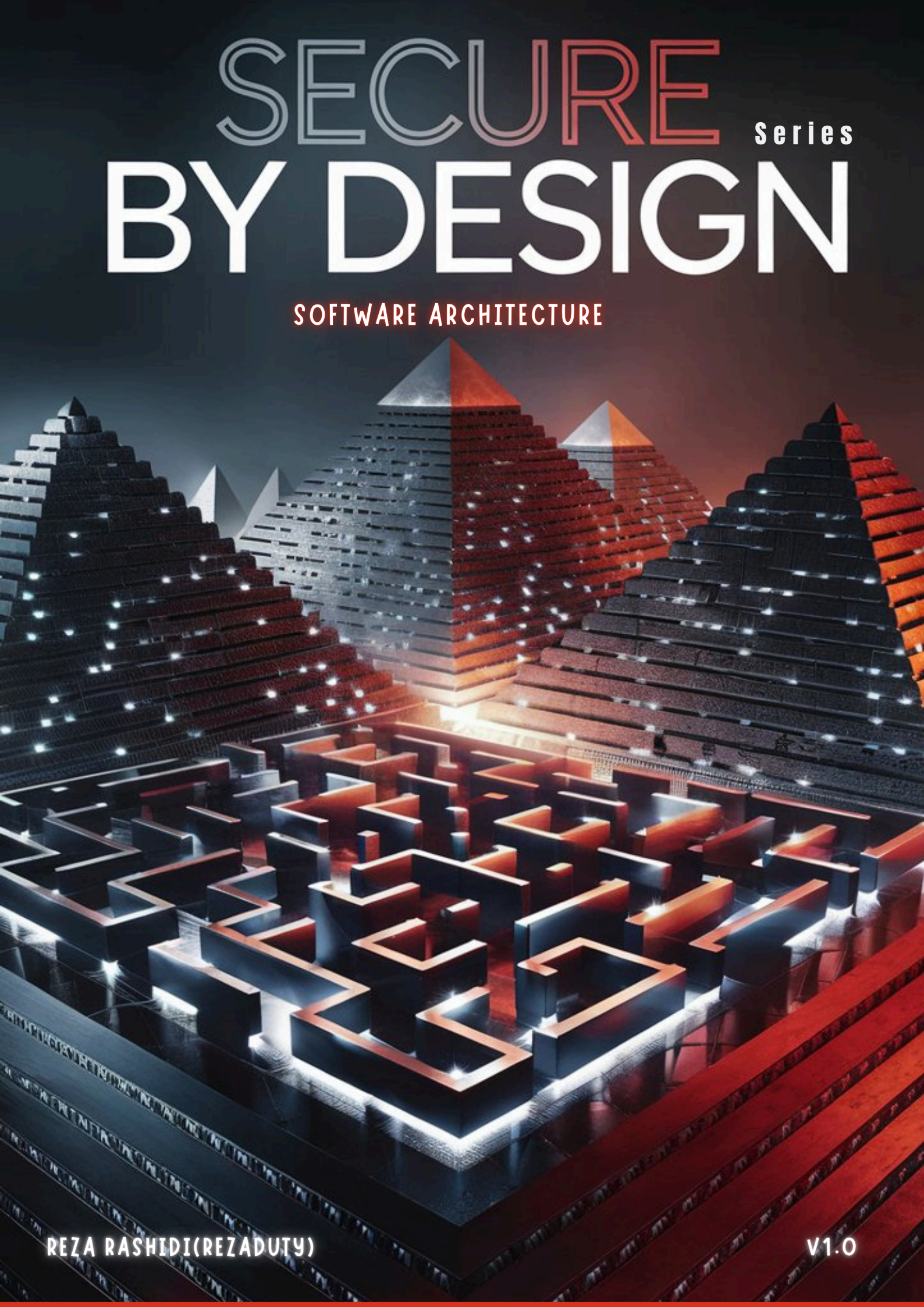
SECURE BY DESIGN

Series

SOFTWARE ARCHITECTURE



REZA RASHIDI(REZADUTY)

V1.0

FOREWORD

In the "Secure By Design" book series, we delve into the critical importance of embedding security into every layer of software architecture from the outset. This series meticulously contrasts insecure and secure design patterns across various architectural scenarios, providing readers with a comprehensive understanding of how to identify vulnerabilities and implement robust security measures. By exploring real-world examples and best practices, this series equips developers, architects, and security professionals with the knowledge and tools needed to build resilient, secure systems that can withstand modern cyber threats, ultimately fostering a culture of security-first thinking in software development.

