ivision

October 24, 2025

The Evolving Complexity of Modern Authentication Systems

The Evolving Complexity of Modern Authentication Systems

Authentication defines the boundary between users and protected systems. Over the past two decades, this boundary has evolved from simple username-password checks to complex ecosystems involving third-party identity providers, federated tokens, multifactor authentication, and multiple devices. Each advancement in security and usability has also introduced new implementation challenges and integration risks.

Modern authentication systems layer multiple technologies: OAuth flows, JWT tokens, biometric verification, hardware keys, SMS codes, and emerging technologies like passkeys. While each component addresses specific weaknesses, the overall complexity creates new attack surfaces and opportunities for misconfiguration.

This paper outlines key findings from ivision's security assessments of modern authentication systems, examining how the pursuit of better security and user experience has created new categories of implementation risks.roduction

1.1. About ivision

ivision provides security assessment services with deep experience testing authentication systems across web, mobile, and connected devices. The Security Assessment Team focuses on design flaws and implementation weaknesses that lead to real-world compromise.

1.2. The Evolution of Authentication Complexity

Early web applications maintained authentication state using simple server-side sessions and cookies. Users provided a username and password, the server verified these credentials against a database, and successful authentication resulted in a session cookie that persisted until logout or expiration.

As architectures shifted toward client-heavy applications, mobile apps, and distributed APIs, managing identity became significantly more complex. multifactor authentication emerged to address password weaknesses, introducing SMS codes, authenticator apps, and hardware tokens. Developers increasingly turned to third-party identity providers (IdPs) such as Auth0, Okta, and AWS Cognito to handle this complexity. OAuth and OpenID Connect (OIDC) became the dominant standards for authorization and authentication delegation.

Each layer of advancement addressed real security problems but created new integration challenges:

- **multifactor authentication** improved security but introduced timing attacks, bypass vulnerabilities, and user experience friction
- **Third-party IdPs** provided scalability and expertise but expanded trust boundaries and attack surfaces
- **JWT tokens** enabled stateless authentication but required careful signature validation and claim verification
- **Mobile and cross-device flows** improved user experience but complicated session management and device trust
- **Emerging technologies like passkeys** promise to eliminate password weaknesses but introduce new implementation and recovery challenges

This evolution improved overall security and user convenience, but also dramatically expanded the attack surface. Each integration point introduced potential misconfigurations, insecure defaults, and opportunities for bypass.

1.3. Modern Authentication Components: The Case of Passkeys

The complexity of modern authentication systems becomes clear when examining specific technologies like passkeys, which exemplify both the promise and pitfalls of advancing authentication security.

What Passkeys Are

A passkey is a cryptographic credential that replaces traditional passwords. It consists of a key pair generated and stored on a device. The private key never leaves the device; authentication occurs through a challenge-response process verified by the relying party. Passkeys use public-key cryptography to eliminate password reuse and phishing attacks.

Unlike passwords, passkeys are bound to the user's device and platform account. They integrate with biometric or local device authentication methods. The combination provides strong user verification without requiring the user to remember or manage credentials.

Why Passkeys Address Current Weaknesses

Passkeys represent an attempt to solve multiple problems in current authentication systems:

- They remove password reuse and phishing risk by replacing shared secrets with cryptographic keys
- Authentication cannot be replayed or intercepted because the private key never leaves the device
- Biometric unlock and hardware-based protection make credential theft significantly harder
- They potentially reduce the need for complex multifactor authentication flows

However, like other advanced authentication technologies, deployment still requires careful design. The loss of a device or synchronization failure between devices can degrade user experience and security if fallback paths are weak.

Cross-Device and Platform Challenges

Passkeys are stored in secure environments managed by the operating system or platform provider. They can synchronize across a user's devices using encrypted cloud storage. On mobile and desktop platforms, the same passkey can be used through local Bluetooth, QR, or platform-specific APIs to authenticate across devices. Roaming authenticators, such as FIDO2 hardware keys, allow passkeys to be portable but require explicit registration.

Each platform implements slightly different synchronization models, which affects recovery and migration. These differences must be considered when planning multi-device authentication, and they represent the type of complexity that modern authentication systems regularly introduce.

1.4. Integration Patterns and Their Complexities

Modern authentication systems typically follow several common integration patterns, each introducing distinct complexity and risk profiles:

Third-Party Identity Provider Integration

Applications delegate authentication to specialized providers like Auth0, Okta, AWS Cognito, or Google Identity. This pattern simplifies development but expands trust boundaries and attack surfaces. Organizations must understand the IdP's full feature set, default configurations, and exposed endpoints, even for features they don't intend to use.

