User Authentication

Determining if a user is who they say they are before giving them access.
## Four means of authentication

<table>
<thead>
<tr>
<th>Know</th>
<th>Have (token)</th>
<th>Are (static biometrics)</th>
<th>Do (dynamic biometrics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password, PIN, answers to prearranged questions</td>
<td>Smartcard, electronic keycard, physical key</td>
<td>Fingerprint, retina, face</td>
<td>Voice pattern, handwriting</td>
</tr>
</tbody>
</table>
Passwords

• Most common authentication mechanism
  ▪ User provides username and password, must match server records
    (For reference, see every computer thing ever)

• The hard parts:
  ▪ How to store passwords?
  ▪ How to communicate passwords?
    (covered later on)
Storing passwords: **Hashing**

- Given setup: store passwords in plaintext
- Threat model:
  - Database of password hashes is compromised (happens a LOT)
  - Attacker wants to figure out password
- Improvement: **Hashing**
  - Don’t store the plaintext password, store a hash
  - Compare hashes
- Why?
  - So the attacker can’t just look at the database and see passwords
Storing passwords: **Salting**

- Given setup: store hashed passwords
- Threat model:
  - Database of password hashes is compromised (happens a LOT)
  - Attacker wants to figure out password for a given hash
- Improvement: **Salting**
  - Add a bit of random stuff (“salt”) to password before hashing
  - Random stuff differs per record
  - Store the salt with the hash so we can use it when verifying given passwords
- Why?
  - If I hash many possible passwords and find that “c00ldude” hashes to `a53d677656e7bcb216b9ef6e38bb7ab1`, then *anyone* with that hash must have that password
    - Can also see users with the same password, even if it’s unknown!
  - With a salt, I need to brute-force search *per user* instead of *once-for-everyone*
Storing passwords: **Iteration count**

- **Given setup:** Store salted hashed passwords
- **Threat model:**
  - Database of password hashes is compromised (happens a LOT)
  - Attacker wants to figure out password for a given hash
  - Attacker has lots of fast computers

- **Improvement:** **Iteration count**
  - Instead of just using $H(data)$, do $H(H(H(... H(data) ... )))$
  - Increase iteration count to make it very hard for attacker while still being feasible for login checks
  - Makes our hash function “slow” (configurably so!)

- **Why?**
  - If default hashing has speed of $X$, then an iteration count of 1000 gives a speed of $X/1000$. Login is a tiny amount of time in normal use, but it makes the attacker’s job 1000x harder for very little cost.
Password Vulnerabilities

- **Offline dictionary attack**: Crack a hashed password
  - Defense: Make harder by salting, iteration count
- **Online dictionary attack**: Try dictionary logins to actual live system
  - Defense: Max attempt counter, password complexity requirements
- **Password spraying**: Try few common passwords on many accounts/systems
  - Defense: Password complexity requirements
- **Credential stuffing**: Try the same user+password many places
  - Defense for individual: Password managers with strong crypto
  - Defense for organization: ????
- **Password guessing**: Do research then guess
  - Defense: User training, password complexity requirements
- **Exploiting user mistakes**: Post-It notes, sharing, unchanged defaults, etc.
  - Defense: Training, single-use expiring passwords for new accounts
- **Electronic monitoring**: Sniffing network, installing keylogger, etc.
  - Defense: Encryption, challenge-response schemes, training
UNIX password scheme

- Originally: hash stored in public-readable /etc/passwd file
  - Hashes were public; relied entirely on them being hard to crack
  - People slowly figured out in the 80s this was feasible (god what an awesome/lazy time to be an attacker...)

- Now: hash stored in separate root-readable /etc/shadow file

- Originally: small hash, few iterations
- Later: MD5 hash, more iterations
- Now: SHA 512 hash, configurable iterations

Passwords normally changed with **passwd** tool
Can generate shadow-compatible hash strings with **mkpasswd**
Password Cracking

• Dictionary attacks
  ▪ Develop a large dictionary of possible passwords and try each against the password file
  ▪ Each password must be hashed using each salt value and then compared to stored hash values

• Rainbow table attacks
  ▪ Pre-compute tables of hash values for all salts
  ▪ A mammoth table of hash values
  ▪ Can be countered by using a sufficiently large salt value and a sufficiently large hash length

• Password crackers exploit the fact that people choose easily guessable passwords
  ▪ Shorter password lengths are also easier to crack
Storing passwords correctly

• Storing password plaintext (or encrypted)
• Storing hashed password
• Storing salted hash of password
• Hash function has iteration count
• Just use PBKDF2, scrypt, bcrypt, etc.
• Have a user management library handle it
Where do stolen hashes go?