Reza Rashidi

TABLE OF CONTENT

Software Architecture

In a bustling tech hub, a passionate team of developers discovered that building great software wasn't enough—they needed resilience against evolving cyber threats. As they embarked on a journey to overhaul their legacy systems, they embraced a "secure by design" mindset. Every line of code and architectural decision was scrutinized for vulnerabilities, ensuring that security was embedded from start to finish. Their efforts, driven by agile risk assessments and modern cryptographic practices, transformed their products into trusted pillars of innovation and safety.

Their success story quickly spread throughout the organization, inspiring stakeholders and executives alike. The team's rigorous approach—from proactive threat modeling to automated security testing—proved that integrating security throughout the software lifecycle was not just a necessity, but also a competitive advantage. This cultural shift not only bolstered customer trust but also set a new industry standard for secure, forward-thinking software architecture.

Authentication/Identity

Enterprise-Wide Passwordless Authentication Implementation

- Deploying **FIDO2/WebAuthn** for passwordless authentication.
- Implementing **biometric authentication** (Windows Hello, Face ID, fingerprint).
- Managing **legacy system authentication** with passwordless bridges.

Stage	Insecure Implementation	Secure Implementation (Best Practice)	Recommendations / Best Practices
User Initiation	User clicks "Login" on a standard UI; authentication prompt has no special security features.	User accesses a passwordless UI explicitly designed for FIDO2/WebAuthn; clear instructions for secure biometric input.	Use modern UIs that prompt users for secure authentication methods and educate about biometric verification best practices.
Challenge Generation	Auth Server returns a weak or static challenge (or none at all) that is easily replayed.	Auth Server generates a unique, dynamic cryptographic challenge following the FIDO2/WebAuthn protocol.	Ensure each login attempt uses a non-repeating, high-entropy challenge to prevent replay and forgery.
Biometric Input & Attestation	Client collects biometric input without secure hardware attestation.	Secure authenticator (e.g., Windows Hello) uses hardware-based key storage to sign the challenge, providing attestation.	Enforce secure hardware attestation so biometric credentials never leave the device in a form that can be intercepted or spoofed.
Token Issuance	Auth Server issues a weakly-signed or unsigned token based solely on the insecure challenge response.	Auth Server validates the signed response and issues a cryptographically secure, signed token (JWT/SAML), binding it to the challenge.	Tokens must be signed and, if needed, encrypted. Use proven cryptographic libraries and standards to avoid implementation flaws.
Legacy System Bridging	Legacy system integration simply passes the weak token along for access, without verification.	A dedicated secure token exchange gateway is used between modern authentication and legacy systems, validating tokens securely.	Use secure APIs, mutual TLS, and token mapping to integrate legacy systems with modern authentication without compromising integrity.
Transmission & Storage	Communications may occur over unencrypted channels; tokens stored without secure flags.	All transmissions occur over HTTPS/TLS; tokens stored only in HttpOnly, Secure cookies and encrypted storage where applicable.	Enforce TLS for all communications. Ensure cookies or tokens are flagged as HttpOnly and Secure, and apply encryption-at-rest for stored tokens.
Logging & Monitoring	Minimal logging with little context, hampering auditing and incident response.	Comprehensive logging captures user identity, file/token hashes, timestamps, and attestation verifications; integrated with SIEM.	Implement detailed logging and monitoring for forensic analysis and compliance, demonstrating measurable program maturity to stakeholders and auditors.

