Password Rotation and CRUD-Based Authentication Management Scheme

Obinexus Computing
Nnamdi Michael Okpala

computing from the heart

April 2025

Executive Summary

This white paper introduces a standardized password lifecycle management scheme based on Create, Read, Update, Delete (CRUD) principles for authentication systems. It addresses challenges in password security by enforcing strong storage practices and regular rotation. Users create accounts with securely salted and hashed passwords, and passwords are never stored or transmitted in plaintext. Routine password updates are encouraged on an annual basis, aligning with modern guidelines that discourage overly frequent changes. The scheme also incorporates mechanisms to prevent immediate password reuse and supports secure password invalidation or account deletion processes. By combining unique salting, hashing, and multi-layered security (e.g., optional peppering and two-factor authentication), the approach ensures that even if a database is compromised, attackers cannot easily derive the original passwords or gain unauthorized access. This model can be implemented on any platform via back-end logic, providing developers and security engineers a clear blueprint for robust user authentication and authorization management.

1 Introduction

User authentication is the cornerstone of security for web and mobile applications. Weak or poorly managed password systems can lead to unauthorized access, data breaches, and erosion of user trust. Industry reports continue to highlight that breached passwords remain one of the most common cybersecurity threats facing organizations. As applications scale, developers and security engineers need a consistent strategy to handle user credentials safely throughout their lifecycle. Recent guidelines and best practices from standards bodies (such as NIST and OWASP) emphasize the importance of strong password storage (hashing and salting), prudent rotation policies, and additional defense layers like two-factor authentication.

This white paper, authored by *Obinexus Computing* (Nnamdi Michael Okpala), presents a CRUD-based Password Rotation and Authentication Management Scheme. Branded as "computing from the heart," this approach is a heartfelt yet rigorous model designed to fortify authentication systems. The intended audience is software developers and security engineers responsible for implementing or auditing authentication flows in web and mobile applications. The goal is to provide a clear, standardized blueprint that can be universally applied to manage passwords securely—from the moment a user creates a password, through regular use and updates, to eventual deletion or deactivation.

2 Problem Statement

Despite advancements in security, many applications still suffer from inadequate password management. Common issues include:

- **Insecure Password Storage:** Storing passwords in plaintext or with weak hashing allows attackers to retrieve credentials if the database is breached. A lack of salting means that identical passwords can be trivially identified across accounts, amplifying the damage of a single leaked hash. This violates fundamental security guidelines that insist passwords should be hashed (one-way) and never stored reversibly.
- **Predictable Password Practices:** When users are forced to change passwords too frequently under weak policies, they often resort to predictable patterns (e.g., incrementing a number or adding a symbol). Attackers are aware of these tendencies; if a previous password

is known or compromised, minor variations (like "Password1" to "Password2") are easily guessed. Conversely, allowing the same password indefinitely gives attackers unlimited time to crack or misuse it.

- **Lack of Standard Lifecycle Enforcement:** Many systems do not implement a comprehensive lifecycle for passwords. Users might not be reminded or forced to update credentials for years, or there may be no mechanism to retire old passwords. Without enforcing some rotation or having a password history policy, users could reuse old (potentially compromised) passwords, reintroducing known vulnerabilities into the system. Additionally, insufficient processes for account or credential deletion (such as not properly removing an outdated password or associated 2FA data) can leave security holes.

These problems highlight a gap: authentication systems need a balanced approach that avoids both extreme laxity and counterproductive rigidity. The solution lies in a scheme that secures password storage and verification, while promoting periodic updates that are manageable for users and effective against attackers.

3 Context

In response to the above challenges, security standards have evolved. Historically, organizations enforced password changes every 30, 60, or 90 days, but research and updated standards have shown that overly frequent rotations can harm security more than help. NIST's 2024 Digital Identity Guidelines, for example, recommend against arbitrary frequent resets, suggesting password expiration only in cases of known compromise or at most once per year. The rationale is to encourage users to choose longer, stronger passwords and reduce reliance on simplistic tweaks to old passwords.

Modern best practices emphasize:

- **Strong Hashing and Salting:** Passwords should be transformed into secure hashes with a unique salt per password entry. Salting each password means that even if two users choose the same password, their stored hashes will differ, preventing attackers from recognizing duplicate passwords in a database dump. Hashing algorithms like Argon2id, bcrypt, or PBKDF2 are recommended, as they are designed to be slow and resist brute-force attacks. These algorithms also automatically handle salting in their process.