- Attacker uses directly, sells on black market, or they leak
- Often, eventually, they hit the public internet:
Importance of password storage illustrated (1)

- Plaintext passwords: 100% are “recovered” by attacker (obviously)
- Sorted hashes.org by “percent recovered” – all are unsalted!

Scroll to lower percent – almost all are salted.
Importance of password storage illustrated (2)

• Scroll to very low percentages...most use bcrypt or similar, which has an iteration count

<table>
<thead>
<tr>
<th>#</th>
<th>Username</th>
<th>Salt</th>
<th>Hash (1)</th>
<th>Hash (2)</th>
<th>Iterations</th>
<th>Date Found</th>
<th>Found</th>
<th>Left</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>971</td>
<td>Forum.lighthope.org</td>
<td></td>
<td>28721</td>
<td>28570</td>
<td>151</td>
<td>2018-08-17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>802</td>
<td>Scufgaming.com</td>
<td>WORDPRESS / WD</td>
<td>2'809</td>
<td>2'901</td>
<td>8</td>
<td>2018-05-30 00:34:17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>778</td>
<td>Pesfan.com</td>
<td>VIRALRATIN</td>
<td>426495</td>
<td>425704</td>
<td>791</td>
<td>2018-08-28 09:26:50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>558</td>
<td>Dailymotion</td>
<td>BCRYPT</td>
<td>16147134</td>
<td>16139263</td>
<td>7871</td>
<td>2018-05-29 02:20:48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>810</td>
<td>Siriusforum.net</td>
<td>BCRYPT</td>
<td>1284</td>
<td>1284</td>
<td>0</td>
<td>2018-05-29 16:17:02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>751</td>
<td>Legion.cm</td>
<td>BCRYPT</td>
<td>23113</td>
<td>23113</td>
<td>0</td>
<td>2018-05-29 02:21:30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>749</td>
<td>Krolop-gerst.com</td>
<td>BCRYPT</td>
<td>27748</td>
<td>27748</td>
<td>0</td>
<td>2018-05-29 02:21:30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>972</td>
<td>Totaljerkface.com</td>
<td>BCRYPT</td>
<td>188055</td>
<td>188055</td>
<td>0</td>
<td>2018-08-16 23:58:37</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Conclusion: How you store password has HUGE effect on what happens if (when) they are breached!
Password Selection Strategies

• **User education**
  - Users can be told the importance of using hard to guess passwords and can be provided with guidelines for selecting strong passwords

• **Computer generated passwords**
  - Users have trouble remembering them (good for single-use, bad for long-term)

• **Reactive password checking**
  - System periodically runs its own password cracker to find guessable passwords

• **Complex password policy**
  - User is allowed to select their own password, however the system checks to see if the password is allowable, and if not, rejects it
  - Goal is to eliminate guessable passwords while allowing the user to select a password that is memorable
Four means of authentication

- **Something you know**
  - Password, PIN, answers to prearranged questions

- **Something you have (token)**
  - Smartcard, electronic keycard, physical key

- **Something you are (static biometrics)**
  - Fingerprint, retina, face

- **Something you do (dynamic biometrics)**
  - Voice pattern, handwriting
Types of tokens (1)

- **Cards** (or card-like things)
  - Magnetic stripe (read-only, **clear** communication)
  - Memory card (read-only/read-write, no processor, **clear** communication)
  - **Smart card** (read-only/read-write, has processor, **encrypted** communication)
    - May be **contact** (e.g., this bank card) or **contactless** (e.g., your DukeCard)

- **Cryptographic token** (AKA one-time passwords)
  - Holds crypto key that can’t (easily) be extracted
  - Uses it to generate a time-sync’d key stream
Types of tokens (2)

• **Communication device** (i.e., your phone)
  - Relies on real-time and secure communication
  - Good: *Dedicated app with cryptographic secrets* (e.g. Duo)
  - Bad: Using *SMS (text messaging)*
    - Many examples of SMS hijacking: Every helpdesk employee at your mobile provider can do it (either because they were fooled or they’re evil)!
    - Better than nothing, though...

