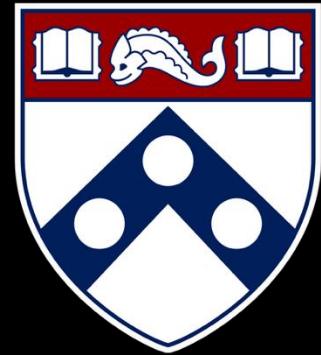
# Secure Systems Engineering and Management



## A Data-driven Approach

Michael Hicks

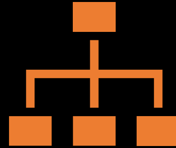
# Secure Design: Principles and Controls (part 1)

Protecting Data and Proving Identity

UPenn CIS 7000-003
Spring 2026

# From Threats to Mitigations

We **enumerate threats** against our security objectives using STRIDE.



For each threat, we **propose a response**: a control and implementation guidance.

These two lectures build the **technical vocabulary** and the **design thinking** for that.

# Recall: Security Objectives

| Requirement | Definition |
| --- | --- |
| **Confidentiality** | Protect information from unauthorized disclosure |
| **Integrity** | Protect information from unauthorized modification |
| **Availability** | Provide access when required |
| **Accountability** | Non-repudiation via permanent audit records |

| Feature | Intent |
| --- | --- |
| **Authentication** | Reliably identifying users |
| **Authorization** | Granting access based on roles and rights |

Requirements expressed in terms offered by these features

# Principles vs. Controls

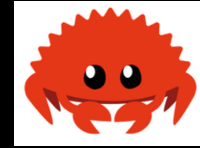
A **principle** is a high-level design goal with many possible manifestations

A **control** is a specific practice consonant with sound design principles

**Design phase** → principles for avoiding *flaws* (design-level problems)

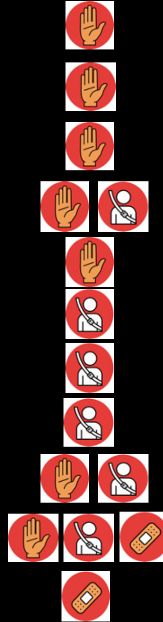
**Implementation phase** → controls for avoiding *bugs* (code problems)

# Three Categories of Security Principles

| | | | |
|---|---|---|---|
| **Prevention** | **Eliminate** defects entirely |  | |
| **Harm reduction** | **Limit damage** from unknown defects | | |
| **Detection & recovery** | **Identify** attacks and **undo** damage | | |

# Secure Design Principles

- Favor simplicity
- Use fail-safe defaults
- Do not expect expert users
- Trust with reluctance
  - Small trusted computing base
  - Grant least privilege
  - Compartmentalize
  - Promote privacy
- Defend in depth
- Use community resources
- Monitor and trace



DRAFT

date October 3, 1974

THE PROTECTION OF INFORMATION IN COMPUTER SYSTEMS

by

Jerome H. Saltzer and Michael D. Schroeder

- Economy of mechanism
- Fail-safe defaults
- Psychological acceptability
- (several S&S principles)
  - Least common mechanism
  - Least privilege
  - Separation of privilege
  - (modern addition)
- (modern addition)
- Open design
- Compromise recording

© 1974 by J. H. Saltzer

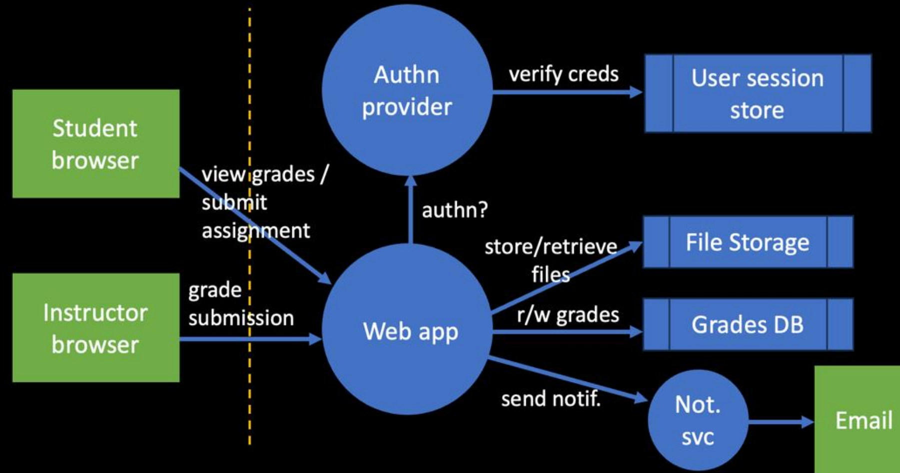
# Outline for Today

- **Principles** and **Controls**
- **Confidentiality** (by encryption)
- **Authentication** – proving identity
- Four principles:
  - **Do not expect expert users**
  - **Fail-safe defaults**
  - **Favor simplicity**
  - **Defend in depth** (throughout)