- **Limited Password Lifetime with Sensible Rotation:** Rather than forcing constant changes, the strategy is to set a reasonable maximum password age (such as 1 year) after which users must update their password. This aligns with the NIST guidance of annual rotations, striking a balance between never changing passwords and changing them too often. Moreover, systems often maintain a password history to prevent immediate reuse of recent passwords. This ensures that if a password is changed, the user cannot switch back to a recently used password for a defined period (for example, one year or a certain number of iterations).

- **Multi-Layered Security:** Beyond passwords, additional layers like two-factor authentication (2FA) are now commonplace. While 2FA is outside the scope of password lifecycle management per se, any robust scheme should coexist with such measures. For instance, if a user's 2FA is tied to their account, the system should accommodate disabling 2FA or regenerating 2FA secrets as part of the credential management process. Another layer could include "peppering" the passwords: using a secret key (stored separately from the database) to HMAC or encrypt hashes, adding protection in case the database alone is compromised.

Despite these well-publicized practices, implementing them correctly across the entire lifecycle is non-trivial. Developers may use frameworks that hash passwords, but unless the full create-read-update-delete cycle is managed, gaps can appear (for example, not handling password updates with the same rigor as initial creation, or failing to properly dispose of credentials on account deletion). This context underscores the need for a unified scheme, which we present next.

4 Gap Analysis

There is a clear gap between available security know-how and its consistent application:

- **Fragmented Practices:** Some applications hash passwords but do not enforce any rotation; others enforce rotation but still use outdated hashing (like unsalted SHA-1), undermining the benefit. The lack of a unified approach means security is only as strong as the weakest link in the lifecycle.
- **Developer Burden and Errors:** Without a blueprint, individual developers implement authentication in ad-hoc ways. Critical steps (salting, using proper hash functions, checking password history, etc.) might be overlooked under delivery pressure. A standardized model can reduce misconfigurations and ensure nothing is missed.
- **User Experience vs Security:** The gap also manifests in failing to balance UX and security. Strict policies (e.g., monthly mandatory changes with complex composition rules) frustrate users and lead to workarounds or poor choices. On the other hand, lenient approaches (never expiring passwords) increase risk. The proposed scheme intends to bridge this gap by providing security by design while remaining practical for users (yearly changes, predictable but secure patterns like suffix increments, etc.).
- **Lifecycle Completion:** Finally, not all systems handle the "Delete" aspect well. For instance, when users delete their account or an admin disables an account, residual data like password hashes or 2FA tokens might linger. Ensuring a clean deletion (or invalidation) process is part of the gap that needs addressing.

In summary, the absence of a comprehensive, easy-to-follow standard leaves many implementations either incomplete or misaligned with best practices. The next section introduces our solution to fill this gap: a CRUD-based password management framework that can be uniformly enforced.

5 Solution Overview: CRUD-Based Password Lifecycle

We propose a solution that treats password management as a predictable, CRUD-oriented lifecycle. In this model, the four stages of password handling are:

5.1 Create (Sign-Up / Onboarding)

The **Create** phase occurs when a user sets up their account with a password for the first time. The system must securely handle this initial password creation:

- Users choose a password (e.g., `nna2001` in our example schema). The raw password is immediately processed on the server side; it should never be stored or logged in plaintext.

- A random **salt** is generated. This salt is a unique per-user (per-password) random string or byte sequence. It will be used to harden the password before hashing.
- The system computes a **hash** of the password combined with the salt, using a strong one-way hashing algorithm (such as Argon2id, bcrypt, or PBKDF2). For instance, `hash = HashFunc(password + salt)`. Modern password hashing libraries perform salting internally. The hash result is what will be stored in the database, along with the salt. The original password is not stored, and indeed cannot be recovered from the hash.
- The new user record is created in the authentication datastore with fields like username, the salt, the hashed password (and possibly metadata like password creation date, password expiration date, and an optional password history list or flag).
- Optionally, if multi-factor authentication is offered at sign-up, the user may also set up 2FA during this stage. Any 2FA secrets (such as an OTP seed) should be stored separately and securely (often encrypted or in a secure vault) and linked to the user account.

The result of the Create phase is a securely stored credential. Even if an attacker were to see the database at this point, they would find only a salted hash. Because of salting (and peppering, if applied), they cannot use precomputed rainbow tables to crack the password, and because of hashing, they cannot retrieve the plaintext password at all.