• **Fallback passwords**
  - Long, random single use passwords that are written down or stored
  - Kept in a secure location for exception situations (e.g. in response to an account hijack)
Types of tokens (3)

• **Physical keys** (they’re made of metal and you have some)
  - Many different types, same idea: mechanically unbind a lock
  
  - Turns out you can attack physical locks many different ways (covered later when we get to physical security)
More on contactless communication

• Recall: smart cards may be **contactless**
  - Has CPU, memory, ROM, maybe even non-volatile storage (EEPROM/flash)

• Terminology and standards:
  - **RFID**: Radio Frequency Identification
    - Broad category
    - Usually *powered* wirelessly (inductively or via RF pulse)
    - May be very short range (DukeCard) or longer (Duke parking pass)
    - May be very dumb (“just transmit this string”) or more advanced (“execute this encrypted read/write command”)
  - **NFC**: Near Field Communication
    - A collection of standards for two-way communication based on RFID
    - Generally on the smarter side in terms of protocol
    - Supported by modern mobile phones
      - Powers things like “ApplePay”, “GooglePay”, etc.
      - Your phone can be your DukeCard using NFC
## Four means of authentication

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Something you <strong>know</strong></td>
<td>Password, PIN, answers to prearranged questions</td>
</tr>
<tr>
<td>Something you <strong>have</strong></td>
<td>Smartcard, electronic keycard, physical key</td>
</tr>
<tr>
<td><strong>(token)</strong></td>
<td></td>
</tr>
<tr>
<td>Something you <strong>are</strong></td>
<td>Fingerprint, retina, face</td>
</tr>
<tr>
<td><strong>(static biometrics)</strong></td>
<td></td>
</tr>
<tr>
<td>Something you <strong>do</strong></td>
<td>Voice pattern, handwriting</td>
</tr>
<tr>
<td><strong>(dynamic biometrics)</strong></td>
<td></td>
</tr>
</tbody>
</table>
Biometric basics

- Authenticate based on unique physical characteristics (pattern recognition)
- More complex/expensive than previous techniques
- Common characteristics:
  - Fingerprint
  - Face
- Less common:
  - Hand geometry
  - Retinal pattern
  - Iris
  - Signature
  - Voice

Figure 3.8 Cost Versus Accuracy of Various Biometric Characteristics in User Authentication Schemes.
Processes of biometric authentication

- **Enrollment**: Add new people
- **Verification**: User asserts identity and proves it
- **Identification**: Pick out which user the given biometric corresponds to (harder)
Analyzing biometric accuracy

- Biometric is pattern matching; naturally imprecise (probabilistic)
  - Will get a **match score**, system *accepts* when score ≥ **threshold**

- Metrics to evaluate a biometric system:
  - **False Accept Rate (FAR)**: Probability it allows the wrong person
    = False positive (FP) rate
  - **False Reject Rate (FRR)**: Probability it disallows the right person
    = False negative (FN) rate
  - **Receiver Operating Characteristic (ROC)**: Comparison of the FAR+FRR with respect to threshold (a general concept for *any* classifier)

Figures from [here](#)
Remote authentication
What about the network?

- Authentication over a network is more complex – more to worry about
  - **Eavesdropping:**
    - Capturing a credential (allowing attacker to login)
    - Capturing a session cookie (evidence of authentication, allows attacker to act as user)
  - **Replay attacks:** Even if attacker doesn’t know credential, can they blindly replay the packets to login?
    - Example: a “pass the hash” attack

- Solution: Various **challenge-response** schemes
A basic challenge-response scheme

- Assume we have some authentication secret $S$
  - Token value, biometric signature, etc...
- Don’t want to send it (*or even its hash!*)
- Instead, server issues a *challenge* (random value $R$) to client that can only be answered if it has $S$, but which doesn’t reveal $S$.

Client

I’m user Bob

Oh yeah? Assume $R=5248$, so compute $h(R + h(S))$ for me, where $S$ is Bob’s secret.

Server

Here’s $h(R + h(S))$

oh ok cool
Challenge-Response: What about passwords?