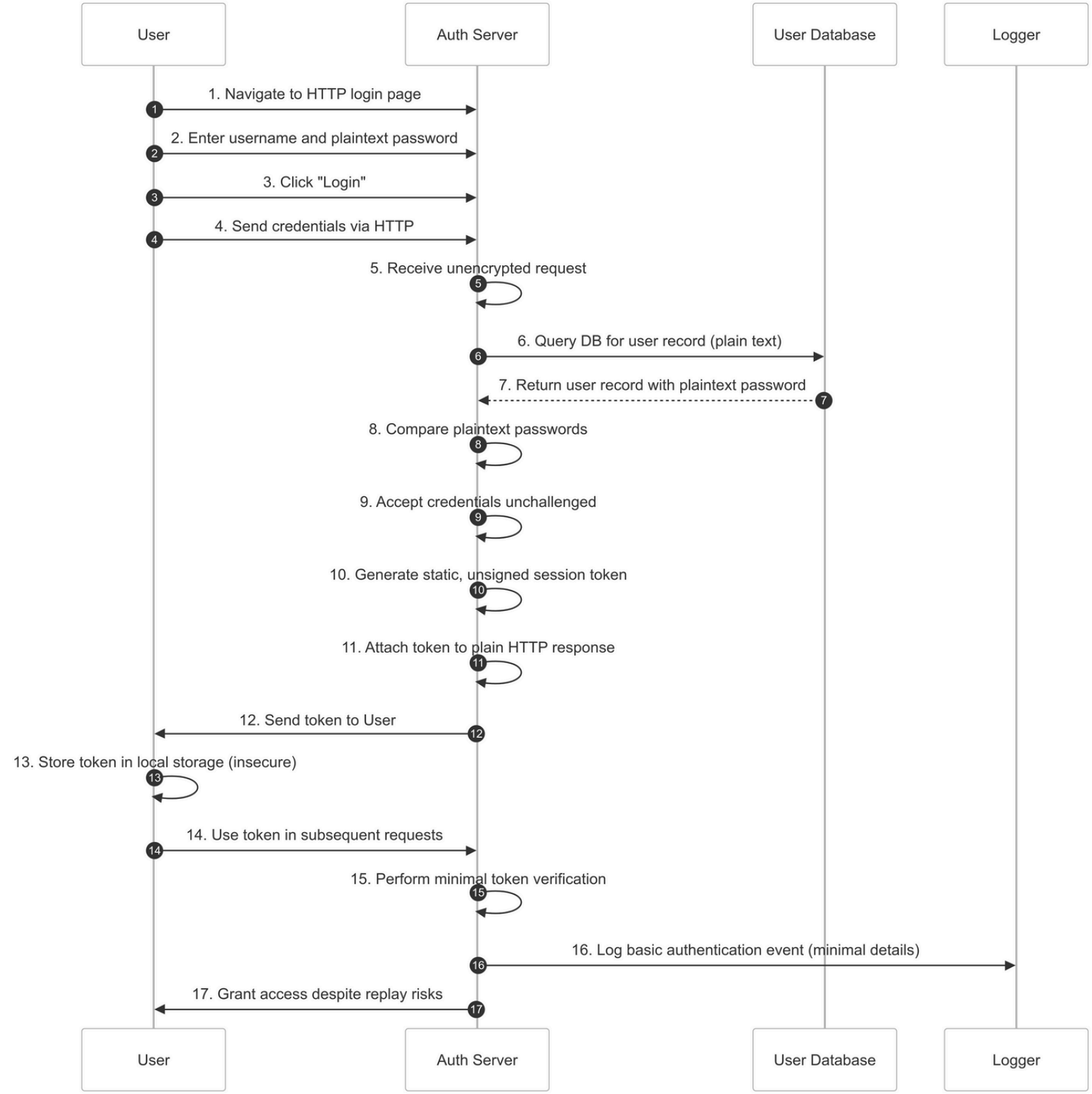
Authentication/Identity

Enterprise-Wide Passwordless Authentication Implementation

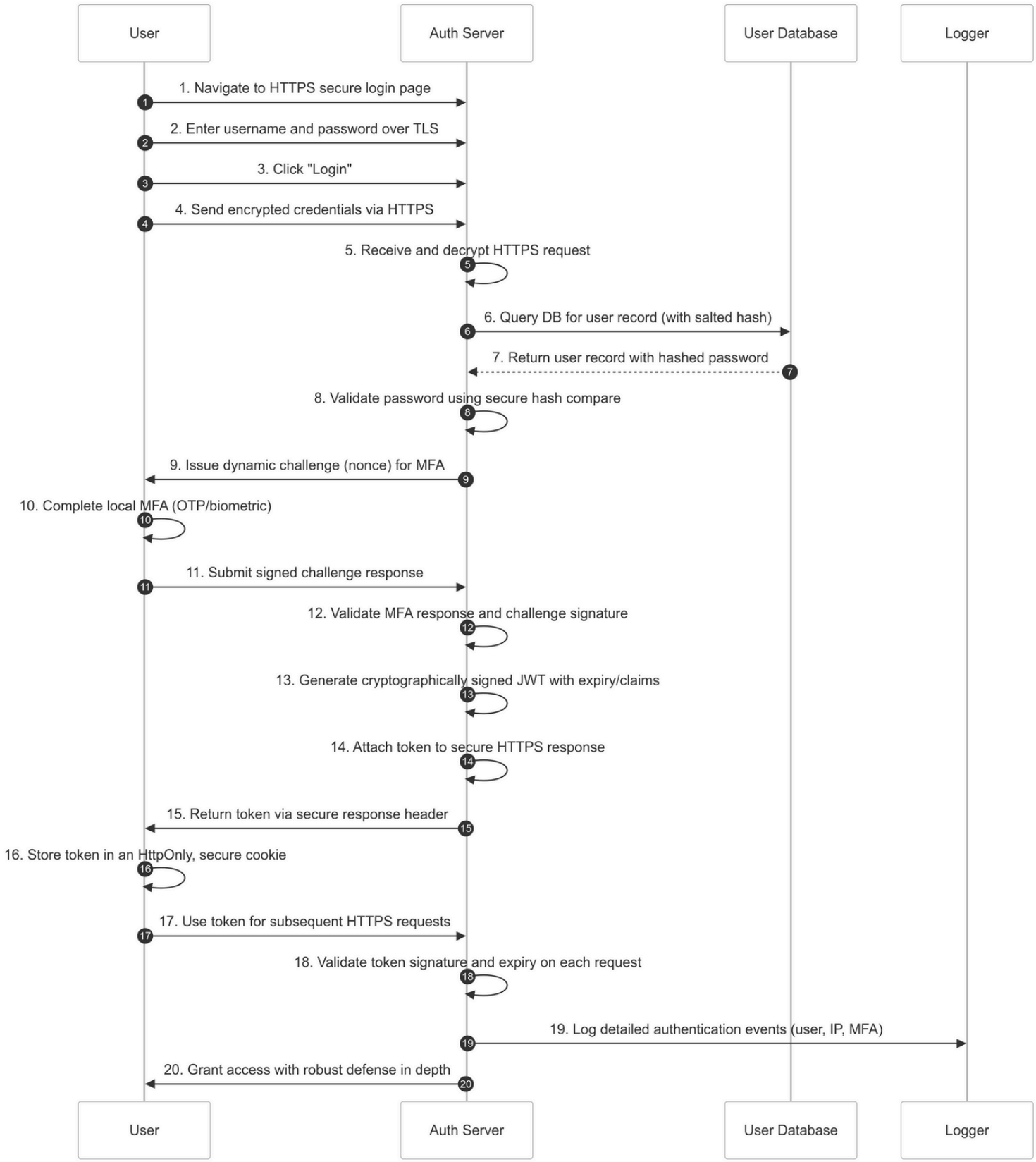
- Deploying **FIDO2/WebAuthn** for passwordless authentication.
- Implementing **biometric authentication** (Windows Hello, Face ID, fingerprint).
- Managing **legacy system authentication** with passwordless bridges.

Stage	Insecure Implementation	Secure Implementation (Best Practice)	Recommendations / Best Practices
User Initiation	User clicks "Login" on a standard UI; authentication prompt has no special security features.	User accesses a passwordless UI explicitly designed for FIDO2/WebAuthn; clear instructions for secure biometric input.	Use modern UIs that prompt users for secure authentication methods and educate about biometric verification best practices.
Challenge Generation	Auth Server returns a weak or static challenge (or none at all) that is easily replayed.	Auth Server generates a unique, dynamic cryptographic challenge following the FIDO2/WebAuthn protocol.	Ensure each login attempt uses a non-repeating, high-entropy challenge to prevent replay and forgery.
Biometric Input & Attestation	Client collects biometric input without secure hardware attestation.	Secure authenticator (e.g., Windows Hello) uses hardware-based key storage to sign the challenge, providing attestation.	Enforce secure hardware attestation so biometric credentials never leave the device in a form that can be intercepted or spoofed.
