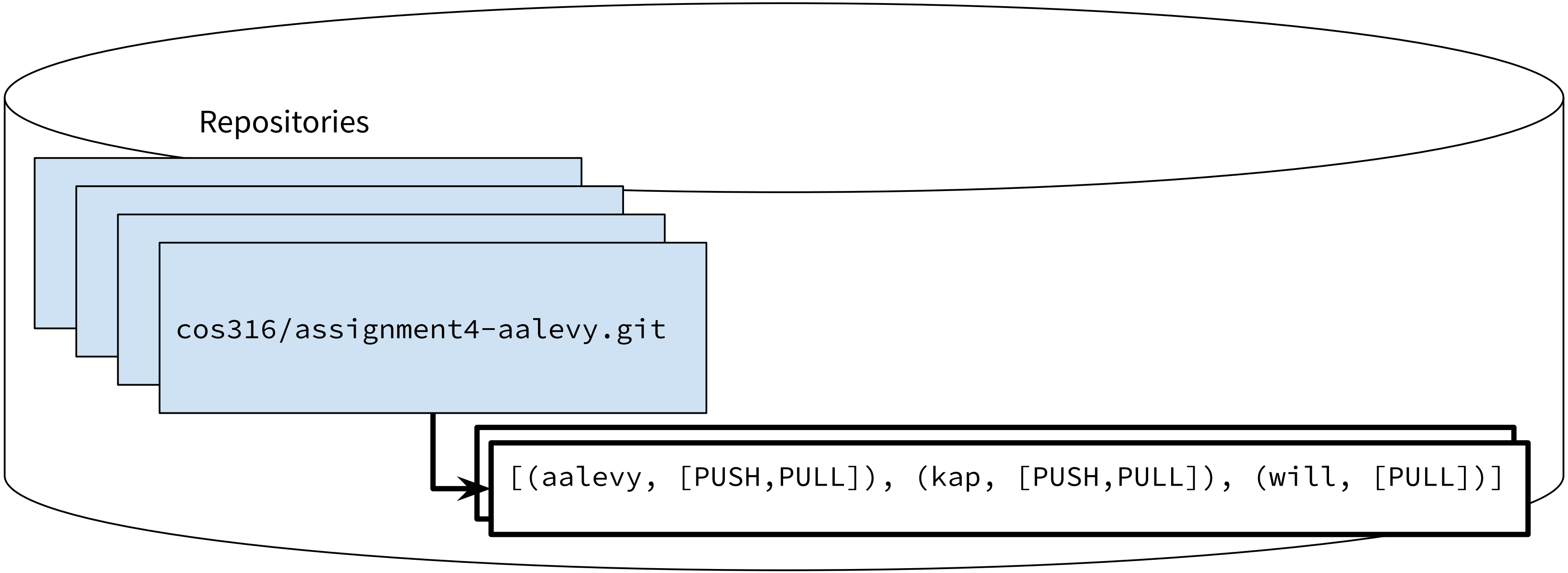


- Central code DB
- Apps access DB resources to provide extra services
- Application access must be restricted:
    - E.g. don't make private repos public

Access Control Lists (ACLs)

Let's Start with User Permissions

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



Implementing ACLs: Inline with Object

Repository Table

id	name	language	acl
1	cos316/assignment4-aalevy	Golang	“[(aalevy, [PUSH,PULL]), (kap, [PUSH,PULL]), ...]”
2	tock/tock	Rust	...
...	...	...	...