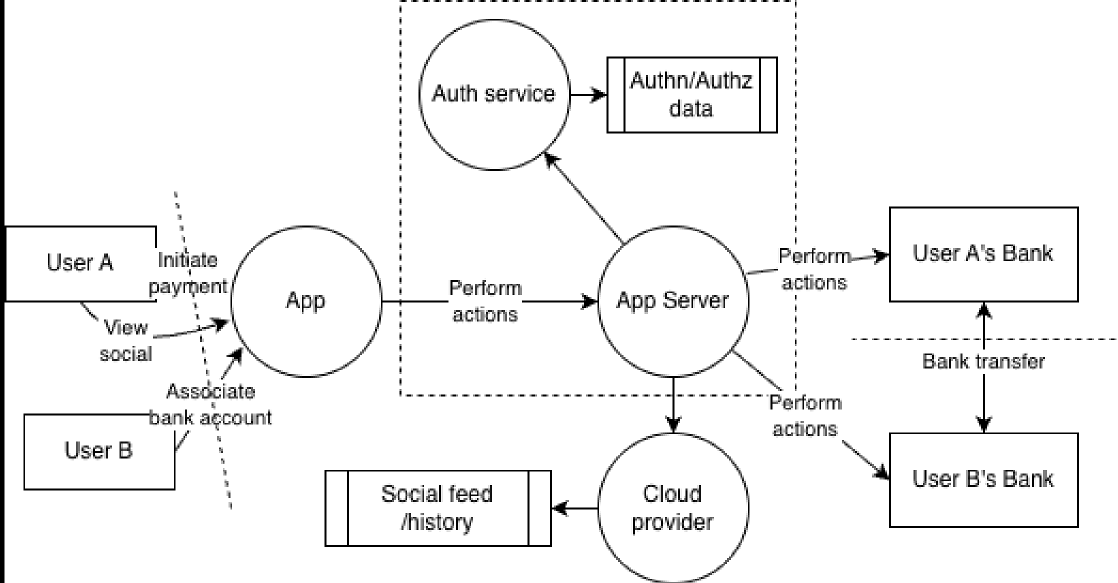
Next time

- **Authorization**
- **Integrity** and **Accountability**
- Remaining principles:
  - **Trust with Reluctance** (all)
  - **Monitoring and Traceability**
  - **Putting it all together**
  - **Use community resources**

# Payment app: DFD



# CMS: DFD



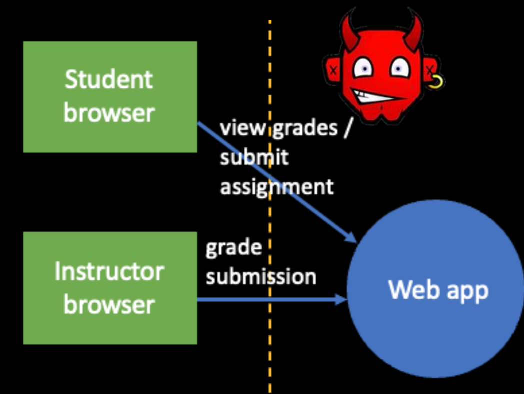
# Confidentiality by Encryption

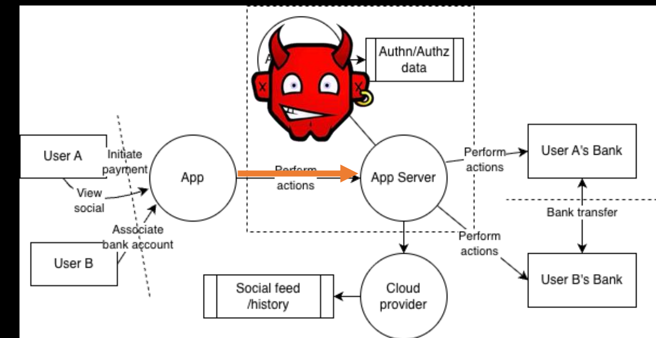
# The Threat: Eavesdropping and Data Theft

**STRIDE category:** Information Disclosure

An attacker eavesdrops on or modifies data as it moves between components – or accesses stored data directly.

- **CMS:** Grades travel from app server to browser over campus Wi-Fi
- **Payment app:** Bank credentials and transfer instructions travel between app and server