Token Issuance	Auth Server issues a weakly-signed or unsigned token based solely on the insecure challenge response.	Auth Server validates the signed response and issues a cryptographically secure, signed token (JWT/SAML), binding it to the challenge.	Tokens must be signed and, if needed, encrypted. Use proven cryptographic libraries and standards to avoid implementation flaws.
Legacy System Bridging	Legacy system integration simply passes the weak token along for access, without verification.	A dedicated secure token exchange gateway is used between modern authentication and legacy systems, validating tokens securely.	Use secure APIs, mutual TLS, and token mapping to integrate legacy systems with modern authentication without compromising integrity.
Transmission & Storage	Communications may occur over unencrypted channels; tokens stored without secure flags.	All transmissions occur over HTTPS/TLS; tokens stored only in HttpOnly, Secure cookies and encrypted storage where applicable.	Enforce TLS for all communications. Ensure cookies or tokens are flagged as HttpOnly and Secure, and apply encryption-at-rest for stored tokens.
Logging & Monitoring	Minimal logging with little context, hampering auditing and incident response.	Comprehensive logging captures user identity, file/token hashes, timestamps, and attestation verifications; integrated with SIEM.	Implement detailed logging and monitoring for forensic analysis and compliance, demonstrating measurable program maturity to stakeholders and auditors.