- In the scheme shown, if the password hash is leaked, it’s *equivalent* to having the actual password, since we only need $h(S)$!
- Other challenge-response schemes avoid this issue, e.g. **Salted Challenge Response Authentication Mechanism (SCRAM)**

```
SaltedPassword = \{salted hash of password\}
ClientKey = HMAC(SaltedPassword, "Client Key")
StoredKey = H(ClientKey)
ServerKey = HMAC(SaltedPassword, "Server Key")
Auth = \{username, salt, iteration, CombinedNonce\}
ClientProof = ClientKey \^\ HMAC(StoredKey, Auth)
ServerProof = HMAC(ServerKey, Auth)
```

For more, see [Wikipedia](https://en.wikipedia.org/wiki/SCRAM) or this article
Multifactor Authentication (MFA)
Multifactor authentication (MFA)

• Now that we’ve covered the modes of authentication (something you know/have/are/do), definition is easy:
  ▪ **Multifactor Authentication**: Require more than one of those categories.
    (that’s all)

• In *practice*, today it usually means password + token.
  ▪ Lame: Password + SMS
  ▪ Better: Password + actual app

• Looking forward:
  ▪ Trusted Platform Modules (TPMs) are hardware chips that can securely hold cryptographic secrets without leaking them (unless there’s a flaw…)
  ▪ Modern standard: **WebAuthn** – use TPM to make MFA easy
WebAuthn: Practical MFA of the future

- WebAuthn incorporates FIDO authentication (an open standard)
  - Web app: Implements WebAuthn standard to ask for a login
  - Browser: Needs WebAuthn support, hooks into support from OS
  - OS: Provides a Client-To-Authenticator Protocol (CTAP). May use:
    - Internal authenticator (using TPM chip), or
    - External token (phone, watch, USB security token)

* FIDO2 Project

Figure from https://fidoalliance.org/fido2/
Access control

So you’ve proven who you are, but what are you allowed to do?
Topics

- Core concepts
- Access control policies:
  - DAC
  - UNIX file system
  - MAC
  - RBAC
- Identity federation
Subjects, Objects, Actions, and Rights

Subject (initiator)
- The thing making the request (e.g., the user)

Verb (request)
- The operation to perform (e.g., read, delete, etc.)

Right (permission)
- A specific ability for the subject to do the action to the object.

Object (target)
- The thing that’s being hit by the request (e.g., a file).
Categories of Access Control Policies

- **Discretionary AC (DAC):** There’s a list of permissions attached to the subject or object (or possibly a giant heap of global rules).

- **Mandatory AC (MAC):** Objects have classifications, subjects have clearances, subjects cannot give additional permissions.
  - An overused/abused term

- **Role-Based AC (RBAC):** Subjects belong to roles, and roles have all the permissions.
  - The current Enterprise IT buzzword meaning “good” security

- **Attribute-Based AC (ABAC):** Subjects and objects have attributes, rules engine applies predicates to these to determine access
  - Allows fine-grained expression
  - Usually complex, seldom implemented
  - *We’re gonna skip this, since I’ve never seen anyone care about it IRL*
Topics

• Core concepts

• Access control policies:
  ▪ DAC
    • UNIX file system
  ▪ MAC
  ▪ RBAC

• Identity federation
Discretionary Access Control (DAC)

- **Discretionary Access Control (DAC):** Scheme in which an entity may enable another entity to access some resource
  - Often provided using an access matrix: subjects × objects
  - Each entry shows the access rights of that subject to that object

**Pseudocode**

```pseudo
def IsActionAllowed(subject, object, action)
    if action in get_permissions(subject, object)
        return true
    return false
```


Implementation

- Can use various data structures, none of which should surprise you

Flat list

<table>
<thead>
<tr>
<th>Subject</th>
<th>Access Mode</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Own</td>
<td>File 1</td>
</tr>
<tr>
<td>A</td>
<td>Read</td>
<td>File 1</td>
</tr>
<tr>
<td>A</td>
<td>Write</td>
<td>File 1</td>
</tr>
<tr>
<td>A</td>
<td>Own</td>
<td>File 3</td>
</tr>
<tr>
<td>A</td>
<td>Read</td>
<td>File 3</td>
</tr>
<tr>
<td>A</td>
<td>Write</td>
<td>File 3</td>
</tr>
<tr>
<td>B</td>
<td>Read</td>
<td>File 1</td>
</tr>
<tr>
<td>B</td>
<td>Own</td>
<td>File 2</td>
</tr>
<tr>
<td>B</td>
<td>Read</td>
<td>File 2</td>
</tr>
<tr>
<td>B</td>
<td>Write</td>
<td>File 2</td>
</tr>
<tr>
<td>B</td>
<td>Write</td>
<td>File 3</td>
</tr>
<tr>
<td>B</td>
<td>Read</td>
<td>File 4</td>
</tr>
<tr>
<td>C</td>
<td>Read</td>
<td>File 1</td>
</tr>
<tr>
<td>C</td>
<td>Write</td>
<td>File 1</td>
</tr>
<tr>
<td>C</td>
<td>Read</td>
<td>File 2</td>
</tr>
<tr>
<td>C</td>
<td>Own</td>
<td>File 4</td>
</tr>
<tr>
<td>C</td>
<td>Read</td>
<td>File 4</td>
</tr>
<tr>
<td>C</td>
<td>Write</td>
<td>File 4</td>
</tr>
</tbody>
</table>