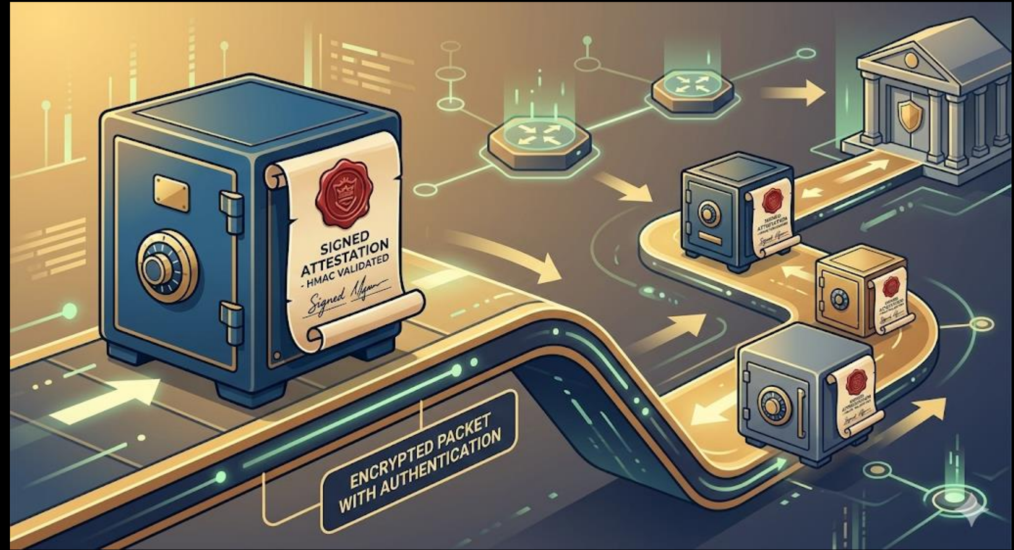
# Encryption 101

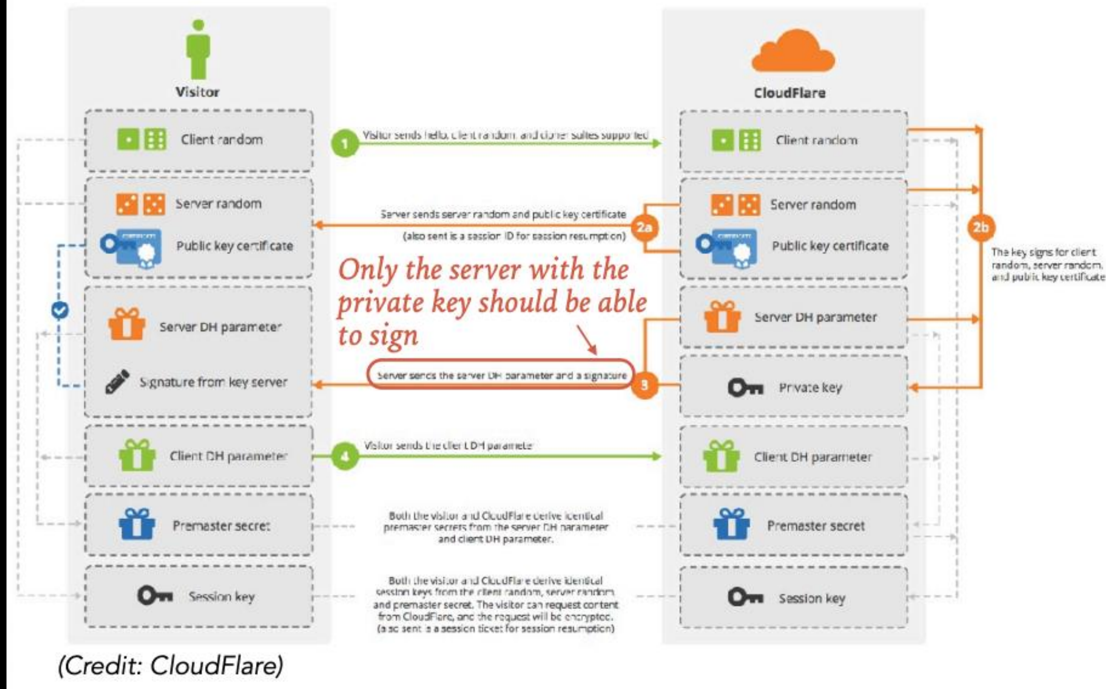
- Symmetric key encryption
  - Cipher
  - Message authentication code (MACs)
- Public key encryption – helps with proof of identity
  - Cipher
  - Signature
- Challenge: Key management

# Encryption in Transit: Transport Layer Security

**What TLS provides**

- **Confidentiality** — encryption of all traffic
- **Integrity** — MACs detect modification
- **Server authentication** — certificates prove identity






**The handshake:** client hello → server certificate → key exchange → symmetric session key

# Use Standards, and Standard Implementations

TLS is a formal standard

- Went through rigorous review for many years. Now ubiquitous.

**Saltzer & Schroeder's open design principle:**

- Security should depend on the secrecy of *keys*, not of *algorithms* or *code*

**Use vetted libraries** (community resource)

- libsodium, OS-provided crypto APIs, cloud KMS

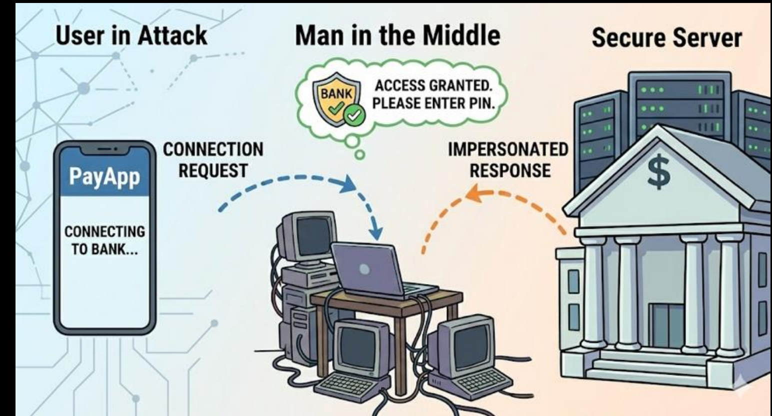
```
Internet Engineering Task Force (IETF)                       E. Rescorla
Request for Comments: 8446                                       Mozilla
Obsoletes: 5077, 5246, 6961                                  August 2018
Updates: 5705, 6066
Category: Standards Track
ISSN: 2070-1721

        The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

   This document specifies version 1.3 of the Transport Layer Security
   (TLS) protocol.  TLS allows client/server applications to communicate
   over the Internet in a way that is designed to prevent eavesdropping,
   tampering, and message forgery.

   This document updates RFCs 5705 and 6066, and obsoletes RFCs 5077,
   5246, and 6961.  This document also specifies new requirements for
   TLS 1.2 implementations.
```

# TLS in Practice

**Certificate pinning:** The payment app can *pin* the server's certificate (or its CA) so a compromised CA or rogue cert can't intercept traffic.

