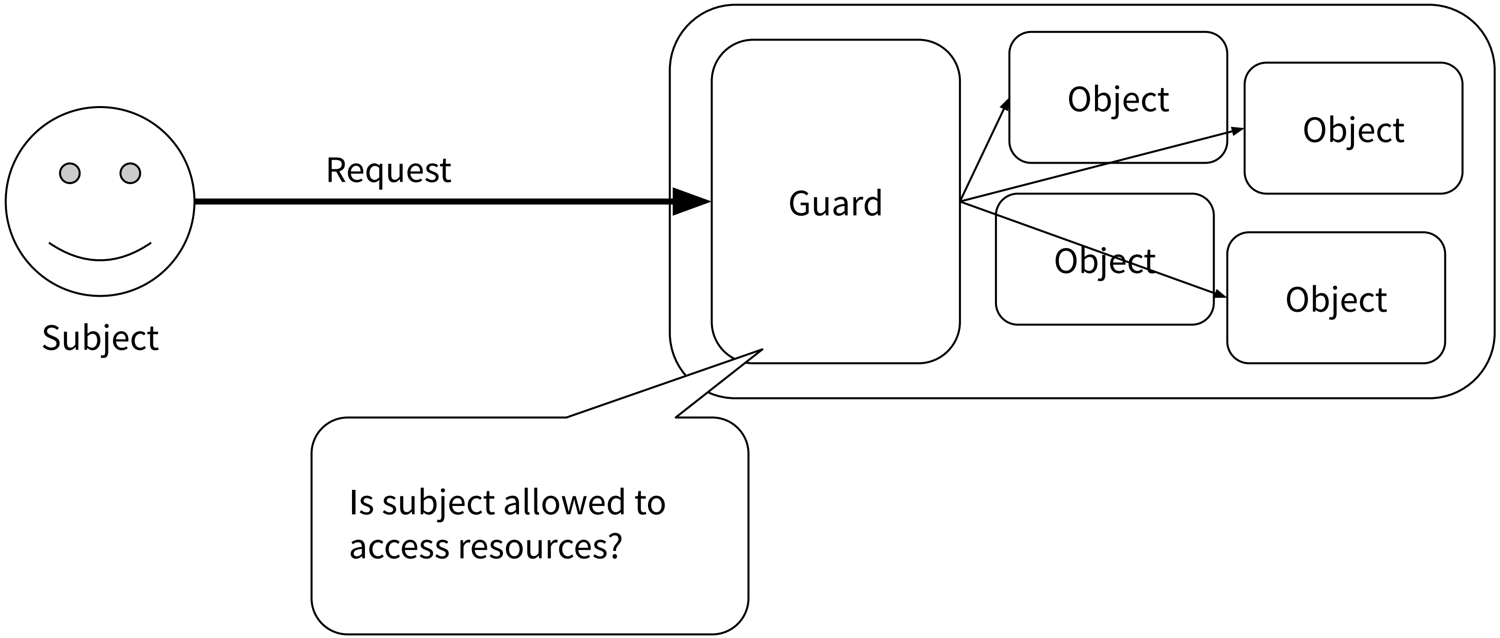
Access Control

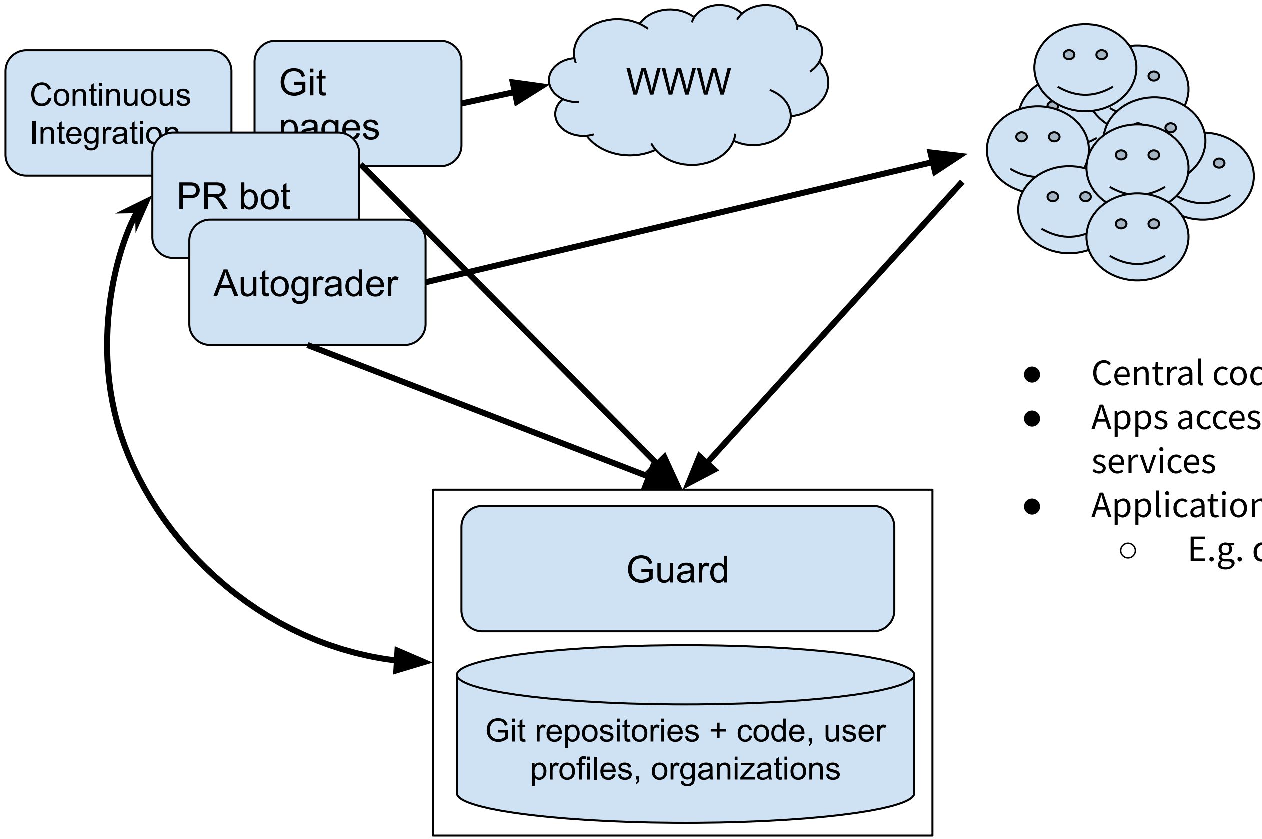
COS 316: Principles of Computer System Design