Matrix

<table>
<thead>
<tr>
<th>SUBJECTS</th>
<th>OBJECTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Own</td>
</tr>
<tr>
<td></td>
<td>Read</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>User B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td>Own</td>
</tr>
<tr>
<td>User C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
</tbody>
</table>

Linked list

Figure 4.2 Example of Access Control Structures
UNIX Philosophy

• “UNIX” here includes Linux, MacOSX, and traditional UNIX

• Major tenet of UNIX philosophy: **everything is a file**
  ▪ Why?
  ▪ **Flexibility**: If you build an API to access files, you can use it for everything 😊
  ▪ **Security**: If you build a permission system for files, you can use it for everything 😊

• How everything is a file:
  ▪ Hardware devices show up as files under `/dev`
  ▪ Info and controls for the running kernel are simulated in `/proc` and `/sys`
  ▪ You can attach (“**mount**”) storage devices to directories all under one global hierarchy
  ▪ You can even turn a pipe or socket into a named file!
UNIX File Access Control

• Your **disk** is a dumb flat array of blocks that can be read/written
• A **filesystem** organizes this, handles allocation of disk regions to **files**, lets you organize these files into hierarchical **directories**.
• (Most) UNIX filesystems store file metadata in **inodes** (index nodes)
  ▪ Inodes store metadata about a file/directory, including ownership/permissions
  ▪ They live on disk in an inode table; in memory in a kernel inode cache
• **Directories** are special files that list names + inode numbers
• There are a few other special file types:
  ▪ **Symbolic links** (also known as **symlinks** or **soft links**)
  ▪ **Device files** (**character** or **block**)
  ▪ **Named pipes** (also known as **fifos**)
  ▪ **Named sockets** (like two-way fifos)
**UNIX File Access Control Basics**

- **Users** have numbers called User ID numbers ("uid")
- Users can belong to one or more **groups**; groups have numbers called Group ID numbers ("gid")
- A file is **owned** by a user (uid) and a group (gid)
  - The reference is numeric; `ls` translates numbers to names for you; can turn off with `-n`
- Twelve permission bits applied to file (file “mode”)
  - Lower 9 bits: `user/group/others : read/write/execute`
  - Upper 3 bits: “Weird” ones (covered next)

(a) Traditional UNIX approach (minimal access control list)
UNIX File Access Control Basics

- Change a file’s owner with `chown` (changes uid)
- Change a file’s group with `chgrp` (changes gid)
- Change a file’s mode (permissions) with `chmod` (changes mode bits)
  - Can express in base-8 octal: `chmod 750` yields `rwxr-x---`
  - Can express symbolically: `chmod u+rw` turns on owner’s read/write
- The other three bits:
  - **SetUID** (u+s) and **SetGID** (g+s):
    - Executables run that have this bit run as the user/group that owns it
    - A way to allow privilege escalation, either
      - legitimately, like for `sudo`, or
      - illegitimately, as in a backdoor created by attackers
  - **Sticky bit** (+t):
    - Applied to directories; when set, only the owner of any file in the directory can rename, move, or delete that file – used for e.g. `/tmp`
- The **root** user (uid 0) is immune from permission bit limitations.
  - Hence using `sudo` to carry out chown/chgrp/chmod commands when you otherwise couldn’t.
Sidebar: Hard vs soft links