**Common mistakes:**

- Allowing fallback to older TLS versions
- Not validating certificates
- Shipping with development-mode certificate bypass left in

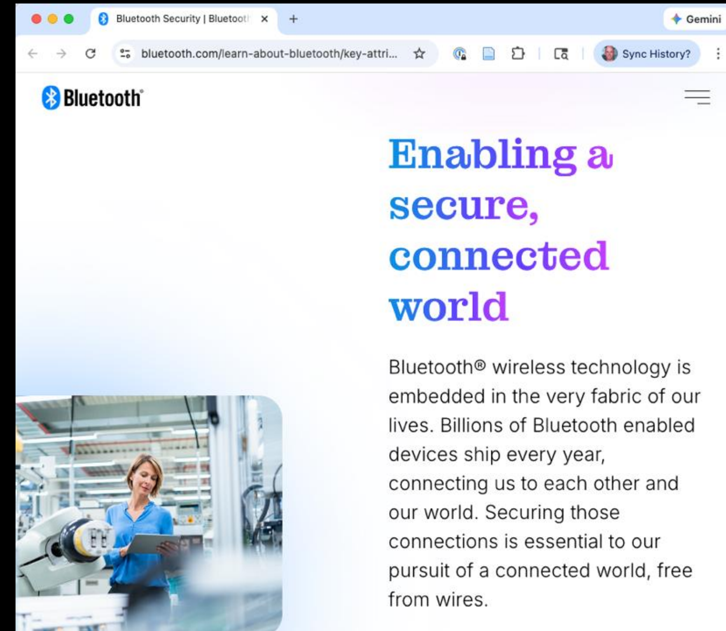


# Not All Comms are HTTP: BLE Security

**Bluetooth Low Energy (BLE)** is used by IoT devices, wearables, and medical devices.

**BLE Secure Connections** (Bluetooth 4.2+): ECDH key exchange + AES-CCM encryption

**Pairing modes for levels of protection**



# Application-Layer Encryption: **Defense in Depth**

Encrypt the payload *inside* the transport connection.

Even if BLE (or TLS) security is bypassed, the data is still protected.

**General principle:** For every communication channel in your DFD:

1. Determine whether the transport provides adequate protection
2. Add application-layer encryption where it doesn't

# Encryption at Rest

**The threat:** An attacker gains access to stored data – in a database, on a device, in a backup.

**What if the database is behind a firewall?**

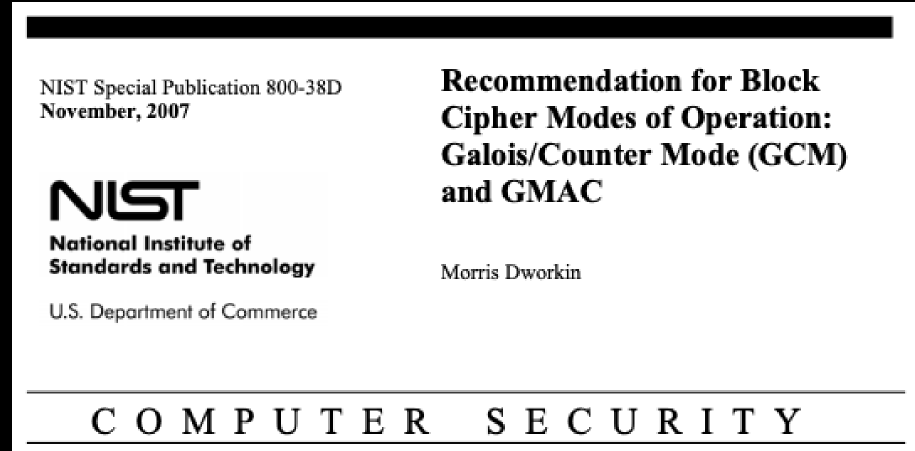
*Defense in depth* assumes outer layers *can* be breached:

- Compromised application server
- Insider with network access
- Stolen backup tape

Encryption at rest is the *next* layer.

# Symmetric Encryption: AES-GCM

**AES** is the standard.
Key sizes: 128 or 256 bits.


NIST Special Publication 800-38D
November, 2007

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

**Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC**

Morris Dworkin

COMPUTER SECURITY

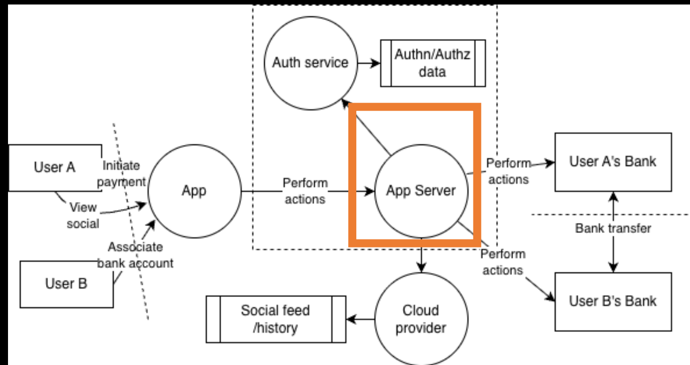
**AES-GCM** (Galois/Counter Mode) is the modern default:

- Provides both **confidentiality** and **integrity** (authenticated encryption)
- Encrypting without integrity (e.g., AES-ECB, AES-CBC without MAC) lets attackers flip bits and thereby corrupt plaintext without detection