Amit Levy & Ravi Netravali

Last Time - The Guard Model



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

Discretionary Access Control

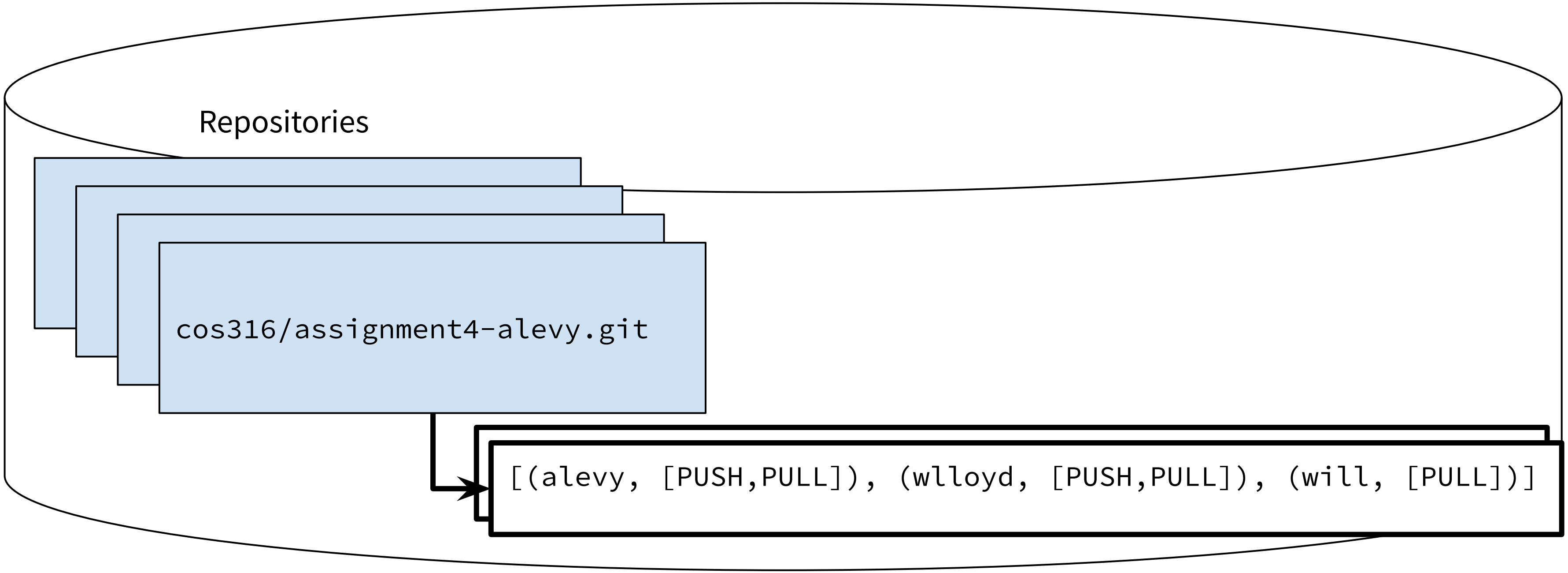
Discretionary Access Control - *[Access] controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).*

- Trusted Computer System Evaluation Criteria, 1985 (the "Orange Book")

- Access Control Lists
  - Restrict access to objects based on the identity of subjects
  - Subjects can pass object contents after reading it
- Capabilities
  - Restrict access to objects based on possession of a capability

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	“[(alevy, [PUSH,PULL]), (wlloyd, [PUSH,PULL]), ...]”
2	tock/tock	Rust	...
...	...	...	...

