# CSci 4271W
# Development of Secure Software Systems
# Day 24: Design Principles and Authentication

Stephen McCamant
University of Minnesota, Computer Science & Engineering

## Outline

Saltzer & Schroeder's principles

More secure design principles

User authentication

Error rate trade-offs

Web authentication

## A classic paper

Jerome H. Saltzer and Michael D. Schroeder, "The Protection of Information in Computer Systems." In *Proceedings of the IEEE*, Sept. 1975. (853 citations per IEEE)

## Economy of mechanism

- Security mechanisms should be as simple as possible
- Good for all software, but security software needs special scrutiny

## Fail-safe defaults

- When in doubt, don't give permission
- Whitelist, don't blacklist
- Obvious reason: if you must fail, fail safe
- More subtle reason: incentives

## Complete mediation

- Every mode of access must be checked
  - Not just regular accesses: startup, maintenance, etc.
- Checks cannot be bypassed
  - E.g., web app must validate on server, not just client

## Open design

- Security must not depend on the design being secret
- If anything is secret, a minimal key
  - Design is hard to keep secret anyway
  - Key must be easily changeable if revealed
  - Design cannot be easily changed

## Open design: strong version

- "The design should not be secret"
- If the design is fixed, keeping it secret can't help attackers
- But an unscrutinized design is less likely to be secure

## Separation of privilege

- Real world: two-person principle
- Direct implementation: separation of duty
- Multiple mechanisms can help if they are both required
  - Password and `wheel` group in Unix

## Least privilege

- Programs and users should have the most limited set of powers needed to do their job
- Presupposes that privileges are suitably divisible
  - Contrast: Unix `root`

## Least privilege: privilege separation

- Programs must also be divisible to avoid excess privilege
- Classic example: multi-process OpenSSH server
- N.B.: Separation of privilege $\neq$ privilege separation

## Least common mechanism

- Minimize the code that all users must depend on for security
- Related term: minimize the Trusted Computing Base (TCB)
- E.g.: prefer library to system call; microkernel OS

## Psychological acceptability

- A system must be easy to use, if users are to apply it correctly
- Make the system's model similar to the user's mental model to minimize mistakes

## Sometimes: work factor

- Cost of circumvention should match attacker and resource protected
- E.g., length of password
- But, many attacks are easy when you know the bug

## Sometimes: compromise recording

- Recording a security failure can be almost as good as preventing it
- But, few things in software can't be erased by `root`

## Outline

## Separate the control plane

- Keep metadata and code separate from untrusted data
- Bad: format string vulnerability
- Bad: old telephone systems

## Defense in depth

- Multiple levels of protection can be better than one
- Especially if none is perfect
- But, many weak security mechanisms don't add up

## Canonicalize names

- Use unique representations of objects
- E.g. in paths, remove ., . . ., extra slashes, symlinks
- E.g., use IP address instead of DNS name

## Fail-safe / fail-stop

- If something goes wrong, behave in a way that's safe
- Often better to stop execution than continue in corrupted state
- E.g., better segfault than code injection

## Outline

Saltzer & Schroeder's principles

More secure design principles

User authentication

Error rate trade-offs

Web authentication

## Authentication factors

- Something you know (password, PIN)
- Something you have (e.g., smart card)
- Something you are (biometrics)
- CAPTCHAs, time and location, . . .
- Multi-factor authentication

## Passwords: love to hate

- Many problems for users, sysadmins, researchers
- But familiar and near-zero cost of entry
- User-chosen passwords proliferate for low-stakes web site authentication

## Password entropy

- Model password choice as probabilistic process
- If uniform, $\log_2 |S|$
- Controls difficulty of guessing attacks
- Hard to estimate for user-chosen passwords
  - Length is an imperfect proxy

## Password hashing

- Idea: don't store password or equivalent information
- Password 'encryption' is a long-standing misnomer
  - E.g., Unix `crypt(3)`
- Presumably hard-to-invert function $h$
- Store only $h(p)$

## Dictionary attacks

- Online: send guesses to server
- Offline: attacker can check guesses internally
- Specialized password lists more effective than literal dictionaries
  - Also generation algorithms (s $\rightarrow$ \$, etc.)
- $\sim$25% of passwords consistently vulnerable

## Better password hashing

- Generate random salt $s$, store $(s, h(s, p))$
  - Block pre-computed tables and equality inferences
  - Salt must also have enough entropy
- Deliberately expensive hash function
  - AKA password-based key derivation function (PBKDF)
  - Requirement for time and/or space

## Password usability

- User compliance can be a major challenge
  - Often caused by unrealistic demands
- Distributed random passwords usually unrealistic
- Password aging: not too frequently
- Never have a fixed default password in a product

## Backup authentication

- Desire: unassisted recovery from forgotten password
- Fall back to other presumed-authentic channel
  - Email, cell phone
- Harder to forget (but less secret) shared information
  - Mother's maiden name, first pet's name
- Brittle: ask Sarah Palin or Mat Honan