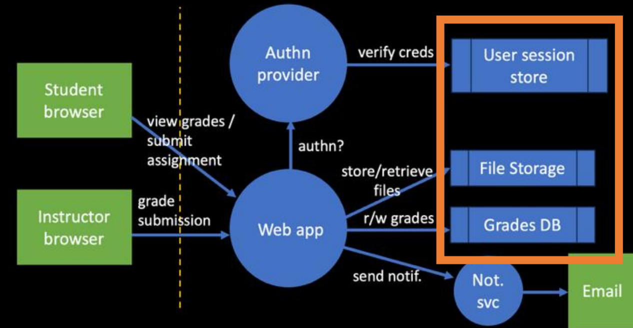
# Symmetric Encryption: AES-GCM

**Examples**

- Payment app: linked bank account numbers encrypted in DB
- CMS: student records subject to FERPA encrypted at rest




(DB not shown)



# Key Management: The Hard Part

Encryption is only as good as the key management

- **Never hardcode keys** in application source or binaries – decompiling a mobile app is trivial
- **Mobile devices:** use the OS keychain/keystore (iOS Keychain, Android Keystore)
- **Servers:** use a KMS (Key Management Service)
- **Hardware:** secure elements and TPMs store keys in tamper-resistant hardware
- Encrypt with **per-entity keys** so compromise of one key doesn't expose everything

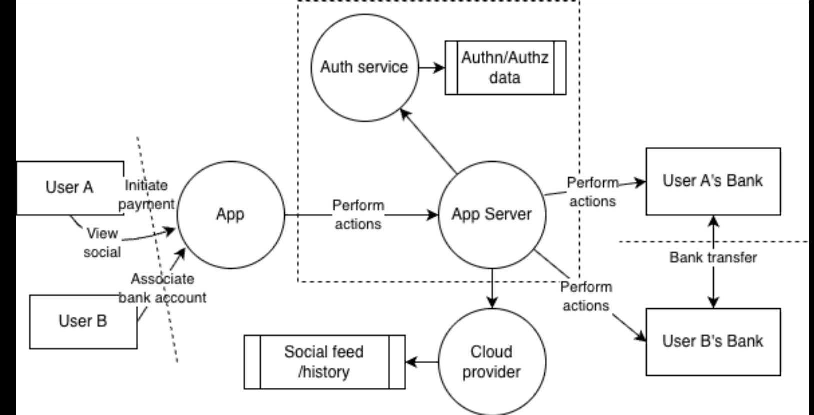
# Authentication – Proving Identity

# Why Authentication Is Foundational

**Authentication** answers: *who are you?*

**Payment app authentication points:**

- User → App (Server)
- App Server → Bank API
- App Server → Cloud API

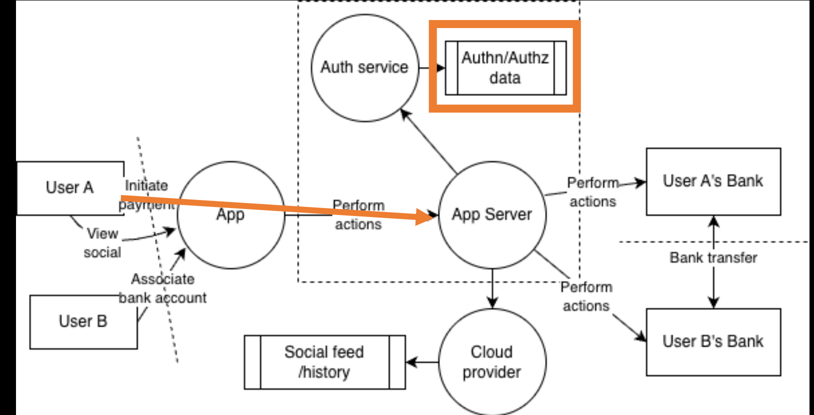



**Primary STRIDE threat: Spoofing**

# Client/Server authentication

User → App (Server)

- Typical approach: Username/password
- How? Store database of username/password pairs, authenticate claimed username against provided password
- Use encryption to protect the password in transit
    - More protection needed …



# The Password Database Threat

Need for **Defense in Depth:** An attacker obtains the password database (SQL injection, cloud misconfiguration, insider).

How should passwords be stored?

| Storage Method | Vulnerability |
|---|---|
| Plaintext | Catastrophic – attacker reads every password |
| Simple hash (SHA-256) | Rainbow tables, brute force at billions/sec |
| Salted hash | Defeats rainbow tables, but fast hashes still brute-forceable |
| **Slow/memory-hard hash** | Deliberately expensive – the right answer |

# Password Hashing

## Argon2: new generation of memory-hard functions for password hashing and other applications


Alex Biryukov
University of Luxembourg
alex.biryukov@uni.lu

Daniel Dinu
University of Luxembourg
dumitru-daniel.dinu@uni.lu

Dmitry Khovratovich
University of Luxembourg
khovratovich@gmail.com



***Abstract*—We present a new hash function Argon2, which is oriented at protection of low-entropy secrets without secret keys. It requires a certain (but tunable) amount of memory, imposes prohibitive time-memory and computation-memory tradeoffs on memory-saving users, and is exceptionally fast on regular PC. Overall, it can provide ASIC- and botnet-resistance by filling the memory in 0.6 cycles per byte in the non-compressible way.**


### I. Introduction

Passwords, despite all their drawbcks, remain the primary form of authentication on various web-services. Passwords are usually stored in a hashed form in a server's database. These databases are quite often captured by the adversaries, who then apply dictionary attacks since passwords tend to have low entropy. Protocol designers use a number of tricks to mitigate these issues. Starting from the late 70's, a password is hashed together with a random *salt* value to prevent detection of identical passwords across different users and services. The hash function computations, which became faster and faster due to Moore's law have been called multiple times to increase the cost of password trial for the attacker.