Implementing ACLs: Normalize

ACL Table

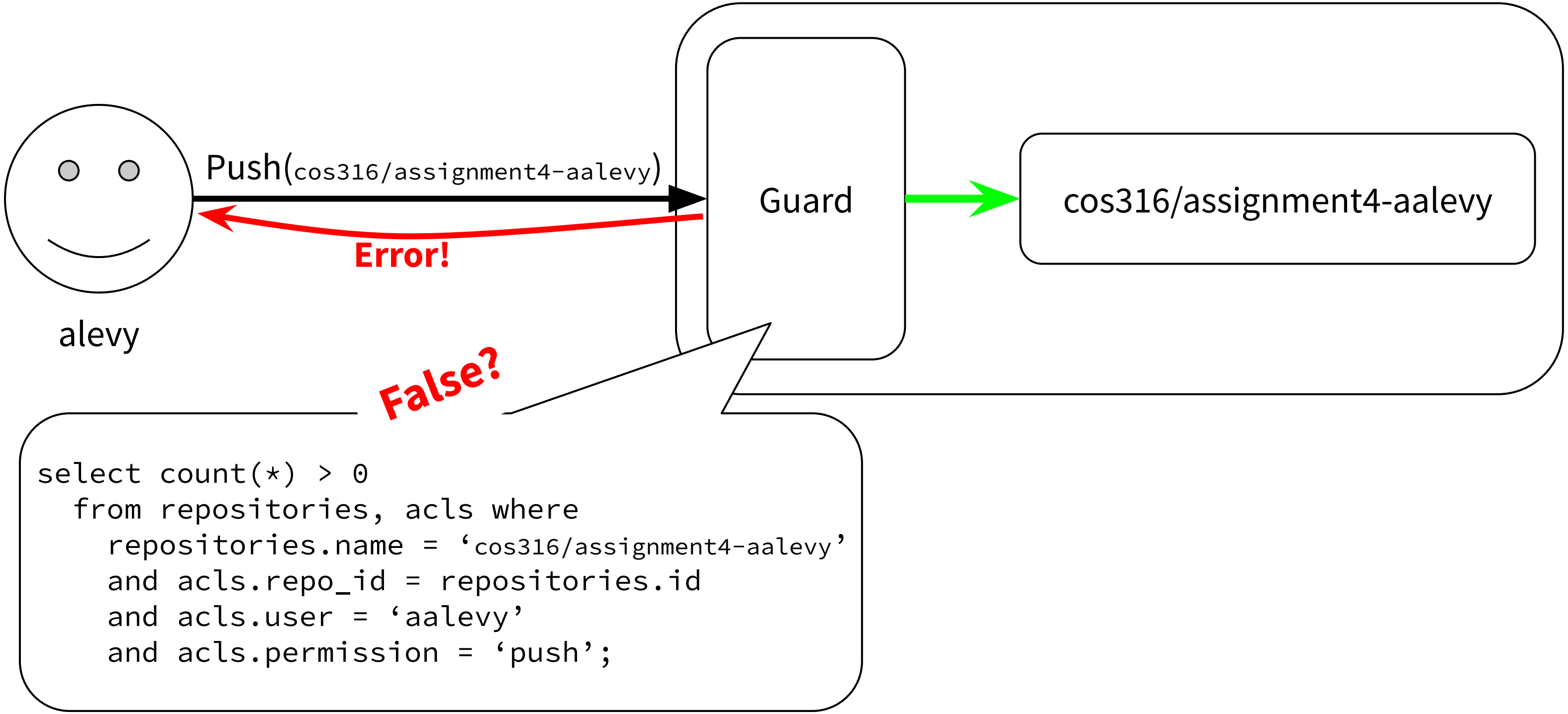
repo_id	user	permission
1	aalevy	push
1	kap	push
1	kap	pull
1	aalevy	pull
1	will	pull
2	aalevy	push
...	...	...

```
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 one 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

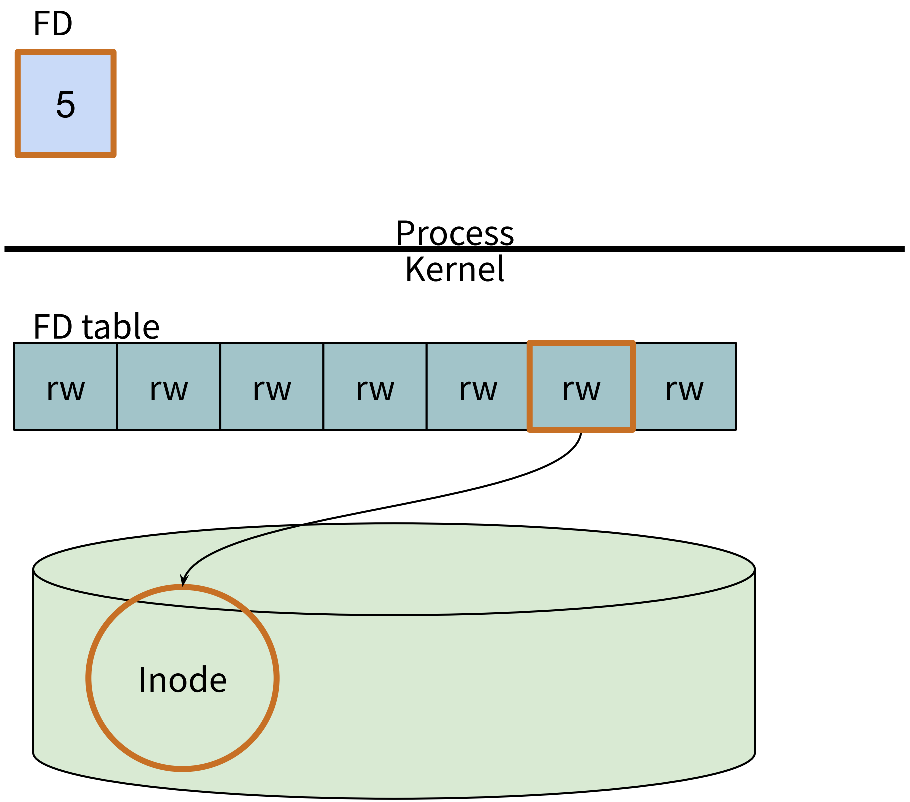
- Tradeoff granularity for simplicity
  - More granular permissions require more complex rules in the guard
- Doesn’t scale well
  - E.g. need up to Users X Repos X Access Right entries in ACL table
- Centralized access control
  - Needs server’s cooperation to delegate access

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*

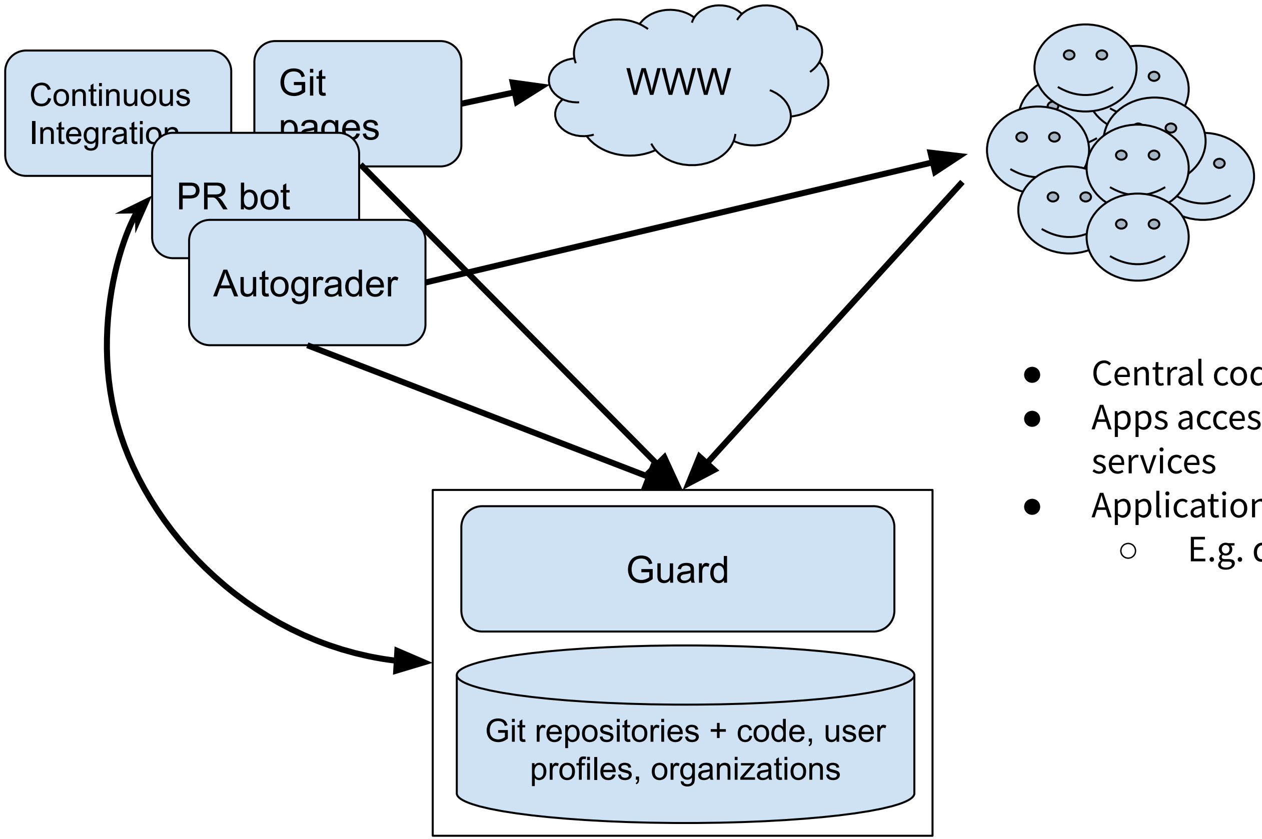
- Self-describing
  - Contains both object name and permitted operations
- Globally meaningful
  - Object and operation names are not subject-specific
- Transferrable
  - A subject can pass a capability to another (e.g. a sub-process, via IPC, a third-party app, etc)
  - Ideally can delegate subset of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

File Descriptors as Proto-Capabilities



- 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-aalevy)`**”

“**`pull(cos316/assignment4-aalevy)`**”

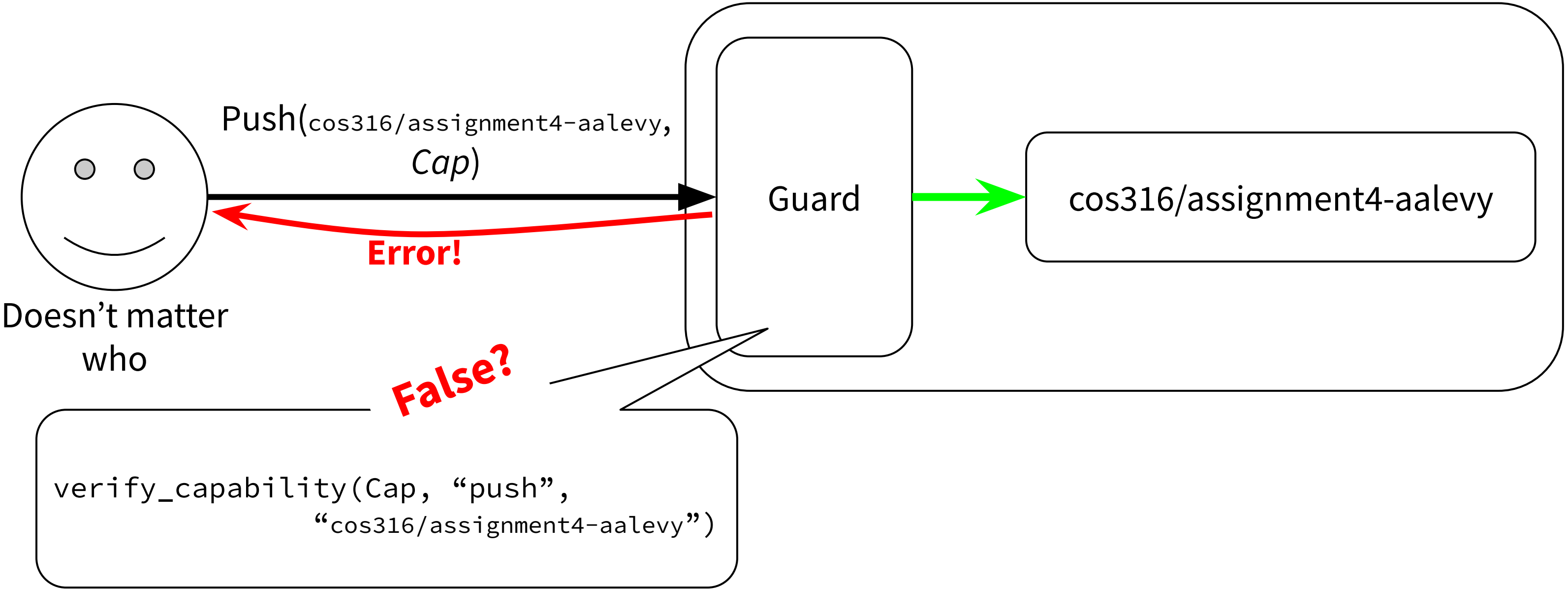
Implementing Capabilities with HMAC

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))
}
```