5.2 Read (Authentication/Login)

The **Read** phase pertains to verifying credentials, essentially the login process. The term "Read" in CRUD is a slight misnomer here, since the system never actually reads the password in plaintext, but rather checks the user's provided password against the stored hash:

- When a user attempts to log in, they provide their username (or email) and password via a secure channel (TLS-protected POST from a web or mobile app). The client-side should not hash the password, as hashing is best done server-side using the server's salt and secrets to avoid vulnerabilities.
- The server retrieves the stored salt and hashed password for the account identified by the username. It then concatenates the provided password with the salt and applies the same hash function (and pepper, if used) as was done during account creation.
- The newly computed hash is then compared with the stored hash in a time-constant manner (to prevent timing attacks). If they match, the password is correct; if not, authentication fails.
- At no point is the stored hash "decrypted" – verification is done by hashing the input and comparing hashes, meaning the actual password remains unknown to the system beyond the moment of verification.
- Systems may implement additional checks at login, such as rate limiting (e.g., throttle or lock the account after a number of failed attempts) in line with security best practices, and secondary authentication if 2FA is enabled.

This read/check process ensures that even during authentication, the password itself is never revealed. Only the user knows their password. From the system's perspective, authentication is a matter of matching hash outputs, which preserves confidentiality of the credential.

5.3 Update (Password Rotation)

The **Update** phase involves changing an existing password, typically initiated by the user (or forced by policy). Our scheme envisions an annual password rotation policy:

- **Prompting Rotation:** If a user's password is nearing or has exceeded one year of age (since last set), the system can prompt the user to update it. This can be done at login (e.g., "Your password has expired, please set a new one") or via notifications before expiration.
- **User Chooses New Password:** Users are encouraged to use a schema or pattern that is easy to remember but still secure. For example, incrementing a year-based suffix: if the original password was `nna2001`, the next could be `nna2002`. This example shows a mnemonic pattern; however, the new password should not be too derivable if the old password was known by an attacker. In practice, users might combine a base phrase with a changing element. The important aspect is the change itself, not necessarily the format, as long as it meets the application's complexity requirements.
- **Salting and Hashing New Password:** Just like in account creation, a new salt can be generated (or the old salt can be reused, though generating a new salt for a new password is generally wise to treat each credential version independently). The new password is salted and hashed with the same algorithm. The stored hash and salt for the user are then updated to the new values. The password update timestamp is recorded.
- **Password History Check:** The system should enforce password history so that the user cannot immediately reuse an old password. One common policy is to disallow the last N passwords (for instance, last 5) or any password used in the last Y days. In our yearly rotation scheme, this effectively means a user should not cycle back to an earlier password for at least a year. Implementation-wise, the application can keep a short list of prior password hashes (with their salts) or, more securely, a hash of those hashes, to compare against new choices. Storing a password history must be done as carefully as storing the current password (hashed and salted) to avoid introducing a new vector of attack. An attempt to reuse a recent password would be detected by matching the hash of the candidate against the history list.
- **Optional Pepper Rotation:** If a pepper (application-wide secret key) is used in hashing, an update operation is a good opportunity to change the pepper value periodically (though pepper rotation can also be done independent of user action). The system would then re-hash existing passwords with the new pepper when users authenticate or via a background migration.

The annual rotation, as suggested, aligns with guidance to not force changes too frequently, while still ensuring that a credential isn't permanent. This limits the window of opportunity for an attacker who might have obtained an old password. Even if they got a password hash from a breach, by the time they crack it (if ever), the password may have already been changed by the legitimate user. Moreover, the old password would be disallowed by history checks, mitigating the risk of the attacker using stale credentials.

Below is pseudocode demonstrating how the update process can be implemented:

```
// Pseudocode for updating (rotating) a user's password
function updatePassword(username, currentPassword, newPassword):
   userRecord = database.findUser(username)
   if userRecord is None:
      return Error("User not found")
```

```
    // Verify current password
    if Hash(currentPassword + userRecord.salt) != userRecord.hashedPassword:
        return Error("Current password incorrect")
    // Enforce password history (check newPassword not equal to recent passwords)
    if isInPasswordHistory(userRecord, newPassword):
        return Error("New password must not reuse a recent password")
    // Everything OK, proceed to update
    newSalt = generateRandomSalt()
    newHash = Hash(newPassword + newSalt)
    database.update(username, { salt: newSalt, hashedPassword: newHash, lastChanged:
        now })
    recordPasswordHistory(userRecord, newHash)
    return Success("Password updated successfully")
end function
```

In this pseudocode, `isInPasswordHistory` would compare the hash of `newPassword` (perhaps hashed with each historical salt, or by storing hashes of old passwords in a normalized way) against the user's stored history. The `recordPasswordHistory` function would add the old password's hash (or some representation) to the history before replacing it. The exact method of storing and comparing historical passwords can vary; some systems store the last N password hashes, others store a hash of each previous hash to avoid keeping actual old hashes around (which might be cracked if the algorithm improves or if they were weaker).