In the meanwhile, the password crackers migrated to new architectures, such as FPGAs, multiple-core GPUs and dedicated ASIC modules, where the amortized cost of a multiple-iterated hash function is much lower. It was quickly noted that these new environments are great when the computation is almost memoryless, but they experience difficulties when operating on a large amount of memory. The defenders responded by designing *memory-hard* functions, which require a large amount of memory to be computed, and impose computational penalties if less memory is used. The password hashing scheme scrypt [13] is an instance of such function.

Memory-hard schemes also have other applications. They can be used for key derivation from low-entropy sources. Memory-hard schemes are also welcome in cryptocurrency designs [11] if a creator wants to demotivate the use of GPUs and ASICs for mining and promote the use of standard desktops.

*a) Problems of existing schemes:* A trivial solution for password hashing is a keyed hash function such as HMAC. If the protocol designer prefers hashing without secret keys to avoid all the problems with key generation, storage, and update, then he has few alternatives: the generic mode PBKDF2, the Blowfish-based bcrypt, and scrypt. Among those, only scrypt aims for high memory, but the existence of a trivial time-memory tradeoff [7] allows compact implementations with the same energy cost.

Design of a memory-hard function proved to be a tough problem. Since early 80's it has been known that many cryptographic problems that seemingly require large memory actually allow for a time-memory tradeoff [10], where the adversary can trade memory for time and do his job on fast hardware with low memory. In application to password-hashing schemes, this means that the password crackers can still be implemented on a dedicated hardware even though at some additional cost.

Another problem with the existing schemes is their complexity. The same scrypt calls a stack of subprocedures, whose design rationale has not been fully motivated (e.g, scrypt calls SMix, which calls ROMix, which calls BlockMix, which calls Salsa20/8 etc.). It is hard to analyze and, moreover, hard to achieve confidence. Finally, it is not flexible in separating time and memory costs. At the same time, the story of cryptographic competitions [12], [15] has demonstrated that the most secure designs come with simplicity, where every element is well motivated and a cryptanalyst has as few entry points as possible.

The Password Hashing Competition[1], which started in 2014, highlighted the following problems:

- Should the memory addressing (indexing functions) be input-independent or input-dependent, or hybrid? The first type of schemes, where the memory read location are known in advance, is immediately vulnerable to time-space tradeoff attacks, since an adversary can precompute the missing block by the time it is needed [3]. In turn, the input-dependent schemes are vulnerable to side-channel attacks [14], as the timing information allows for much faster password search.
- Is it better to fill more memory but suffer from time-space tradeoffs, or make more passes over the memory to be more robust? This question was quite difficult to answer due to absence of generic tradeoff tools, which would analyze the security against tradeoff attacks, and the absence of unified metric to measure adversary's costs.
- How should the input-independent addresses be computed? Several seemingly secure options have been at-

[1] Argon2 was selected as the PHC winner in July 2015.

RFC 9106: Argon2 Memory-H ×

rfc-editor.org/rfc/rfc9106

Gemini

Relaunch to update

Table of Contents

| | |
|---|---|
| Status: | Informational |
| More info: | Errata exist \| Datatracker \| IPR \| Info page |


| | |
|---|---|
| Stream: | Internet Research Task Force (IRTF) |
| RFC: | 9106 |
| Category: | Informational |
| Published: | September 2021 |
| ISSN: | 2070-1721 |



Authors:
A. Biryukov — *University of Luxembourg*
D. Dinu — *University of Luxembourg*
D. Khovratovich — *ABDK Consulting*
S. Josefsson — *SJD AB*


## RFC 9106
## Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications

### Abstract


This document describes the Argon2 memory-hard function for password hashing and proof-of-work applications. We provide an implementer-oriented description with test vectors. The purpose is to simplify adoption of Argon2 for Internet protocols. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

# Password Hashing: Argon2id

**Argon2id** is the current best practice. Bcrypt, scrypt alternatives.

- Tunable cost parameters: time cost, memory cost, parallelism.

**Practical guidance:**

- Store: `salt || hash` (salt is not secret, just unique per user)
- Don't truncate or limit password length unreasonably
- Check passwords against breach databases (HaveIBeenPwned API)

# Multi-Factor Authentication

**something you know + something you have + something you are**

| Factor | Examples | Notes |
|---|---|---|
| Knowledge | Password, PIN | Can be phished |
| Possession | Phone, authenticator, FIDO2 key | FIDO2/WebAuthn is phishing-resistant |
| Biometric | Fingerprint, face | Can’t be changed if compromised |