Layered multifactor Authentication

Many applications add custom MFA layers on top of existing authentication systems. While this can improve security, it often creates bypass opportunities when enforcement is handled client-side or when different authentication stages aren't properly integrated.

Direct IdP Client Integration

Some applications allow client-side code to communicate directly with identity providers. This pattern requires exposing IdP configuration and credentials to untrusted clients, creating opportunities for credential theft and unauthorized API access.

1.4.1. Hybrid Approaches with New Technologies

Organizations increasingly combine traditional authentication with newer technologies like passkeys, which typically involve:

- Native platform integration through Apple, Google, or Microsoft identity frameworks
- Third-party IdP integration where passkey support is handled by an external provider
- Custom WebAuthn implementation managed directly by the application backend

Each integration pattern carries tradeoffs in complexity, control, and risk exposure. IdP integrations simplify deployment but expand trust boundaries. Custom implementations provide control but require expertise in protocol details like WebAuthn registration, attestation, and signature verification.

Server-side enforcement remains critical across all patterns. All security decisions, such as MFA checks, fallback handling, and token validation, must occur on the server, not in client-side code.

1.5. Common Pitfalls in Complex Authentication Systems

ivision assessments have identified recurring patterns of authentication weaknesses that emerge from system complexity:

- **IdP integration misuse**: Exposed client IDs and direct client-to-IdP communication create attack paths for unauthorized user creation or enumeration

- **Custom MFA layer bypasses**: Adding MFA outside of IdP control often leads to bypass vulnerabilities when enforcement is not properly integrated server-side
- **JWT validation errors**: Applications that parse tokens without verifying signatures or claims are easily compromised
- **Offline or fallback flow weaknesses**: Alternate operation modes, such as for offline functionality, can unintentionally disable authentication entirely
- **Weak IdP defaults**: Password policies, rate limiting, and session management are often insecure unless explicitly hardened
- **Client-side security enforcement**: Relying on client-side code to enforce authentication decisions creates trivial bypasses

These failures stem from the inherent complexity of modern authentication systems and misunderstanding of component behavior and interaction. While new technologies like passkeys may reduce some specific risks, they do not eliminate the fundamental challenges of secure integration and configuration validation.

1.6. Security Finding Examples from ivision Assessments

These examples are drawn from real-world assessments, with identifying details removed.

1.6.1. AWS Cognito: Client ID Used to Create Unauthorized Accounts

When an adversary obtains a Cognito client ID, they can interact directly with Cognito endpoints without authenticating to the application itself. This allows unauthorized user account creation through Cognito's APIs. The fundamental issue is that client IDs are designed to be public identifiers, but many applications fail to implement proper server-side validation of who should be allowed to create accounts through their Cognito user pool.

The exploitation chain becomes particularly dangerous when combined with additional weaknesses, such as bypassing email-invite prerequisites or inadequate user role validation. Once an adversary creates an unauthorized account through direct Cognito interaction, applications often fail to anticipate "unauthorized but authenticated" users. This leads to privilege escalation and workflow abuse, as the application trusts that any user with a valid Cognito JWT was created through the intended registration flow.

1.6.2. Custom MFA Bypass

A web application chose to implement its own multifactor authentication layer on top of AWS Cognito rather than using Cognito's native MFA capabilities. After users successfully authenticated with Cognito, they received valid JWTs, but the application required an additional MFA step enforced only by client-side JavaScript. The critical flaw was that the server-side API trusted any valid Cognito JWT for sensitive operations, regardless of whether the custom MFA step had been completed.

Attackers could bypass the MFA requirement entirely by intercepting the JWT after the initial Cognito authentication and using it directly against the API endpoints, skipping the client-side MFA enforcement. This vulnerability amplified the risks of credential stuffing attacks and use of stolen credentials from phishing campaigns, as the additional security layer provided no actual protection. The lesson is clear: all security enforcement must occur server-side, and custom authentication layers should be carefully integrated with the primary IdP's security model.

1.6.3. Client-to-IdP Direct Integration

Applications that allow client-side code to communicate directly with identity providers must expose IdP configuration details, client secrets, or API keys in the client. This pattern was observed in several assessments where mobile apps or single-page applications made direct calls to third-party identity services. The exposed credentials and configuration details created multiple attack vectors, including username enumeration through error message differences and unauthorized access to IdP APIs.