Implementing ACLs: Normalize

ACL Table

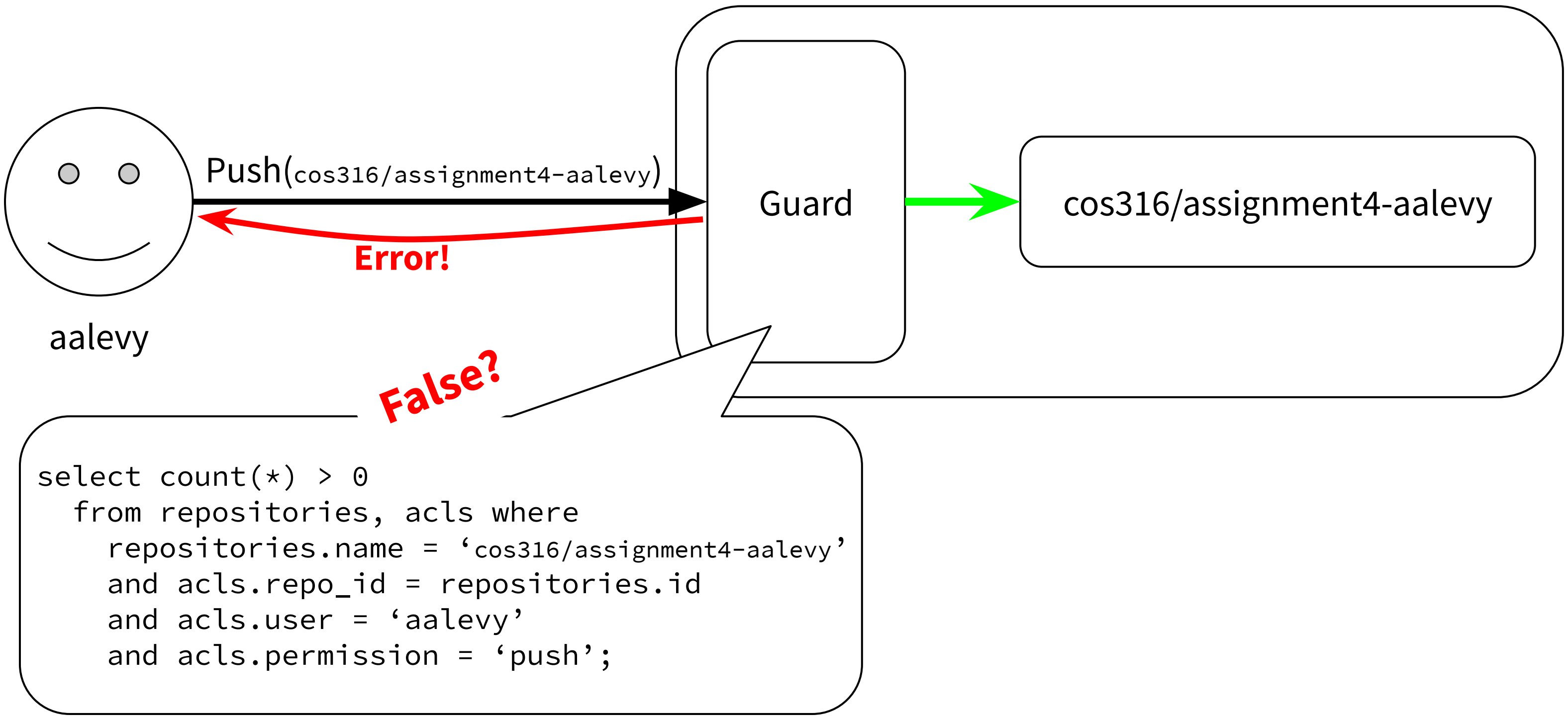
repo_id	user	permission
1	aalevy	push
1	kap	push
1	kap	pull
1	aalevy	pull
1	will	pull
2	aalevy	push
...	...	...

```
select (acls.user, acls.permission)
  from repositories, acls where
    repositories.name = ‘cos316/assignment4-aalevy’
    and acls.repo_id = repositories.id;
```

Repository Table

id	name	language
1	cos316/assignment4-aalevy	Golang
2	tock/tock	Rust
...	...	...

ACLs in Action



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
- Why might this be suboptimal?

Extending ACLs to Apps: Special Principles

- Create a unique principles for each app
    - E.g., the “autograder” principle
    - 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 user’s access
        - 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 X Repos X Access Right entries in ACL table
- Centralized access control
  - Needs server’s cooperation to delegate access

Capabilities

User Permissions using Capabilities

Hand out communicable, unforgeable tokens encoding:

- Object
- Access right

Users store capabilities, not the database

E.g.

“`push(cos316/assignment4-aalevy)`”

“`pull(cos316/assignment4-aalevy)`”