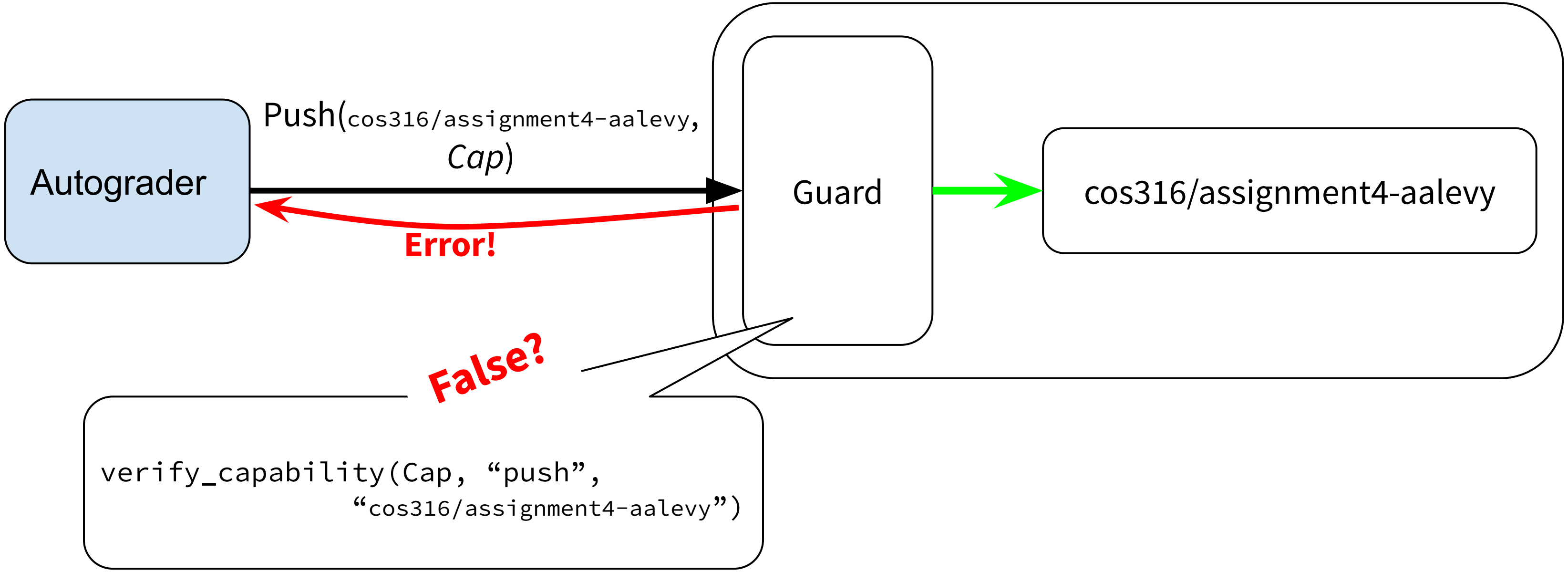
Capabilities in Action



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

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

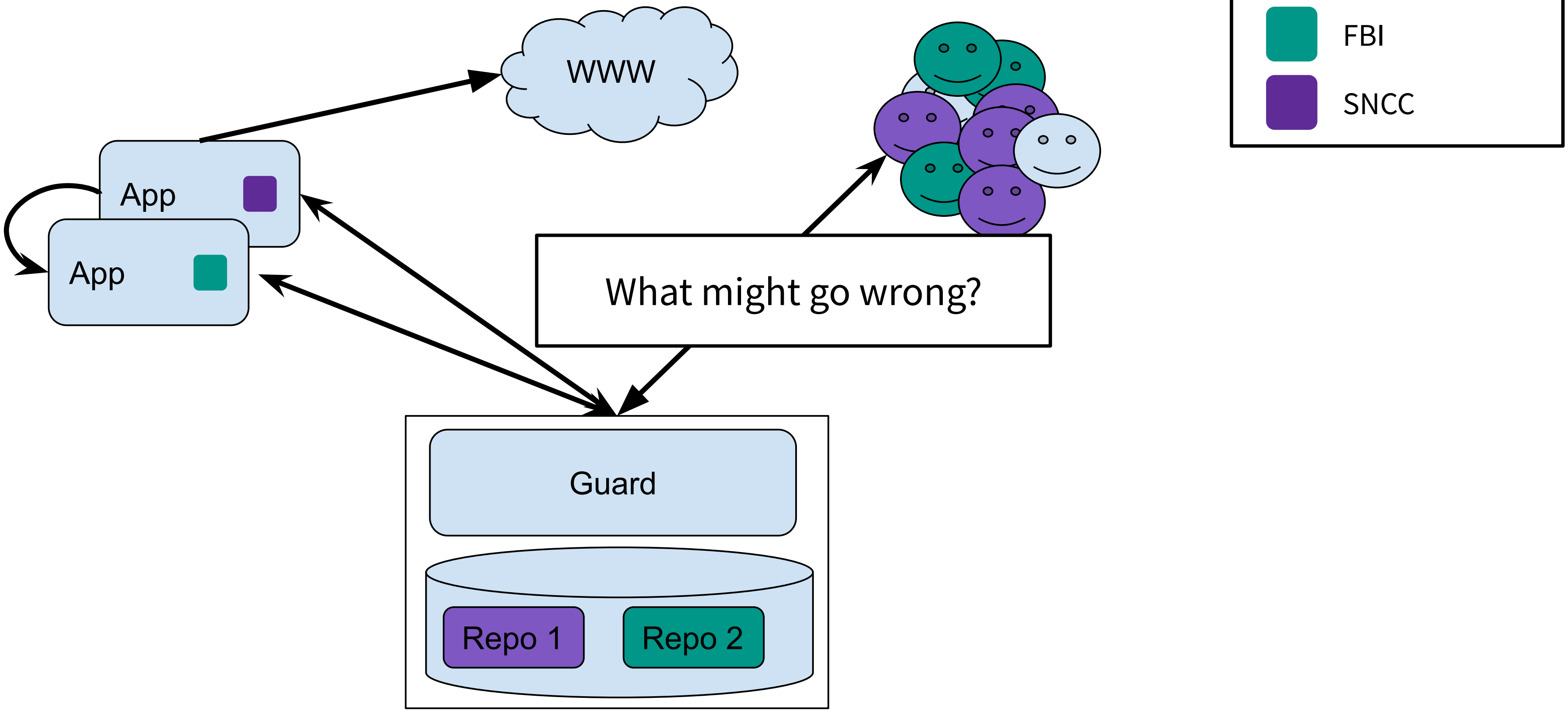
We Still Have a Problem

The autograder is allowed to:

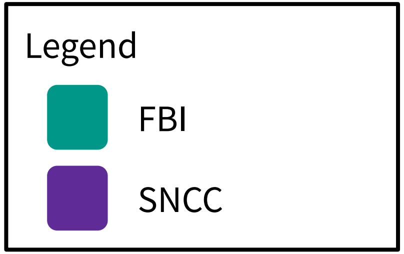
- read all cos316/ repositories
- comment on all cos316/ repositories

