Discretionary Access Control

COS 316
The Guard Model

Subject

Request

Guard

Object

Object

Object

Object

Is subject allowed to access resources?
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

- Continuous Integration
- Git pages
- PR bot
- Autograder

Guard

- 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

Repositories

cos316/assignment4-aalevy.git

[(aalevy, [PUSH,PULL]), (kap, [PUSH,PULL]), (will, [PULL])]
## Implementing ACLs: Inline with Object

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>language</th>
<th>acl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cos316/assignment4-aalevy</td>
<td>Golang</td>
<td>“[(aalevy, [PUSH,PULL]), (kap, [PUSH,PULL]), ...]”</td>
</tr>
<tr>
<td>2</td>
<td>tock/tock</td>
<td>Rust</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementing ACLs: Normalize

Repository Table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cos316/assignment4-aalevy</td>
<td>Golang</td>
</tr>
<tr>
<td>2</td>
<td>tock/tock</td>
<td>Rust</td>
</tr>
</tbody>
</table>

ACL Table

<table>
<thead>
<tr>
<th>repo_id</th>
<th>user</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aalevy</td>
<td>push</td>
</tr>
<tr>
<td>1</td>
<td>kap</td>
<td>push</td>
</tr>
<tr>
<td>1</td>
<td>kap</td>
<td>pull</td>
</tr>
<tr>
<td>1</td>
<td>aalevy</td>
<td>pull</td>
</tr>
<tr>
<td>1</td>
<td>will</td>
<td>pull</td>
</tr>
<tr>
<td>2</td>
<td>aalevy</td>
<td>push</td>
</tr>
</tbody>
</table>

```sql
select (acls.user, acls.permission)
from repositories, acls
where
repositories.name = 'cos316/assignment4-aalevy'
and acls.repo_id = repositories.id;
```
ACLs in Action

```
select count(*) > 0
from repositories, acls
where
    repositories.name = 'cos316/assignment4-aalevy'
    and acls.repo_id = repositories.id
    and acls.user = 'aalevy'
    and acls.permission = 'push';
```
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: \( \text{hmac(secret\_key, data)} \) hash of data

```rust
def gen_capability(op, repo):
    hmac(db_secret, fmt.Sprintf("%s(%s)", op, repo))
```

```rust
def verify_capability(cap, op, repo):
    cap == hmac(db_secret, fmt.Sprintf("%s(%s)", op, repo))
```
Capabilities in Action

Doesn't matter who

\[ \text{Push}(\text{cos316/assignment4-aalevy}, \text{Cap}) \]

Guard

\[ \text{cos316/assignment4-aalevy} \]

Error!

False?

\[ \text{verify\_capability}(\text{Cap}, \text{"push"}, \text{"cos316/assignment4-aalevy"}) \]
Extending Capabilities to Applications

- Users can simply give applications a subset of their capabilities
Extending Capabilities to Applications

```
verify_capability(Cap, "push", "cos316/assignment4-aalevy")
```

Error!
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?