# Multi-Factor Authentication and Sessions

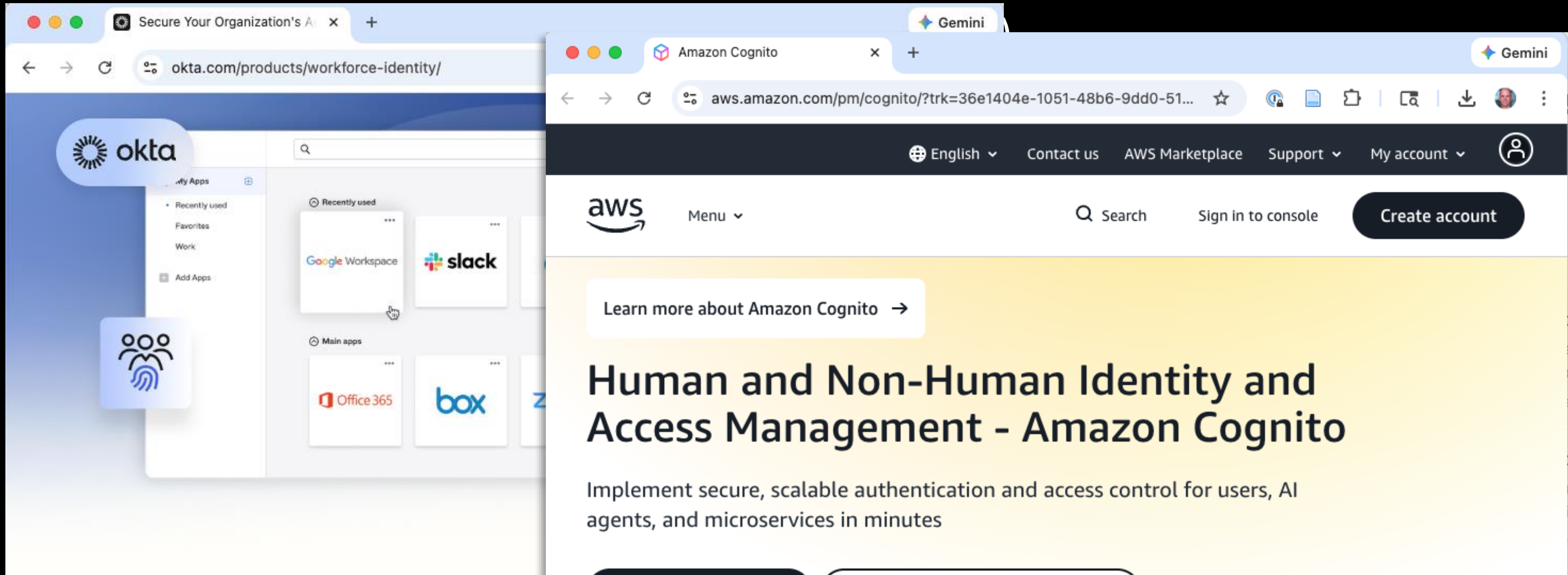
- **Payment app:** MFA required – you're protecting money
  - TOTP or FIDO2 preferred; SMS good, but vulnerable to SIM swap.
- **CMS:** MFA for instructors (who can change grades) is higher priority than for students
  - But MFA-for-all makes sense once you have it

**Session tokens** must be random, transmitted only over TLS, time-limited, revocable.

# Identity Providers: Don't Roll Your Own Auth

Just as you shouldn't roll your own crypto, consider not rolling your own auth.

**Identity providers (IdPs)** handle the hard parts



# Identity Providers: Don’t Roll Your Own Auth

Just as you shouldn’t roll your own crypto, consider not rolling your own auth.

**Identity providers (IdPs)** handle the hard parts:

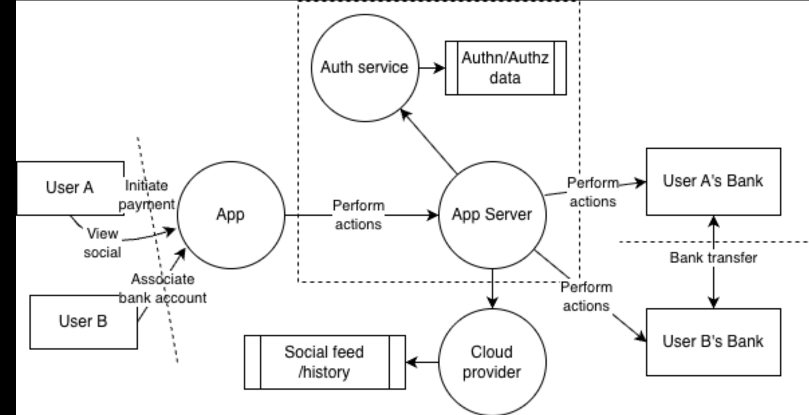
- Credential storage (bcrypt/Argon2, auto-updated)
- MFA management (TOTP, FIDO2, push, SMS)
- Session management (secure tokens, refresh, expiration, revocation)
- SSO via SAML or OpenID Connect
- Breached password detection
- Rate limiting and bot detection
- Compliance certifications (SOC 2, HIPAA, PCI-DSS)

# Mutual Authentication and API Identity

In client-server systems, the server authenticates to the client (TLS certificate) and the client authenticates to the server (password/token).

**But what about server-to-server?**

- **Mutual TLS (mTLS):** Both sides present certificates. Payment app server ↔ bank API: both sides verify each other.
- **Mobile app → API server:** OAuth 2.0 tokens, device attestation (Google Play Integrity, Apple App Attest). Never a hardcoded API key




**For every arrow in your DFD that crosses a trust boundary:**

*How does each side know who it's talking to?*

# Do Not Expect Expert Users

# The Principle: Psychological Acceptability

**Saltzer & Schroeder:** Security mechanisms should not make the system harder to use than it would be without them.

If security is annoying, users will bypass it.