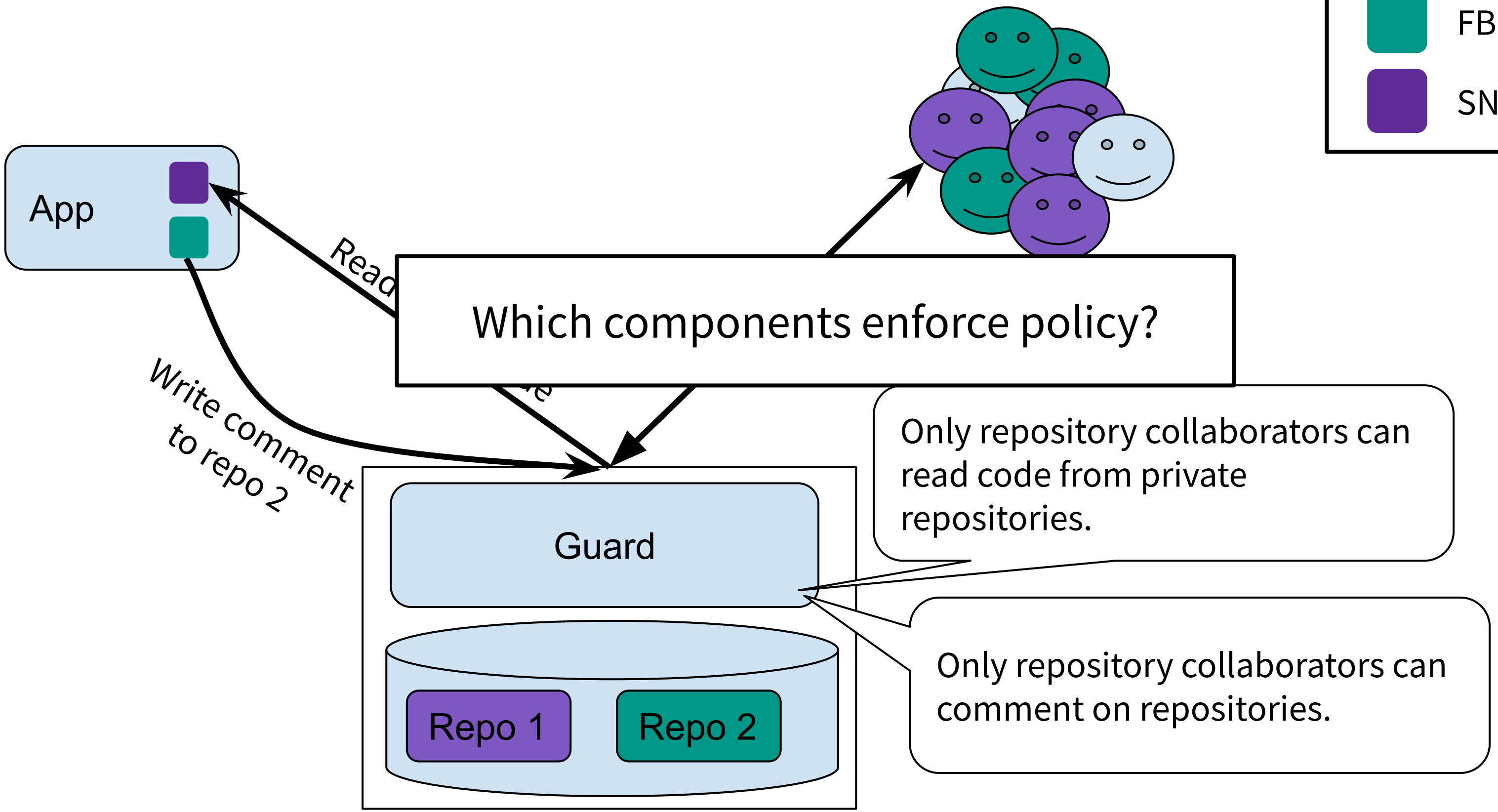
Can code from a private repository end up in a comment on a public repository?

Who enforces policy under DAC?

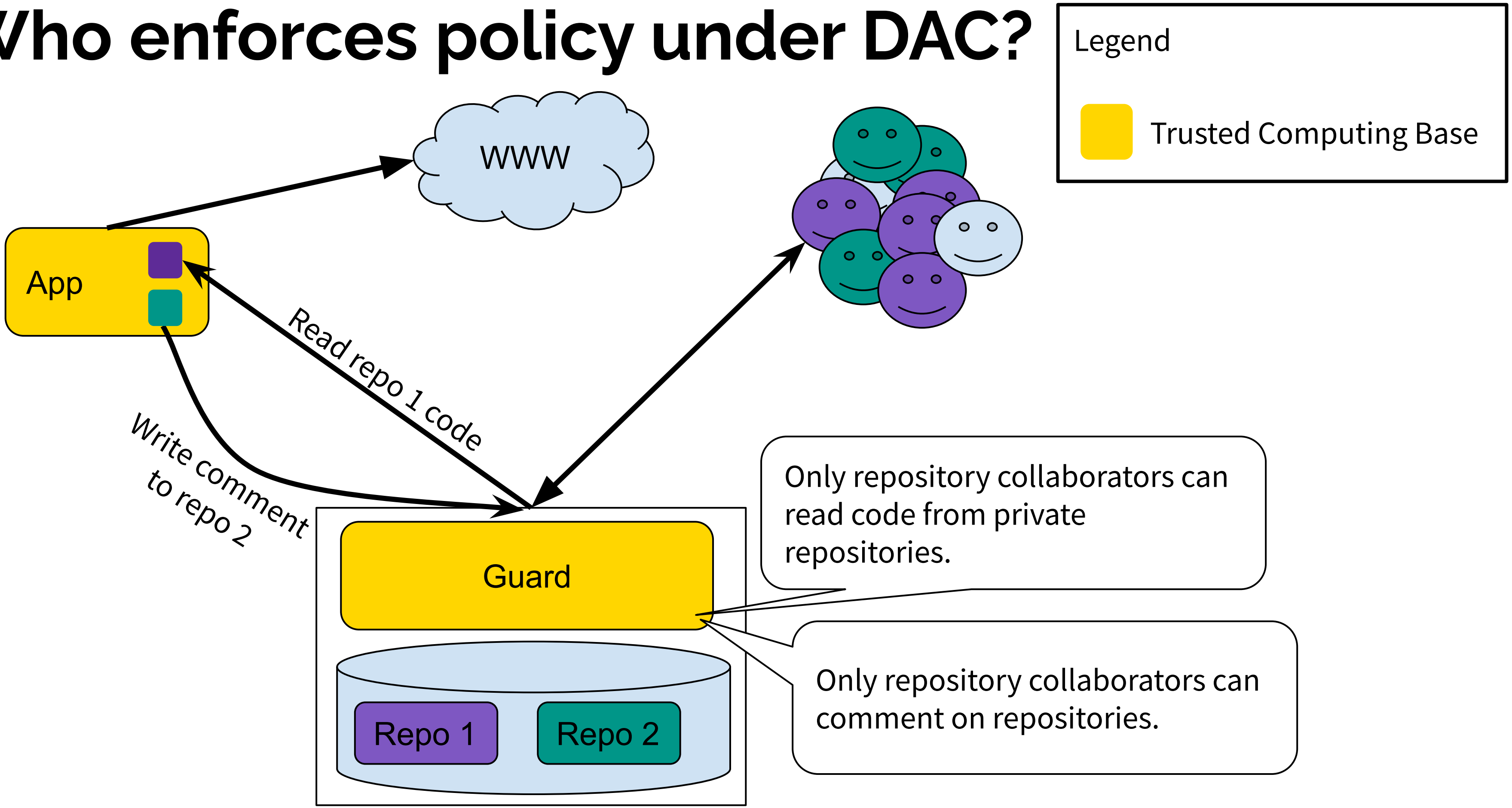


Who enforces policy under DAC?





Who enforces policy under DAC?



Limitations of Discretionary Access Control

- Discretionary means a *subject* with access to an *object* can propagate information:
  - In UNIX, owners determine read/write/execute access for themselves, group, and “other”
  - Subject can pass capabilities to anyone
  - UNIX process reads `~/.ssh/ida_rsa` and writes output to public log
  - Can’t (trivially) revoke capabilities
- This is one reason it’s sufficient to compromise a single high privilege application, rather than whole system, in order to extract private data

The non-interference property

Informally:

A program is non-interferent if it’s transformations of data in low security domains (*low*) are not influenced by data in higher security domains (*high*)

The non-interference property

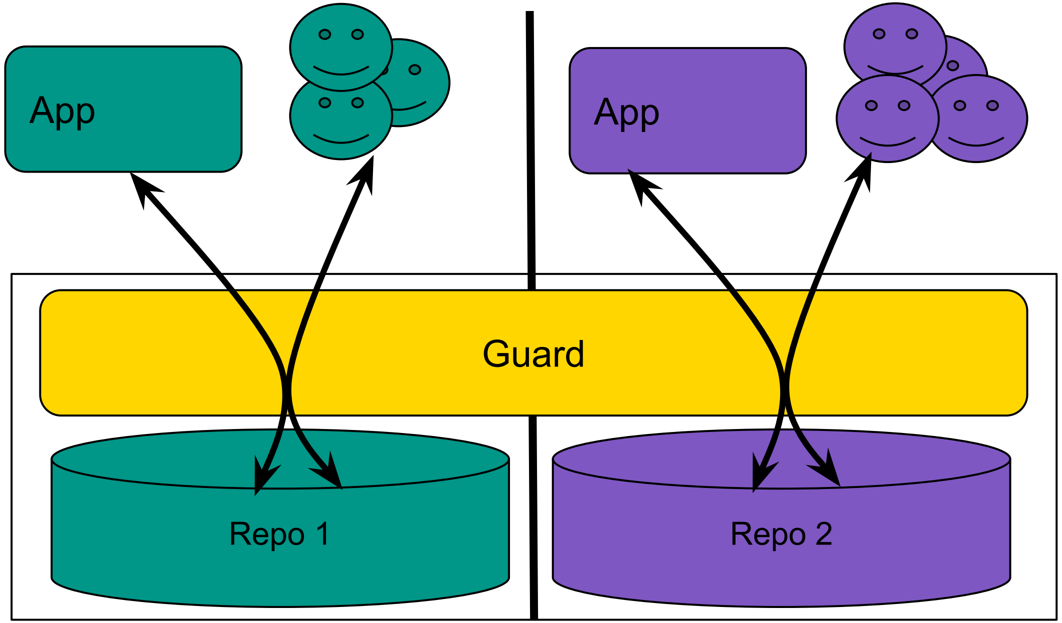
M, a memory state including *low* and *high* memory, M_H and M_L, respectively