## Centralized authentication

- Enterprise-wide (e.g., UMN ID)
- Anderson: Microsoft Passport
- Today: Facebook Connect, Google ID
- May or may not be single-sign-on (SSO)

## Biometric authentication

- Authenticate by a physical body attribute

\+ Hard to lose

– Hard to reset

– Inherently statistical

– Variation among people

## Example biometrics

- (Handwritten) signatures
- Fingerprints, hand geometry
- Face and voice recognition
- Iris codes

## Outline

- Saltzer & Schroeder's principles
- More secure design principles
- User authentication
- Error rate trade-offs
- Web authentication

## Imperfect detection

- Many security mechanisms involve imperfect detection/classification of relevant events
- Biometric authentication
- Network intrusion detection
- Anti-virus (malware detection)
- Anything based on machine learning

## Detection results

- True positive: detector says yes, reality is yes
- True negative: detector says no, reality is no
- False positive: detector says yes, reality is no
- False negative: detector says no, reality is yes
- Note: terminology may flip based on detecting good or bad

## Why a trade-off?

- Imperfect methods have a trade-off between avoiding FPs and avoiding FNs
- Sometimes a continuous trade-off (curve), e.g. based on a threshold
  - E.g., spam detector "score"
- May need to choose both a basic mechanism and a threshold

## Two ratios to capture the trade-off

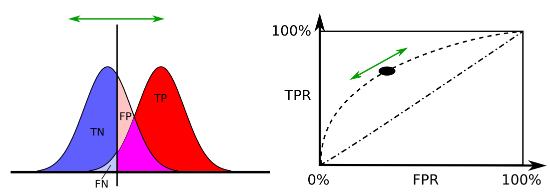
- True positive rate:

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = 1 - \mathrm{FNR}$$

- False positive rate:

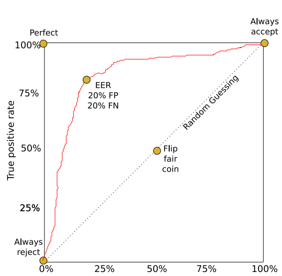
$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{N}} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} = 1 - \mathrm{TNR}$$

## ROC curve intro




Source: https://commons.wikimedia.org/wiki/File:ROC_curves.svg CC-BY-SA 3.0 "Sharpr"

## Error rates: ROC curve




## Extreme biometrics examples

- `exact_iris_code_match`: very low false positive (false authentication)
- `similar_voice_pitch`: very low false negative (false reject)

## Where are these in ROC space?

```
A if (iris()) return REJECT; else return ACCEPT;
B return REJECT;
C if (iris()) return ACCEPT; else return REJECT;
D if (iris() && pitch()) return ACCEPT; else return REJECT;
E return ACCEPT;
F if (rand() & 1) return ACCEPT; else return REJECT;
G if (pitch()) return ACCEPT; else return REJECT;
H if (iris() || pitch()) return ACCEPT; else return REJECT;
```

## Outline

Saltzer & Schroeder's principles

More secure design principles

User authentication

Error rate trade-offs

Web authentication

## Per-website authentication

- Many web sites implement their own login systems
  - \+ If users pick unique passwords, little systemic risk
  - – Inconvenient, many will reuse passwords
  - – Lots of functionality each site must implement correctly
  - – Without enough framework support, many possible pitfalls

## Building a session

- HTTP was originally stateless, but many sites want stateful login sessions
- Built by tying requests together with a shared session ID
- Must protect confidentiality and integrity

## Session ID: what

- Must not be predictable
  - Not a sequential counter
- Should ensure freshness
  - E.g., limited validity window
- If encoding data in ID, must be unforgeable
  - E.g., data with properly used MAC
  - Negative example: crypt(username || server secret)

## Session ID: where

- Session IDs in URLs are prone to leaking
  - Including via user cut-and-paste
- Usual choice: non-persistent cookie
  - Against network attacker, must send only under HTTPS
- Because of CSRF, should also have a non-cookie unique ID

## Session management

- Create new session ID on each login
- Invalidate session on logout
- Invalidate after timeout
  - Usability / security tradeoff
  - Needed to protect users who fail to log out from public browsers

## Account management

- Limitations on account creation
  - CAPTCHA? Outside email address?
- See previous discussion on hashed password storage
- Automated password recovery
  - Usually a weak spot
  - But, practically required for large system

## Client and server checks

- For usability, interface should show what's possible
- But must not rely on client to perform checks
- Attackers can read/modify anything on the client side
- Easy example: item price in hidden field

## Direct object references

- Seems convenient: query parameter names resource directly
  - E.g., database key, filename (path traversal)
- Easy to forget to validate on each use
- Alternative: indirect reference like per-session table
  - Not fundamentally more secure, but harder to forget check

## Function-level access control

- E.g. pages accessed by URLs or interface buttons
- Must check each time that user is authorized
  - Attack: find URL when authorized, reuse when logged off
- Helped by consistent structure in code