Insecure


User
Auth Server
User Database
Logger
1. Navigate to HTTP login page
2. Enter username and plaintext password
3. Click "Login"
4. Send credentials via HTTP
5. Receive unencrypted request
6. Query DB for user record (plain text)
7. Return user record with plaintext password
8. Compare plaintext passwords
9. Accept credentials unchallenged
10. Generate static, unsigned session token
11. Attach token to plain HTTP response
12. Send token to User
13. Store token in local storage (insecure)
14. Use token in subsequent requests
15. Perform minimal token verification
16. Log basic authentication event (minimal details)
17. Grant access despite replay risks
User
Auth Server
User Database
Logger

Secure


User
Auth Server
User Database
Logger
1. Navigate to HTTPS secure login page
2. Enter username and password over TLS
3. Click "Login"
4. Send encrypted credentials via HTTPS
5. Receive and decrypt HTTPS request
6. Query DB for user record (with salted hash)
7. Return user record with hashed password
8. Validate password using secure hash compare
9. Issue dynamic challenge (nonce) for MFA
10. Complete local MFA (OTP/biometric)
11. Submit signed challenge response
12. Validate MFA response and challenge signature
13. Generate cryptographically signed JWT with expiry/claims
14. Attach token to secure HTTPS response
15. Return token via secure response header
16. Store token in an HttpOnly, secure cookie
17. Use token for subsequent HTTPS requests
18. Validate token signature and expiry on each request
19. Log detailed authentication events (user, IP, MFA)
20. Grant access with robust defense in depth
User
Auth Server
User Database
Logger

Stage	Insecure Approach	Secure Approach	Remediation / Best Practice
1. Login Request Submission	HTTP login page; plaintext credentials submitted unencrypted.	HTTPS secure login page; credentials sent over encrypted channels.	Always use HTTPS/TLS to encrypt credentials during transmission.
2. Credential Verification	Direct query to DB returning plaintext password; simple password comparison with no challenge.	Query DB for salted-hashed record; perform secure hash comparison; issue dynamic challenge (nonce) for additional verification.	Use salted hashes for passwords and implement dynamic challenges to mitigate replay and injection attacks.
3. Token Generation / MFA	Generate static, unsigned token; no MFA invoked.	After MFA challenge, generate cryptographically signed JWT with expiry and claims.	Integrate multi-factor authentication and issue signed tokens (e.g., JWT) for session management.
4. Session Establishment & Logging	Token is stored in insecure local storage; minimal logging, no robust token verification.	Token is stored in HttpOnly, secure cookie; extensive logging (user, IP, MFA event) and continuous token verification.	Enforce secure storage of tokens and implement comprehensive logging and monitoring for incident response and regulatory compliance.

Multi-Factor Authentication (MFA) & Phishing-Resistant Methods

- Implementing **smartcards, hardware tokens (YubiKey), and passkeys.**
- **MFA fatigue protection** (number matching, risk-based MFA).
- Secure fallback for **MFA recovery scenarios.**