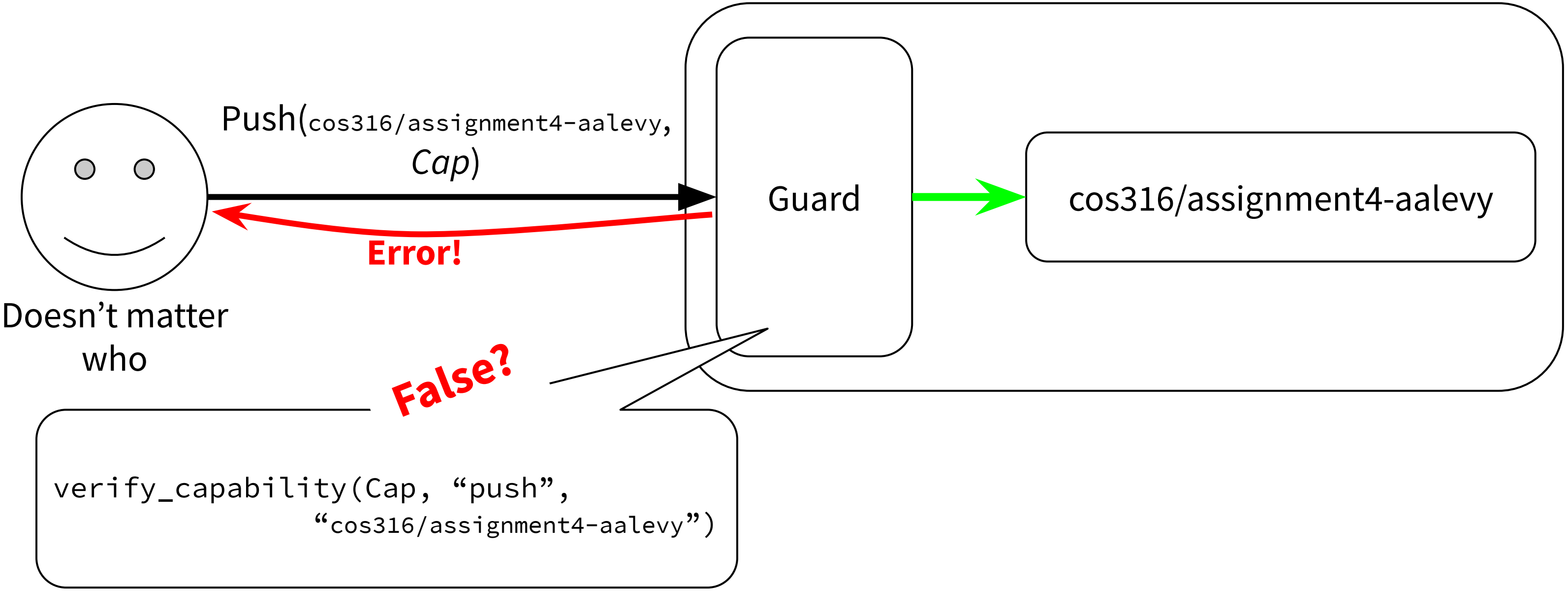
Implementing Capabilities with HMAC

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))
}
```

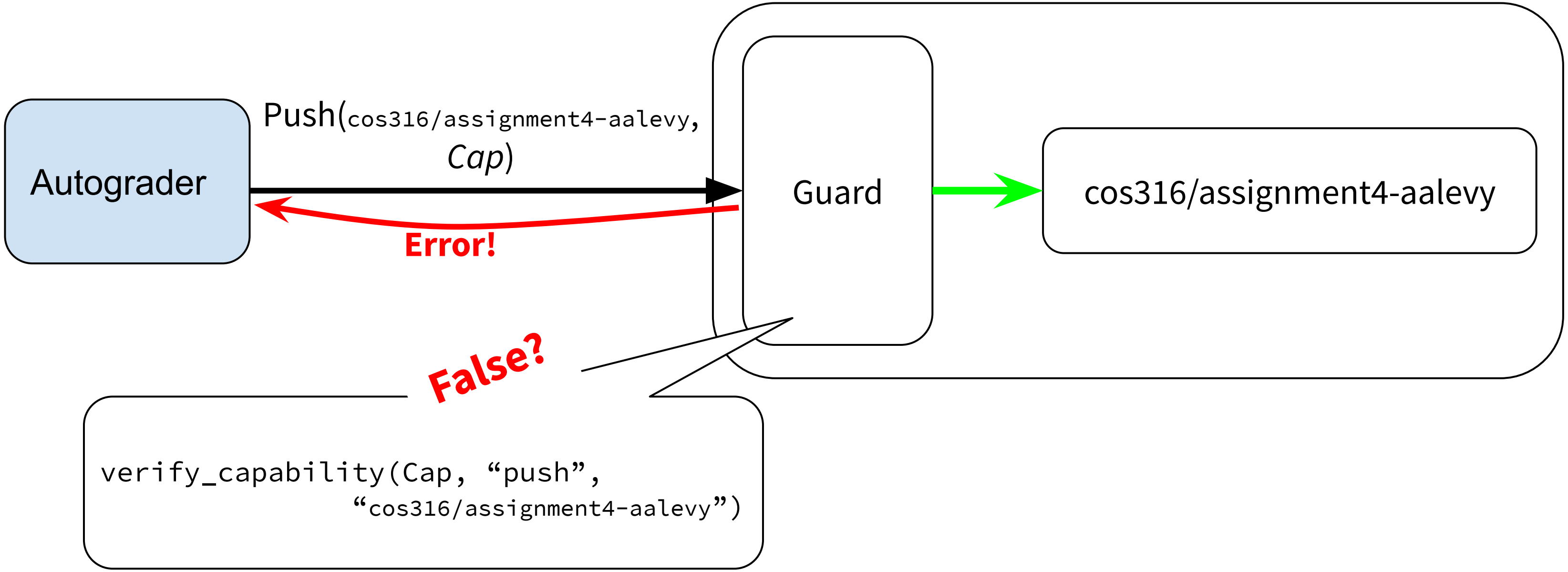
Capabilities in Action



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 have to manage their capabilities now

Next time...

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?