5.4 Delete (Credential Invalidation and Account Deletion)

The **Delete** phase encompasses the secure invalidation or removal of a password or account. There are a few scenarios:

- **User-Initiated Account Deletion:** When a user decides to delete their account, the system should erase or anonymize personal data, including authentication credentials. The password hash and salt in the database should be securely destroyed (or the user record as a whole dropped). If full deletion is not immediately possible due to audit logs or legal holds, the password can at least be invalidated: e.g., replaced with a random hash that no one knows, effectively locking the account.
- **Password Reset/Invalidation by Admin or Security Process:** In cases of a suspected compromise, an administrator might want to invalidate a user's current password, forcing them to use a recovery flow to set a new one. This is akin to deletion of the credential, followed by prompting a Create flow (new password). The implementation could mark the account such that no login is accepted until a fresh password is set. This ensures a compromised password cannot be used, even if an attacker obtained it.
- **Removing 2FA or Other Adjuncts:** Sometimes users might want to remove a second factor (for example, they lose their device and want to disable 2FA, or simply opt-out). Removing 2FA data should be treated carefully: it should require re-authentication and possibly additional verification (since it's a security downgrade). Upon confirmation, the system deletes the stored 2FA secret/key and/or any backup codes associated with the account.
- **Session and Token Revocation:** As part of credential deletion or reset, it's important to invalidate active sessions or tokens. For example, if a user deletes their account or an admin resets their password, any existing authentication tokens (JWTs, session cookies, etc.) should be revoked. This prevents a scenario where an attacker with an active session continues to be authenticated after the password is changed or account removed.

Pseudocode for a simple account deletion might look like this:

```
// Pseudocode for deleting a user account
function deleteAccount(username, password):
   userRecord = database.findUser(username)
   if userRecord is None:
      return Error("User not found")
   // Verify the user's identity via password (and possibly 2FA)
   if Hash(password + userRecord.salt) != userRecord.hashedPassword:
      return Error("Authentication failed")
   // Remove sensitive data: password hash, salt, 2FA secrets
   database.update(username, { hashedPassword: null, salt: null, otpSecret: null })
   // Optionally mark account as deleted or remove completely
   database.deleteUser(username)
   // Revoke sessions or tokens (implementation depends on session management)
   sessionManager.revokeAllSessions(username)
   return Success("Account deleted and credentials purged")
end function
```

In practice, one might not set the hash and salt to null before deletion (one could just delete the record), but it illustrates the intention to remove all credential material. If a soft-delete is used (marking an account inactive rather than fully dropping the row), nullifying or randomizing the password hash ensures that even if the record lingers, it cannot be used to authenticate.

By completing the delete phase properly, we ensure no orphaned credentials remain to be exploited. This closes the loop on the CRUD cycle, covering the entire lifespan of a user password in the system.

6 Implementation Feasibility and Considerations

The CRUD-based scheme described is designed to be implementation-agnostic. Whether the application is built in Java, Python, JavaScript/Node.js, Go, or any other modern stack, the concepts remain the same. Most frameworks and languages provide libraries for secure password handling:

- **Hashing Libraries and Algorithms:** Use well-vetted functions (e.g., Argon2id via libsodium, BCrypt implementations, PBKDF2 via OpenSSL or standard libraries, etc.). These functions handle salting internally or allow salt input, and are tuned to be slow (configurable work factor) to thwart brute force. As noted, Argon2id is a recommended modern choice due to its resistance to GPU attacks and flexibility. BCrypt remains widely used and acceptable for many cases, and PBKDF2 (with high iteration counts) is FIPS-compliant for regulated industries.

- **Storing Salts:** Salts do not need to be secret; they should be unique. Typically, the salt is stored alongside the hash in the database record (often concatenated or in separate column). If using a hashing library that produces a combined output (like BCrypt which outputs a string containing version, cost, salt, hash), the entire output can be stored as the hashed password field.