Diagram A: Insecure Authentication/SSO Workflow

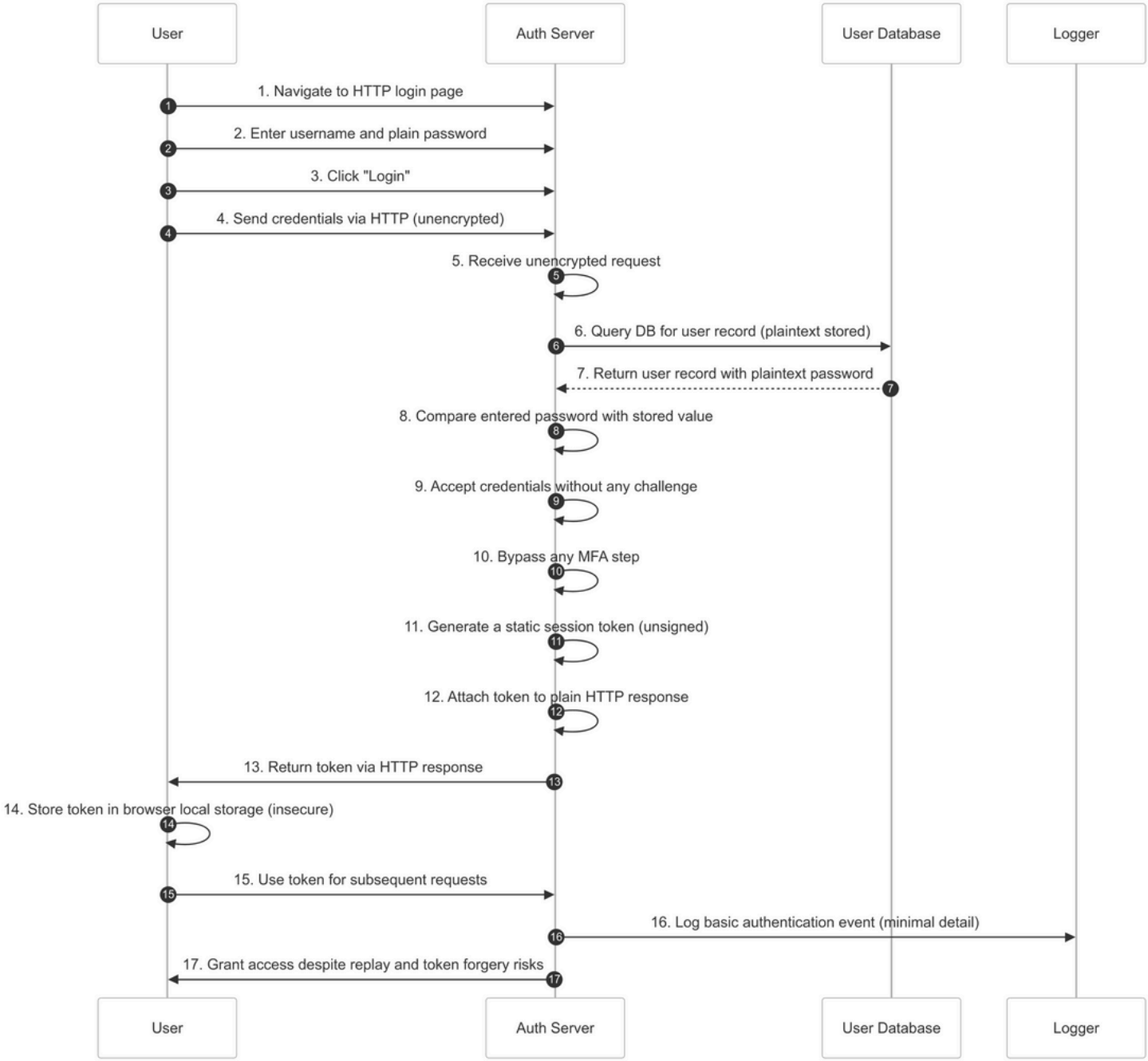
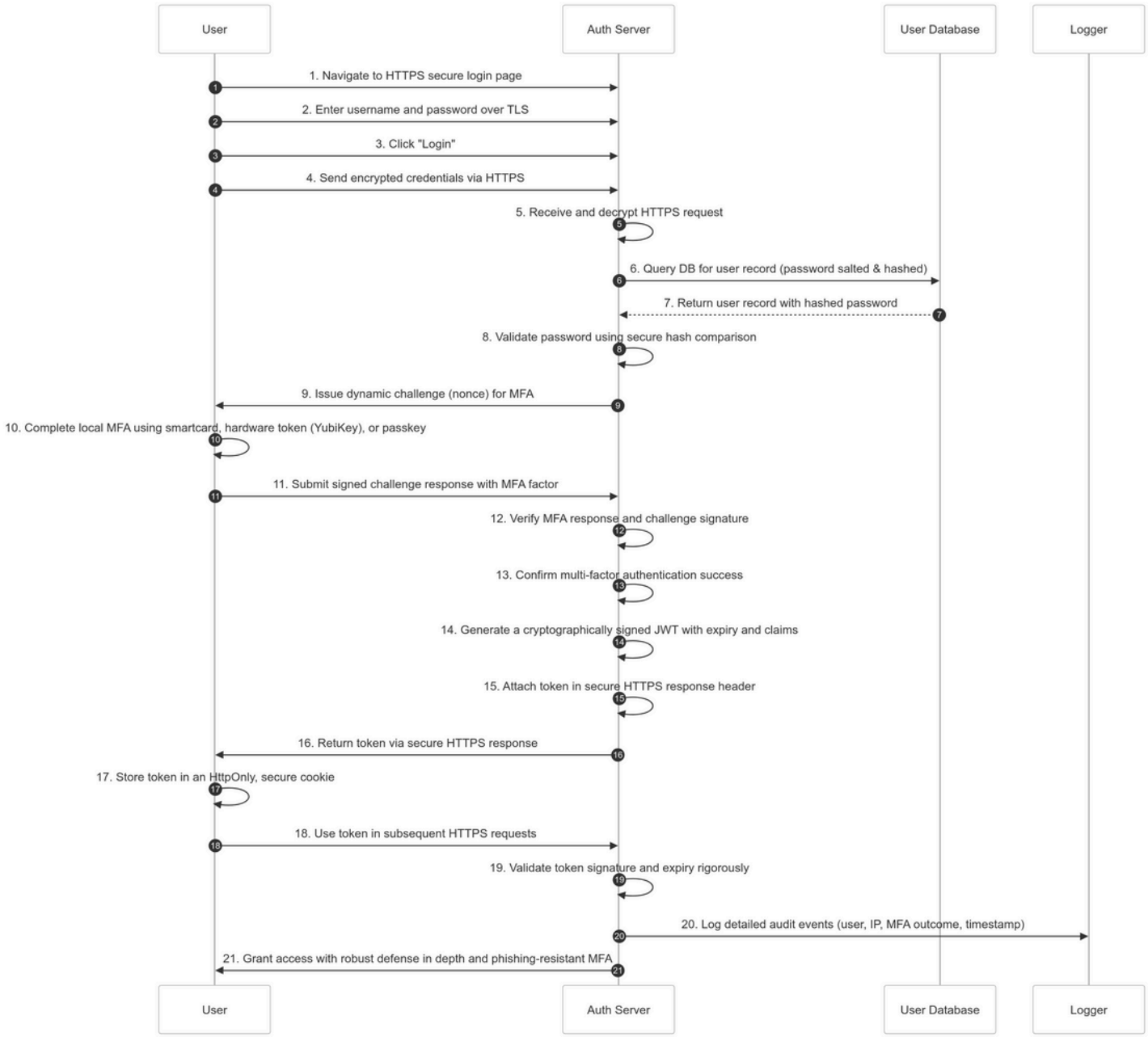


