Secure Password Manager

Password Hashing with Salt Implementation

Vaibhav Bajoriya
IMT2022574

International Institute of Information Technology, Bangalore

October 2025

Outline

1. Introduction
2. Algorithms & Research
3. Implementation
4. Results & Testing
5. Key Learnings
6. Conclusion

Problem Statement

Project Goal

Create a small password manager that hashes and stores passwords securely using **salting** and **hashing** techniques.

Why This Matters?

- 81% of data breaches involve weak or stolen passwords
- Storing passwords in plain text is extremely dangerous
- Rainbow table attacks can crack billions of hashes
- User credentials must be protected even if database is breached

Objectives:

- Implement secure password hashing with bcrypt
- Generate unique salt for each password
- Prevent rainbow table and brute-force attacks
- Demonstrate secure credential storage and verification

Cryptographic Hash Functions

What is a Hash Function?

- One-way mathematical function: `password` $\rightarrow$`hash`
- Fixed output size (e.g., 256 bits)
- Cannot reverse: `hash` $\not\rightarrow$ `password`
- Same input always produces same output (deterministic)

Key Properties:

1. **Deterministic:** Same input = same output
2. **One-way:** Cannot reverse the hash
3. **Avalanche Effect:** Small change $\Rightarrow$completely different hash
4. **Collision Resistant:** Extremely hard to find two inputs with same hash

Algorithm Comparison

Algorithm	Speed	Security	Status
MD5	Very Fast	Broken	Never use
SHA-1	Fast	Deprecated	Avoid
SHA-256	Fast	Good	Not for passwords
bcrypt	**Slow**	**Strong**	**Chosen**
Argon2	Configurable	Very Strong	Future work

Why bcrypt?

- Adaptive: Configurable work factor (future-proof)
- Built-in salt: Automatic unique salt generation
- Battle-tested: Used since 1999 by major companies
- Slow by design: Resists brute-force attacks
- Work factor: $2^{12} = 4,096$ iterations

Salt: Defense Against Rainbow Tables

Without Salt: Vulnerable!

- User A: `password123` →`hash_a3f5e8b2`
- User B: `password123` →`hash_a3f5e8b2` (same hash!)
- Attacker: Pre-compute hashes for common passwords

With Salt: Secure!

- User A: `password123 + salt_1` →`hash_9d2e4f7a`
- User B: `password123 + salt_2` →`hash_1c5f8e3b` (different!)
- Attacker: Must compute rainbow table for each salt value

Salt Properties:

- 16 random bytes (128 bits) per password
- Stored with hash (not secret, just unique)
- Makes pre-computation attacks impractical

Key Stretching: Slowing Down Attacks

Concept: Hash the password multiple times

bcrypt with 12 rounds = 2^{12} = 4,096 iterations

Each password is hashed 4,096 times before storage

Impact on Attack Speed:

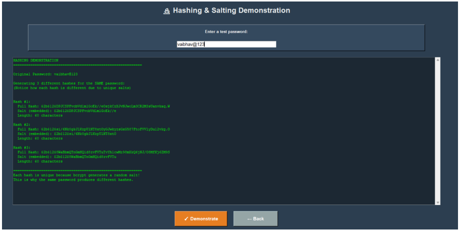
- **Legitimate user:** 1 login $\times$ 100ms = 100ms (acceptable)
- **Attacker:** 1 billion guesses $\times$ 100ms = 31.7 years!

Benefits:

- Reduces attack speed from billions to thousands of guesses/second
- Future-proof: Can increase rounds as hardware improves
- Transparent to users: 100ms is imperceptible

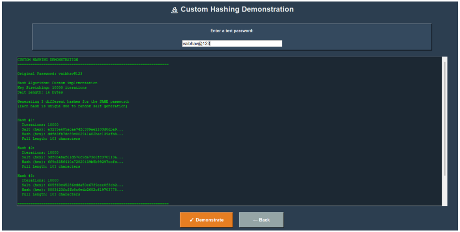
Two Implementations

1. Using Bcrypt



Hash Format: `$2b$12$[22 chars salt][31 chars hash]`

2. Custom Implementation



Hash Format: `10000$[salt_hex]$[hash_hex]`

Learning Value

Custom implementation demonstrates principles, but production systems should always use bcrypt/Argon2

Core Implementation

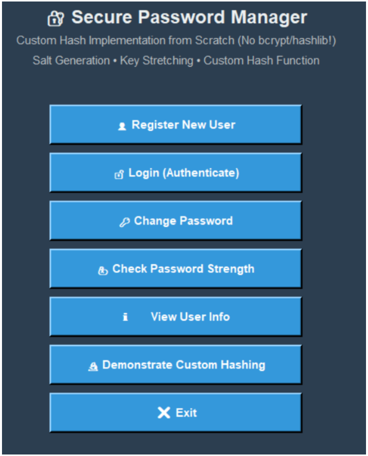
1. **Password Hashing (bcrypt)**
   - Generate salt with 12 rounds, Hash password automatically and Salt embedded in output