- **Using Peppers:** If an additional pepper is used, it should not be stored in the database. The application would store it in a secure configuration (environment variable, secret management service, or HSM). Implementing peppering might involve doing something like: `hash = HashFunc(password + salt)`, then `storedHash = HMAC(pepper, hash)` or a similar construction. This means an attacker needs both the database and the separate pepper

value to crack passwords. However, peppering adds complexity in managing the secret key and rotating it, so it's an option for high-security contexts.

- **Password History Storage:** This can be implemented by a separate table or fields that log previous hashes. Care must be taken that these old hashes are also protected (e.g., if using BCrypt, store the full BCrypt output of old passwords). Some implementations choose to store a hash-of-hash to avoid keeping the actual old hashes around. When the user changes their password, the oldest entry in history can be dropped if exceeding the limit. All such operations should be atomic and secure.

- **Expiration and Notification:** The system should have a way to mark when a password was set, to calculate expiry. A nightly job or on-login check can notify users who are near or past expiry. This is a UX consideration to smoothly enforce the policy. Logging of these events (password change, expiration reminders, etc.) is also important for audit trails.

- **Front-End and API Considerations:** The enforcement of this scheme is primarily back-end. The front-end (web or mobile app) should simply relay user inputs securely to the server and handle responses (e.g., redirect to a password change form if the password is expired, etc.). Front-end should not attempt to implement its own password logic (like hashing on the client or imposing different validation) beyond guiding the user on password policy (e.g., showing strength meters, etc.).

- **Testing and Verification:** Implementers should include tests for all CRUD paths: creating accounts (ensure the hash is stored and plaintext is not), logging in (correct vs incorrect password), forced updates (password change flow, including history checks), and deletions (ensuring no login after delete, data removed). Penetration testing should be done to verify that the stored credentials cannot be easily retrieved or bypassed.

Implementing this model is feasible with current technology stacks. It primarily requires discipline and clarity in the authentication flow design. By following this scheme, developers can rely on a known-good pattern rather than inventing their own, reducing the chance of security oversights.

7 Conclusion

Passwords will continue to be a fundamental aspect of authentication for the foreseeable future, even as passwordless technologies emerge. It is therefore crucial to manage passwords in a secure yet user-conscious manner. The CRUD-based Password Rotation and Authentication Management Scheme provides a structured approach to do exactly that.

By treating password management as a lifecycle with defined create, read, update, and delete stages, security engineers can systematically address each part of the process. The scheme enforces that passwords are created securely (with hashing and salting), never exposed during use, regularly refreshed to mitigate long-term risks, and cleanly removed when no longer needed. This layered strategy (augmented by salting, and optionally peppering and multi-factor auth) ensures that even if one layer is broken (e.g., a database leak), other safeguards still protect user accounts.

The annual rotation policy strikes a balance between security and usability, aligning with modern best practices that discourage burdensome password policies. Users benefit by having a predictable, yearly reminder to update their credentials, ideally choosing a new password that is related enough to remember but different enough to be secure. Developers and administrators benefit from a clear framework that covers edge cases (like preventing re-use of old passwords and handling account deletions properly).

In essence, this white paper has presented not just a theoretical framework but a practical template. It is a model that can be "enforced by any platform or app via backend logic," meaning it can be integrated into existing systems with minimal disruption. Whether one is building a new application from scratch or tightening the security of an existing one, adopting a CRUD-based password management scheme is a step toward robust, standardized security.

Security is an ongoing journey, not a destination. While this scheme greatly enhances the security of password-based authentication, it should be complemented with other best practices such as user education, account lockout policies, monitoring for compromised passwords (e.g., checking against known breach databases), and continual updates to cryptographic algorithms as new threats emerge. By doing so, organizations truly embrace a comprehensive, heart-felt commitment to protecting their users—computing from the heart, with security in mind.

References

[1] OWASP Cheat Sheet Series. *Password Storage Cheat Sheet*. (2023). This resource provides guidelines on secure password hashing, salting, and peppering to protect stored passwords.

[2] NIST Special Publication 800-63B (Digital Identity Guidelines). (2024). Summary of password management recommendations, including annual password rotation and emphasis on password length over complexity.

[3] LoginRadius Blog. *Password History, Expiration, and Complexity: Explained!* (2021). Discussion on enforcing password history and expiration policies to improve security.

[4] AuditBoard. *NIST Password Guidelines*. (2024). An overview of updated NIST password guidelines highlighting hashing, salting, and reduced rotation frequency.