Diagram B: Secure Authentication/SSO Workflow (Defense in Depth + MFA)



Cheatsheet Table: Insecure vs. Secure Workflow per Stage

Stage	Insecure Implementation	Secure Implementation	Best Practice / Remediation
1. Login Request Submission	HTTP login page; plaintext credentials sent unencrypted.	HTTPS secure login page; credentials transmitted over TLS.	Use HTTPS/TLS to secure data in transit.
2. Credential Verification	Queries DB returning plaintext passwords; simple, unsalted comparison; no dynamic challenge issued.	Queries DB for salted-hashed password; secure hash comparison; issues dynamic challenge (nonce) to avoid replay attacks.	Employ salted hashes; enforce dynamic challenges during authentication.
3. Multi-Factor Authentication	No MFA applied; static, unsigned token generated without additional validation.	Enforces MFA using smartcards, hardware tokens (YubiKey), or passkeys; verifies signed challenge response to prevent phishing.	Integrate robust, phishing-resistant MFA methods to supplement password-based authentication.
4. Token Issuance, Storage & Logging	Token is attached in plain HTTP response; stored insecurely in local storage; minimal logging.	Generates cryptographically signed JWT with expiry & claims; token sent via secure HTTPS, stored in HttpOnly cookies; comprehensive logging.	Use signed tokens, secure cookie storage, and detailed logging to ensure session integrity and regulatory compliance.