P: $(M) \rightarrow M^*$, a program execution over a memory state resulting in a new memory state, is non-interferent if:

$$\forall\, M1, M2 \text{ s.t. } M1_L = M2_L$$
$$\land\ P(M1) \rightarrow M1^*$$
$$\land\ P(M2) \rightarrow M2^*$$
$$\Rightarrow\ \boldsymbol{M1^*_L = M2^*_L}$$

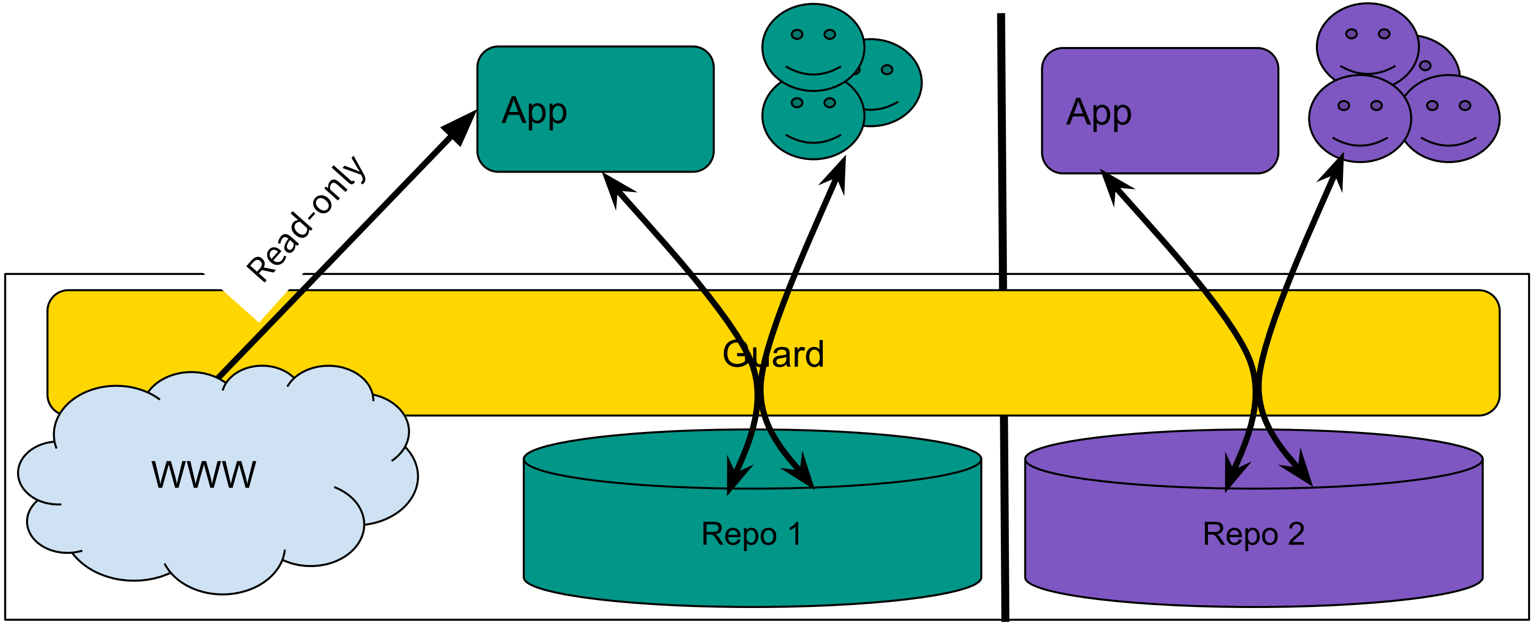
Enforcing Non-Interference with DAC

Discretionary Access Control policies can enforce non-interference by completely partitioning the system



Enforcing Non-Interference with DAC

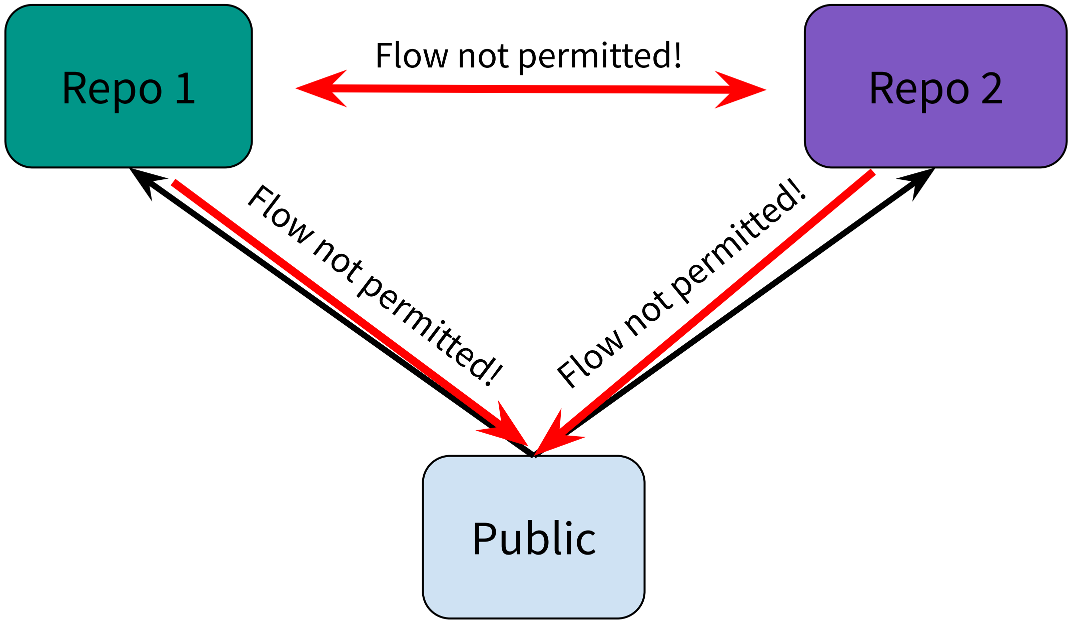
Discretionary Access Control policies can enforce non-interference by completely partitioning the system, or with careful, static sharing



Mandatory Access Control (MAC)

- Goal: data secrecy & integrity don't rely on trusting applications *at all*
- All resource accesses governed by a global policy
- Subjects cannot change global policy
- Typically policy articulated in terms of data sources and sinks
- E.g.
  - *label* data with it's sensitivity
  - define permitted flows between labels
  - Permit operations as long as information flow rules are not violated

A simple security label lattice



Implementing MAC

There are very few MAC systems used *in practice*:

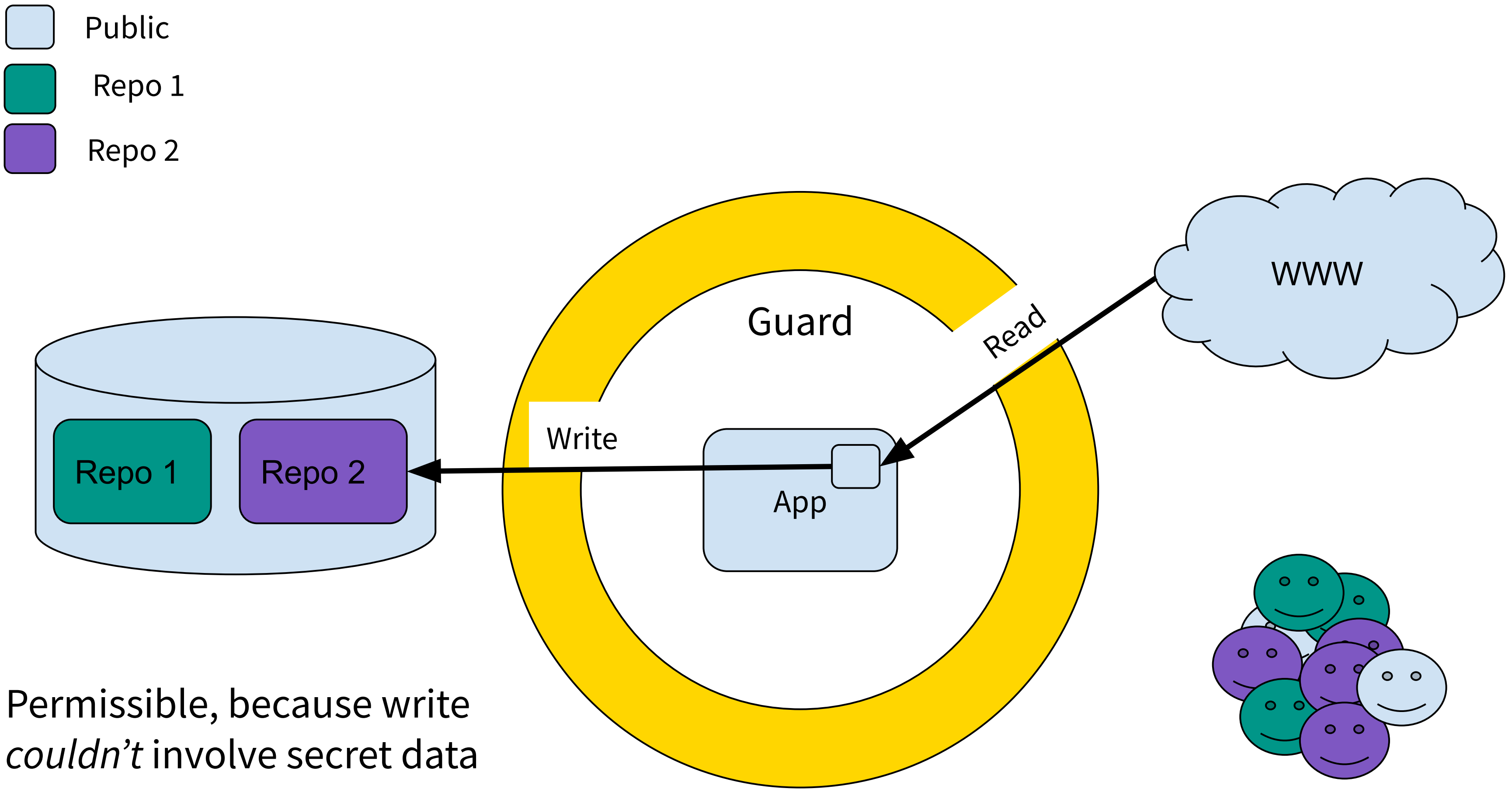
- SELinux - an extension to Linux originating from the NSA
    - Used in Android
- Mandatory Integrity Control - a Windows kernel subsystem limited to integrity
- TrustedBSD (in development)
- …

But lots of *research* systems

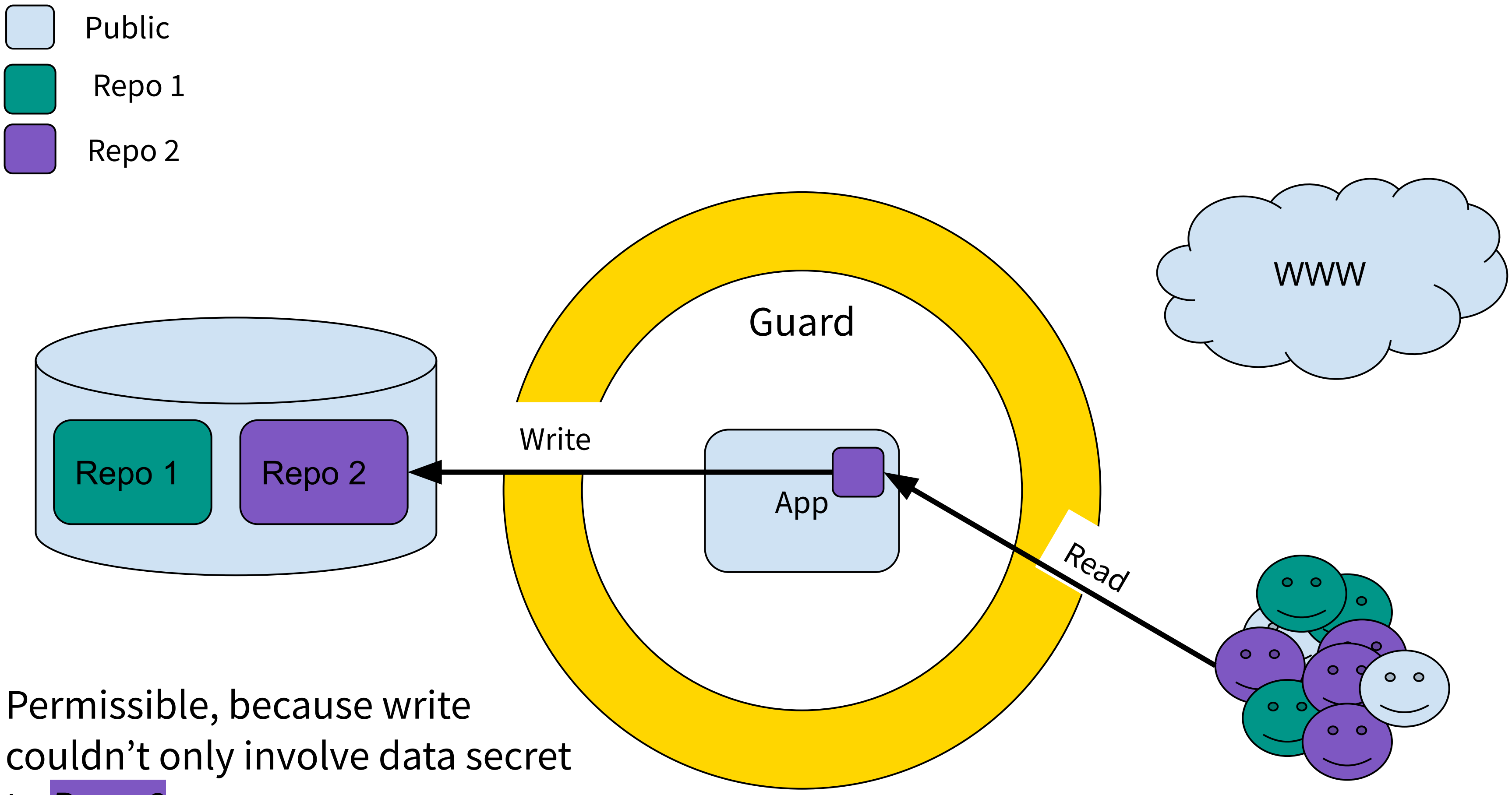
Implementing MAC

One general approach:

