Access Control

COS 316: Principles of Computer System Design

Amit Levy & Ravi Netravali
Why might we want to control access to resources?
The Complete Guide to Facebook Privacy

Despite repeated privacy lapses, Facebook offers a fairly robust set of tools to control who knows what about you.

Facebook has never been particularly good at prioritizing your privacy. Your data powers its business, after all. But recent revelations that a firm called Cambridge Analytica harvested the personal information of 50 million unwitting Facebook users in 2015 has created new sense of urgency for those hoping for some modicum of control over their online life. If you ever needed a wake-up call, this is it.

The good news: Despite the repeated, public privacy lapses, Facebook does offer a fairly robust set of tools to control who knows what about you—both on the platform and around the web. The bad news: Facebook doesn’t always make those settings easy to find, and they may not all offer the level of protection you want.

Fear not! Below, we’ll walk you through the steps you need to take to keep advertisers, third-party apps, strangers, and Facebook itself at bay. And if after all that you still feel overly exposed? We’ll show you how...
Election hoax spreading through text messages in Michigan

The text claims a ‘typographical error’ is switching peoples’ votes

By Zoe Schiffer | @ZoeSchiffer | Nov 3, 2020, 2:16pm EST

A text message campaign is targeting people in Michigan with misinformation about “ballot stuffing.”

Illustrator by Alex Castro / The Verge
Why might we *not* want to control access to resources?
Boundless Informant: the NSA's secret tool to track global surveillance data

Revealed: The NSA's powerful tool for cataloguing global surveillance data - including figures on US collection

- Boundless Informant: mission outlined in four slides
- Read the NSA's frequently asked questions document

![Map of global surveillance data]

The color scheme ranges from green (least subjected to surveillance) through yellow and orange to red (most surveillance). Note the '2007' date in the image relates to the document from which the interactive map derives its top-secret classification, not to the map itself.
Access control design is a subtle craft that must be informed by real-world, human, considerations.
A (slightly) formal model

- **Objects**: the things being accessed
  - A file, database table, network socket, satellite imagery of “nuclear facilities,” missile launcher...
- **Subjects**: an entity that requests access to an object
  - A process, network endpoint, etc…
  - **Principal**: some unique account or role, such as a user
- **Authentication**: a proof that a subject *speaks for* some principal
  - E.g. logging in with a username & password
- **Authorization**: the particular rules that govern subjects’ access to objects
- **Secrecy**: who might learn the contents of an object
- **Integrity**: who may have influenced the contents of an object
Ad-hoc access control

- Access policy enforcement is scattered throughout system

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

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

- Very common in applications with lots of users. Why?
Ad-hoc access control

- Application-specific access rules
- Data for rules stored separately from data objects
  - Really a problem of granularity

Profile Table

<table>
<thead>
<tr>
<th>id</th>
<th>full_name</th>
<th>profile_pic</th>
<th>handle</th>
<th>bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amit Levy</td>
<td>/i/1f3.png</td>
<td>alevy</td>
<td>Motivational speaker, futurist...</td>
</tr>
<tr>
<td>2</td>
<td>Wyatt Lloyd</td>
<td>/i/a60.png</td>
<td>wllloyd</td>
<td>Enjoys long function names...</td>
</tr>
</tbody>
</table>

Friends Table

<table>
<thead>
<tr>
<th>follower</th>
<th>followee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

...
Problems Ad-hoc access control

- Policy is emergent

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

- Who can view a user’s full name?

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

Subject → Request

Guard

Object → Object → Object

Is subject allowed to access resources?
Examples of the Guard Model

- **Kernel**
  - File system permissions: as long as objects modeled as files, access checks are centralized
  - Reference monitor

- **Networks**
  - Firewall
  - Apache HTTP Server’s .htaccess rules

- **Databases**
  - Table/database visibility
  - Limit ability to ALTER, UPDATE, DROP, etc
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.
What kinds of rules?

There are many “policy languages”

- Access control lists: which subjects can read/write which objects
- Capabilities: unforgeable tokens that encode specific rules on objects
  - Subjects unnamed
- Information flow: the relationship between data sources and data sinks
  - Neither subjects nor objects named, instead
Who sets the rules?

We will discuss two broad categories:

- **Discretionary Access Control (DAC)**
  - Very common, e.g. UNIX user/group permissions
- **Mandatory Access Control (MAC)**
  - Pretty uncommon, much more robust
  - E.g. SE-Linux & AppArmore, and lots of research systems
Granularity

Why doesn’t database just re-use UNIX file permissions?

- The objects in UNIX file permissions are *files*, with read/write/execute permissions
- But...
- Tables & schemas might span many files
- Databases might include several schemas or tables in a single file
- Alter, update, drop don’t map well to read/write/execute
  - E.g. UPDATE should retain layout of data in a file
Granularity

Why doesn’t web application re-use database permissions

Profile Table

<table>
<thead>
<tr>
<th>id</th>
<th>full_name</th>
<th>profile_pic</th>
<th>handle</th>
<th>bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amit Levy</td>
<td>/i/1f3.png</td>
<td>aalevy</td>
<td>Dog dad, foodie, yog...</td>
</tr>
<tr>
<td>2</td>
<td>Alan Kaplan</td>
<td>/i/a60.png</td>
<td>kap</td>
<td>Enjoys long function names...</td>
</tr>
</tbody>
</table>

Friends Table

<table>
<thead>
<tr>
<th>follower</th>
<th>followee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Centralized vs. Decentralized Access Control

Why don’t web applications re-use UNIX users/groups?

- Facebook does not have a UNIX user for you on their servers. Why?
- UNIX does not allow unprivileged users to create new principals
- Web applications run as a single UNIX user, and re-implement:
  - Authentication
  - Authorization
  - Guard
  - ...
Consider a GitHub-like Ecosystem

- Continuous Integration
- Git pages
- PR bot
- Autograder
- WWW
- Guard
- Git repositories + code, user profiles, organizations

- 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

[(alevy, [PUSH,PULL]), (wlloyd, [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>“[(alevy, [PUSH,PULL]), (wlloyd, [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 Query

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

```sql
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';
```

False?

Guard

Error!
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
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 helps separate security enforcement from other functionality
- Behavior of a security system is determined by:
  - Policy rules
  - Granularity of subjects/objects
  - Mandatory vs. Discretionary
  - Centralized vs. Decentralized Principals
- Access Control Lists:
  - Common, but extremely limited
  - Next few lectures will explore more obscure but richer mechanisms