Software designers should consider how the mindset and abilities of *the least sophisticated* users will affect security.

**The overarching lesson:**

> If your security depends on users doing something reliably, and evidence says they won't, the mitigation must be a **design change**, not a **policy change**.

**Category:** Prevention

# Fail-safe Defaults and the Safety-Security Tension

# The Fail-safe Defaults Principle

**S&S:** Base access decisions on *permission* rather than *exclusion*. The default is “denied.”

**Applied broadly:**

- Configuration choices that affect security should default to the secure option
- Define what *is* permitted (allowlist) rather than what *isn’t* (denylist)
    - Attackers only need to find something the denylist *missed*
    - With an allowlist, they must find something already on the list

**Category:** Prevention

# Secure Default Configurations

| Setting | Insecure Default | Secure Default |
| --- | --- | --- |
| Course visibility (CMS) | Public | Private |
| API access (CMS) | Enabled | Disabled, requires admin opt-in |
| Session timeout | “Forever” | Reasonable duration |
| File uploads (CMS) | All types | Allowlisted types only |
| Encryption | Optional | Required (TLS everywhere) |
| Social feed (Payment app) | Public | Private |
| Default credentials | admin/admin | No default – must set on first use |

# Designing for Failure

**Fail-secure:** When a component fails, deny access.

- CMS auth service goes down → no one can view grades until it’s back
- Protects confidentiality and integrity

**Fail-safe (safety sense):** When a component fails, protect human welfare.

- If the payment app’s fraud detection is unreachable: block all transactions (secure but availability-destroying) or allow them (available but risk-accepting)?

**These can conflict.**

# Availability, DoS, and the Safety-Security Tension

**Denial-of-Service mitigations:**

- **Rate limiting:** Limit requests per user/IP. But don't lock out legitimate users during peak usage.
- **Account lockout tradeoffs:** Lock after N failed attempts? This prevents brute force but lets an attacker lock out any user. Alternatives: progressive delays, CAPTCHA.

**The safety-security tension:**

- A hospital records system that locks out an ER doctor during a critical procedure because of a session timeout
- Design principle: make these tradeoffs *explicit and documented*
- Provide **"break glass" mechanisms** – emergency access that is granted immediately but logged and reviewed after the fact

# Favor Simplicity

# Economy of Mechanism

Keep the design so simple that it is **obviously correct**, rather than so complex that it has **no obvious errors**.

**S&S:** Keep the security-critical code as small and simple as possible – every line is a potential source of flaws.

> "The more complex a system is – the more options it has, the more functionality it has, the more interfaces it has, the more interactions it has – the harder it is to analyze its security."
>
> – Bruce Schneier

**Category:** Prevention

# Simplicity in the External Interface

The question is not just “is this feature useful?” but “is this feature worth the security cost?” **Every feature is attack surface.**

| Complex API | Simple API |
|---|---|
| CMS grade API: 15 optional parameters (format, encoding, timezone, locale, override flags, …) | Three required fields: `student_id`, `assignment_id`, `grade` |
| Payment transfer: dozens of optional fields (memo, scheduled date, recurring, split, currency conversion, promo code, …) | One thing: move amount X from A to B |

The simpler API has a smaller attack surface and is easier to validate.

# Simplicity in Design and Implementation

If the security-critical code path fits on a whiteboard, you can reason about its correctness.

If it spans multiple services, event queues, and caches – good luck.

**The goal:** Keep complexity *out of the security-critical path*.

- Push complexity to non-security-critical components
- Prefer well-understood mechanisms over clever ones
- When in doubt, choose the boring technology

**Next lecture:** We'll concretize this as the **Trusted Computing Base** concept.

# Defense in Depth

# Recap in one slide

DID: No single mechanism is sufficient. Layer defenses so that a failure in one doesn’t compromise the system

**Payment app examples** we’ve seen so far

- Attacker steals session token → per-transaction re-authentication prevents transfers
- TLS (transport encryption) bypassed → database is encrypted at rest
- Password database is stolen → passwords hashed

# Lecture 1 Summary

| Topic | Techniques | Key Principles |
|---|---|---|
| **Confidentiality** | TLS, BLE security, AES-GCM, key management | Open design, Defense in depth |
| **Authentication** | Argon2id, MFA, IdPs, mTLS | Separation of privilege, Defense in depth |
| **Usable security** | Password managers, FIDO2 | Don’t expect expert users |
| **Fail-safe defaults** | Secure defaults, break glass, graceful degradation | Fail-safe defaults |
| **Simplicity** | Minimal APIs, boring technology | Economy of mechanism |

**Next time:** Authorization, integrity, trust minimization, compartmentalization, monitoring – and pulling it all together into a security architecture.

# References


- Saltzer, J.H. & Schroeder, M.D. “The Protection of Information in Computer Systems” (1975)
- Schneier, B. *Secrets and Lies: Digital Security in a Networked World* (2000)
- Ur, B. et al. “Does My Password Go Up To Eleven?” (USENIX Security ’12)
- Cranor, L.F. “A Framework for Reasoning About the Human in the Loop” (UPSEC 2008)
- NIST SP 800-63B: Digital Identity Guidelines – Authentication and Lifecycle Management
- OWASP Authentication Cheat Sheet
- Biryukov, A. & Khovratovich, D. “Argon2” (Password Hashing Competition, 2015)