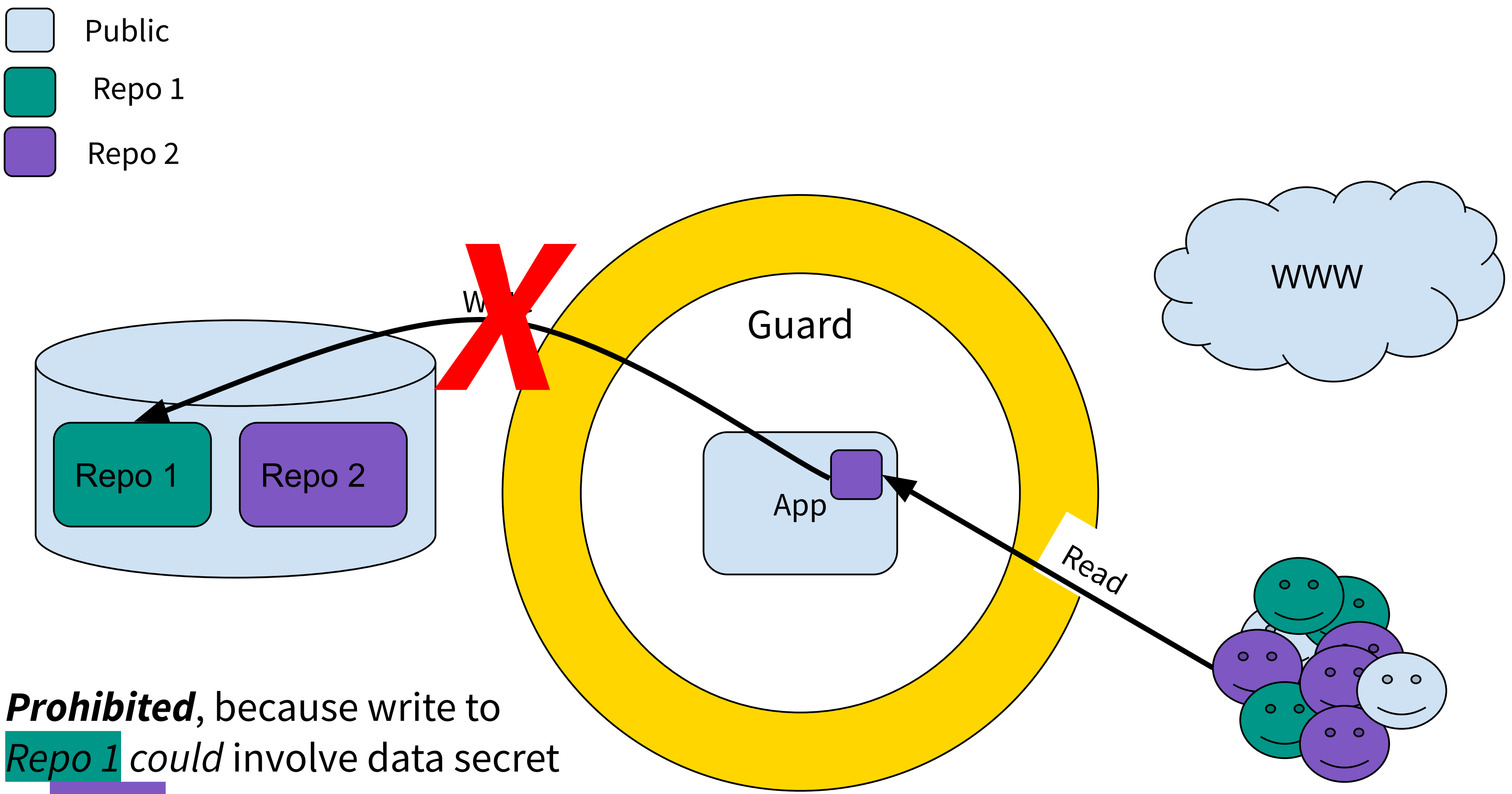
- Assign a security label to object (file, network endpoint, console, etc)
- Assign a *floating* label to subjects (running processes)
    - “Floating” because it changes dynamically
- Whenever moving/copying data, check that source label *can flow to* sink label
- Allow subject to “raise” its floating label, but not to “lower” it



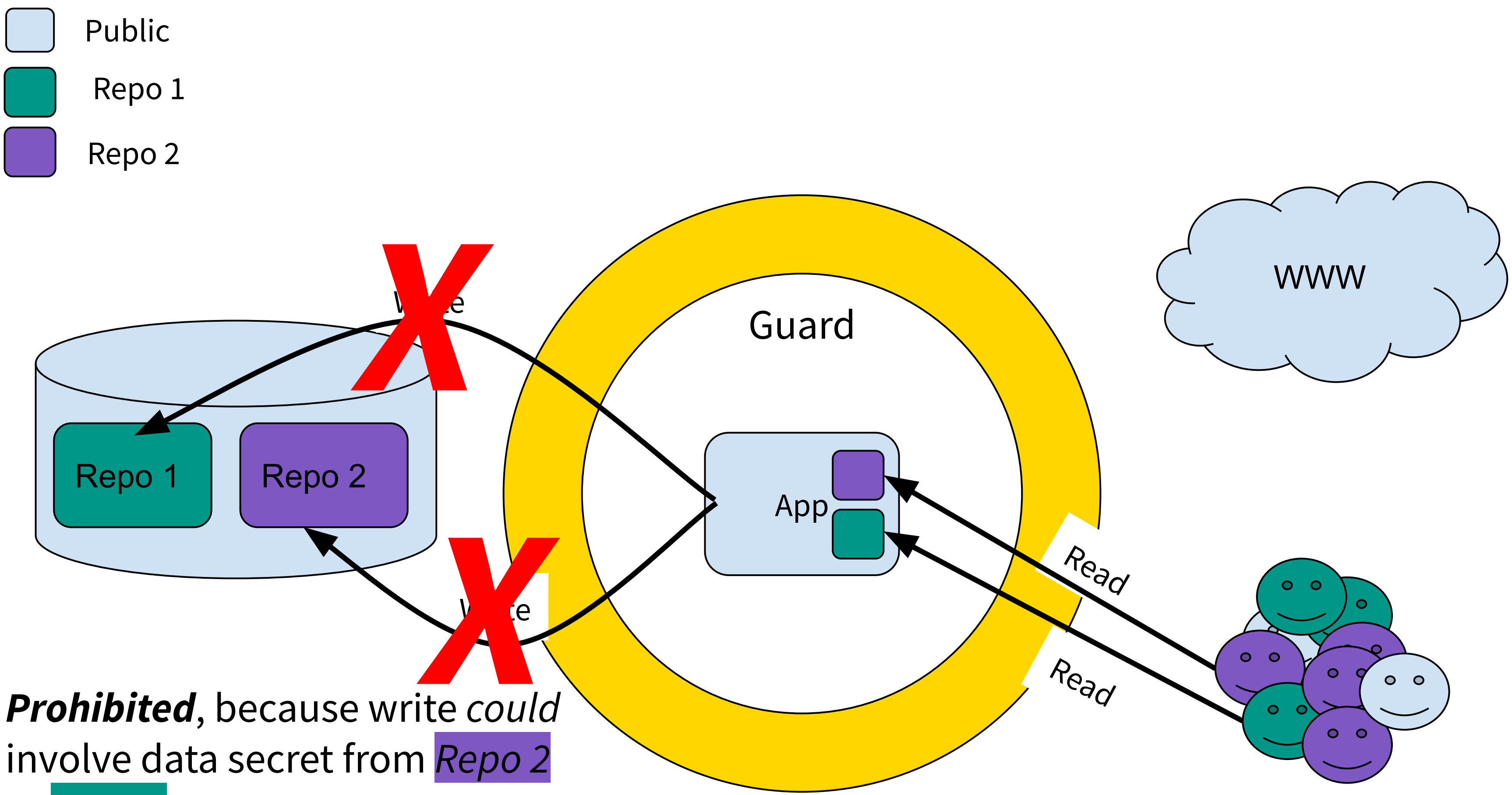
Permissible, because write *couldn't* involve secret data



Permissible, because write couldn’t only involve data secret to *Repo 2*



Prohibited, because write to *Repo 1* *could* involve data secret to *Repo 2*



Prohibited, because write *could* involve data secret from *Repo 2* or *Repo 1*

Mandatory Access Control in Practice

- Dates back to at least 1983
  - Defined in the DoDs *Trusted Computer System Evaluation Criteria* (aka the Orange Book)
- Very powerful guarantee!
  - Security policies on data *do not* rely on application correctness
- Why is it not more prevalent?

Why isn't MAC more prevalent?

- Complexity: implementing MAC can be hard to get right
- Performance: lattice checks can be slow
- Flexibility: by design, applications cannot get around security policy
- Simplicity: MAC is harder to administer