2. **Password Verification**
   - Extract salt from stored hash, then Re-hash input password with same salt, then Constant-time comparison (prevents timing attacks)

3. **Custom Hash Function (Inspired by SHA-256 Principles):**

   1. **Initialize** with 8 prime numbers, then **Pad** input data to fixed chunk size then **Process** chunks with:
      - Bit rotation (diffusion)
      - XOR operations (non-linearity)
      - Prime multiplication (distribution)
      - Multiple mixing rounds
   2. **Apply** 10,000 iterations (key stretching)
   3. **Output** 256-bit hash

Features Implemented



Storage Format: JSON database

- Username →password_hash, email, timestamps
- No plain-text passwords ever stored

Security Analysis

Attack Type	Defense
Rainbow Tables	Unique salt per password
Brute Force	10,000+ iterations
Dictionary Attack	Password strength requirements
Timing Attack	Constant-time comparison
Database Breach	Only hashes stored

Defense in Depth:

1. Hashing: One-way function
2. Salting: Unique per password
3. Key Stretching: 4,096-10,000 iterations
4. Validation: Minimum strength requirements

Technical Learnings

1. **Hashing vs Encryption**
   - Hashing is one-way (cannot reverse)
   - Encryption is two-way (can decrypt)
   - Passwords should be hashed, not encrypted
2. **Salt is Essential**
   - Without salt: same password $\Rightarrow$same hash
   - With salt: same password $\Rightarrow$different hashes
   - Prevents rainbow table attacks
3. **Key Stretching Matters**
   - More iterations = slower attacks
   - 100ms delay is acceptable for users
   - Makes attacks 10,000$\times$ more expensive

Implementation Insights

1. **Bit Manipulation**
   - Bit rotation, XOR, modular arithmetic
   - Creates avalanche effect in hashing
2. **Timing Attacks are Real**
   - Response time can leak information
   - Constant-time comparison is necessary
   - Even milliseconds matter
3. **Don't Roll Your Own Crypto**
   - Custom implementation: educational only
   - Production systems: use bcrypt/Argon2
   - Battle-tested libraries are essential

Security Principles

1. **Defense in Depth**
   - Multiple security layers
   - Hashing + Salt + Key Stretching + Validation
2. **Fail Securely**
   - Error messages don't reveal information
   - Prevents username enumeration
3. **Balance Security with Usability**
   - 100ms hash time is imperceptible
   - Strong requirements without frustrating users
4. **Trust but Verify**
   - Use established libraries
   - Understand principles by implementing them

Challenges & Solutions

1. **Understanding Key Stretching**
   - *Problem:* Why hash multiple times?
   - *Solution:* Each iteration multiplies attacker's work
2. **Salt Storage**
   - *Problem:* How to store salt securely?
   - *Solution:* Salt doesn't need to be secret, just unique
3. **Constant-Time Comparison**
   - *Problem:* Standard comparison leaks timing
   - *Solution:* XOR-based comparison always processes all bytes
4. **Hash Function Design**
   - *Problem:* Creating proper avalanche effect
   - *Solution:* Bit rotation + XOR + prime multiplication

Real-World Applications

These principles apply to:

Systems:

- Web applications
- Mobile apps
- Desktop software
- API authentication
- Password managers

Companies Using bcrypt:

- Facebook
- GitHub
- Twitter
- Stack Overflow
- Many Fortune 500

Industry Standard

bcrypt has been the gold standard for password hashing since 1999

Future Enhancements

Potential Improvements:

1. **Upgrade to Argon2**
   - Winner of Password Hashing Competition (2015)
   - Better resistance against GPU attacks
2. **Multi-Factor Authentication**
   - OTP via email/SMS
   - Google Authenticator integration
3. **Enhanced Security**
   - Account lockout after failed attempts
   - Password expiry policies
   - Breach detection (Have I Been Pwned API)
4. **Production Features**
   - Migrate to PostgreSQL with encryption
   - Session management
   - Password recovery mechanism

Key Takeaways

Most Important Lesson

Never store passwords in plain text. Ever.

Remember:

1. Simple hashing is insufficient
2. Salt and key stretching are essential
3. Implementation details matter (timing attacks, storage)
4. Use proven libraries for production
5. Understanding principles makes you a better developer

"Security is not an afterthought—it's a design principle."

Thank You!

Vaibhav Bajoriya
IMT2022574

GitHub Repository:
`https://github.com/bajoriya-vaibhav/Cyber_Security_Project`

Questions and Discussion