The attack surface extends beyond just credential exposure. Third-party identity providers often expose additional functionality through their APIs that applications don't intend to use, but which becomes available to attackers who can reverse-engineer the client configuration. A better architectural approach involves implementing a server-side proxy or wrapper API that mediates all communication with the IdP, keeping sensitive configuration and credentials on the server while providing only the necessary functionality to client applications.

1.6.4. IoT Shadow Auth Bypass

A cloud-enabled backyard grill used AWS Shadow IoT service for remote management, syncing system information to the cloud when Wi-Fi was configured. The mobile application could communicate with grills either via local Bluetooth or remotely through the Shadow service when out of Bluetooth range. The critical flaw was that Shadow service communication used only unauthenticated Cognito identity without any additional AWS-level authentication controls. To communicate with a specific grill, attackers needed only the last three bytes of the device's MAC address, which could be enumerated through AWS APIs or discovered in publicly accessible S3 buckets.

While the firmware included grill password verification functionality, this protection was easily circumvented through multiple attack paths. The reset password functionality required no authentication whatsoever, allowing attackers to reset and control any grill's password remotely. Additionally, grill passwords were stored in a public S3 bucket, making them directly accessible to unauthorized users. Through these combined vulnerabilities, attackers could remotely turn any grill on or off by enumerating MAC addresses and either resetting passwords or extracting them from the exposed S3 storage. This case demonstrates how IoT device security depends on the entire ecosystem, not just the device firmware itself.

1.6.5. JWT Without Verification

Several applications parsed JSON Web Tokens received from identity providers but failed to perform proper cryptographic verification of the token signatures or validate critical claims such as issuer, audience, and expiration time. This fundamental security failure allowed attackers to modify JWT contents arbitrarily, extending token validity indefinitely or changing user identity and privilege information within the token payload.

The impact of this vulnerability was severe: attackers could impersonate any user, including those with administrative privileges, simply by modifying the JWT claims and re-encoding the token. Since the applications trusted the token contents without signature verification, there was no way to detect the tampering. Proper JWT validation requires verifying the signature using the IdP's public key, confirming that the issuer and audience claims match expected values, and checking that the token hasn't expired. These validation steps are required for secure JWT implementation.

1.6.6. Weak Default Policies

Identity providers often ship with permissive default configurations that prioritize ease of initial setup over security. In one assessment involving Google Identity Toolkit, the default password policy allowed

passwords as short as six characters with no complexity requirements. This configuration made user accounts highly susceptible to credential stuffing attacks using common password lists and simple brute-force attempts.

The security impact extends beyond just weak passwords. Default configurations may also include overly permissive rate limiting, disabled account lockout mechanisms, and insufficient anomaly detection. Organizations must explicitly review and harden all IdP security settings before deployment, as vendor defaults are typically designed for quick demos rather than production security requirements. Regular audits of IdP configurations are essential, as provider updates can sometimes reset security settings or introduce new features with insecure defaults.

1.7. Recommendations for Secure Authentication System Implementation

Managing the complexity of modern authentication requires disciplined approaches to integration and validation:

- **Enforce all authentication state transitions on the server** - Never rely on client-side code for security decisions
- **Validate all IdP responses and tokens comprehensively** - Verify signatures, claims, issuer, audience, and expiration
- **Review all fallback and recovery flows for privilege escalation risks** - Test offline modes, password resets, and account recovery paths
- **Harden IdP defaults and minimize exposed endpoints** - Review password policies, rate limiting, and unused features
- **Implement server-side proxy patterns for IdP communication** - Avoid exposing IdP credentials and configuration to clients
- **Test authentication systems under realistic attacker conditions** - Include testing of MFA bypasses, token manipulation, and client-side enforcement failures
- **Document and regularly audit the complete authentication flow** - Map all components, trust boundaries, and failure modes

1.8. Conclusion

The evolution toward more secure and usable authentication systems—including multifactor authentication, third-party identity providers, and emerging technologies like passkeys—has created significant complexity that must be carefully managed. While technologies like passkeys offer real progress in eliminating phishing and credential reuse, their benefits are maximized only when integrated with secure backend validation, correct IdP configuration, and disciplined fallback design.

The case studies presented demonstrate that authentication failures rarely result from cryptographic weaknesses but instead emerge from integration complexity, configuration errors, and misunderstood trust boundaries. The future of authentication security will depend as much on implementation discipline and architectural understanding as on the underlying cryptographic strength of individual components.