- Directories are special files that list file names and inode numbers
- **Hard link**: When multiple directory entries refer to the same inode
  - Such “files” are actually the same content; change one = change all
  - Useful for creating cheap “clones” of files, no extra storage
- **Soft link**: A special file that refers to another path
  - Also called **symbolic link** or **symlink**.
  - Path can be relative or absolute
  - Can traverse file systems or even point to nonexistent things
  - Can be used as file system organization “duct tape”
    - Example: Symlink a long, complex path to a simpler place, e.g.:
      ```
      $ ln -s /remote/codebase/projectX/beta/current/build ~/mybuild
      $ cd ~/mybuild
      ```

Figure from “Computer Forensics: Unix File Systems” by Thomas Schwarz (Santa Clara Univ)
File system access control lists (ACLs)

- Issue: UNIX model can’t represent all permission situations (e.g. multiple groups or users having access); use **Access Control Lists (ACLs)**
  - Arbitrary list of rules governing access per-file/directory
  - More flexible than classic UNIX permissions, but more metadata to store/check

**Windows ACL UI**

**Examples of Linux ACL commands**

```
Set all permissions for user johny to file named "abc".
# setfacl -m "u:johny:rw-" abc
```

```
Check permissions
# getfacl abc
```

```
Change permissions for user johny.
# setfacl -m "u:johny:r-x" abc
```

```
Check permissions
# getfacl abc
```

```
masked entries
{ user: :rw- 
  user:joe:rw- 
  group::r-- 
  mask::rw- 
  other::--- }
```

![Figure 4.5 UNIX File Access Control](From Arch Wiki)
Topics

• Core concepts

• Access control policies:
  ▪ DAC
    • UNIX file system
  ▪ MAC
  ▪ RBAC

• Identity federation
MAC example: SELinux

- Developed by U.S. Dept of Defense
- General deployment starting 2003
- Can apply rules to virtually every user/process/hardware pair
- Rules are governed by system administrator only
  - No such thing as “selinux_chmod” for users

```pseudocode
bool IsActionAllowed(subject, object, action) {
    for each rule in rules:
        if rule allows (subject, object, action) return true
    return false
}
```
MAC example: SELinux
Topics

• Core concepts

• Access control policies:
  ▪ DAC
    • UNIX file system
  ▪ MAC
  ▪ RBAC

• Identity federation
RBAC: The thing you invent if you spend enough time doing access control

- Scenario:
  - Frank: “Bob just got hired, please give him access.”
  - Admin: “What permissions does he need?”
  - Frank: “Same as me.”

- Later, a new system is added
  - Bob: “Why can’t I access the new system?!"
  - Admin: “Oh, I didn’t know you needed it too…”
  - Bob: “I need everything Frank has!”

- Later, Frank is promoted to CTO
  - Admin: “Welp, looks like Bob also needs access to our private earnings, since this post-it says he gets everything Frank has…”

- The admin is later fired amidst allegations of conspiracy to commit insider trading with Bob. He dies in prison. 😞
RBAC

- Decide what KINDS of users you have (roles)
- Assign permission to roles.
- Assign users to roles.

- When a role changes, everyone gets the change.
- When a user’s role changes, that user gets a whole new set of permissions.
- No more special unique snowflakes.

- Roles may be partially ordered, e.g. “Production developer” inherits from “Developer” and adds access to the production servers

Figure 4.6 Users, Roles, and Resources
RBAC implementation

- Unsurprisingly, you can represent this using various data structures.
  - Anything that can represent two matrices:

```
R1  R2  Rn
  control  owner  owner  control
  read ♦  read ♦  wakeup  wakeup
  seek ♦  seek ♦  owner  owner
```

```
Pseudocode
bool IsActionAllowed(subject, object, action) {
    if (action ∈ get_permissions(subject.role, object))
        return true
}
```
Topics

• Core concepts

• Access control policies:
  ▪ DAC
  ▪ UNIX file system
  ▪ MAC
  ▪ RBAC

• Identity federation
Identity Federation made simple

- **Identity Federation**: System to allow an organization to trust identities/credentials managed by another organization
  - Allows you to provide access to users from external orgs (and vice versa)
- Translation:
  - Allow one entity to manage the concept of “logging in” (credentials, etc.), and communicate that to another entity on behalf of the user
- Want a standard to support federation from any provider? **OAuth**
- Duke has an authentication system: **Duke NetID**
  - You can write apps that use OAuth to allow login via Duke NetID!
- Corporate providers: Google/Facebook
- Open provider framework: OpenID
Any questions?