Single Sign-On (SSO) with Identity Federation (SAML, OAuth, OIDC, Kerberos)

- Federating identities across **cloud (Azure AD, Okta) and on-prem (Active Directory, LDAP)**.
- Implementing **Just-in-Time (JIT) provisioning** for dynamic user creation.
- Managing **cross-domain authentication** in **multi-cloud environments**.

Diagram A: Insecure SSO / Identity Federation Workflow

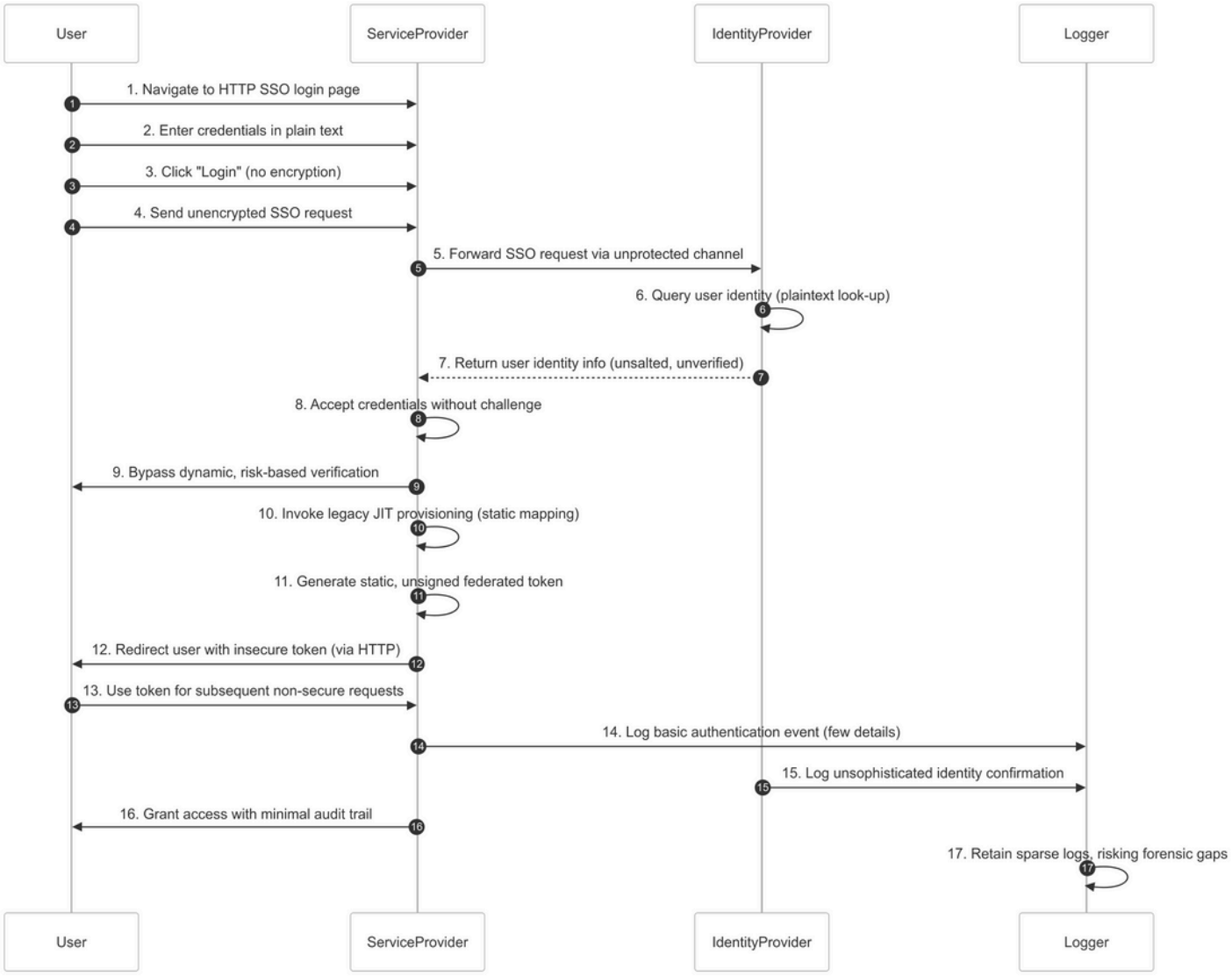
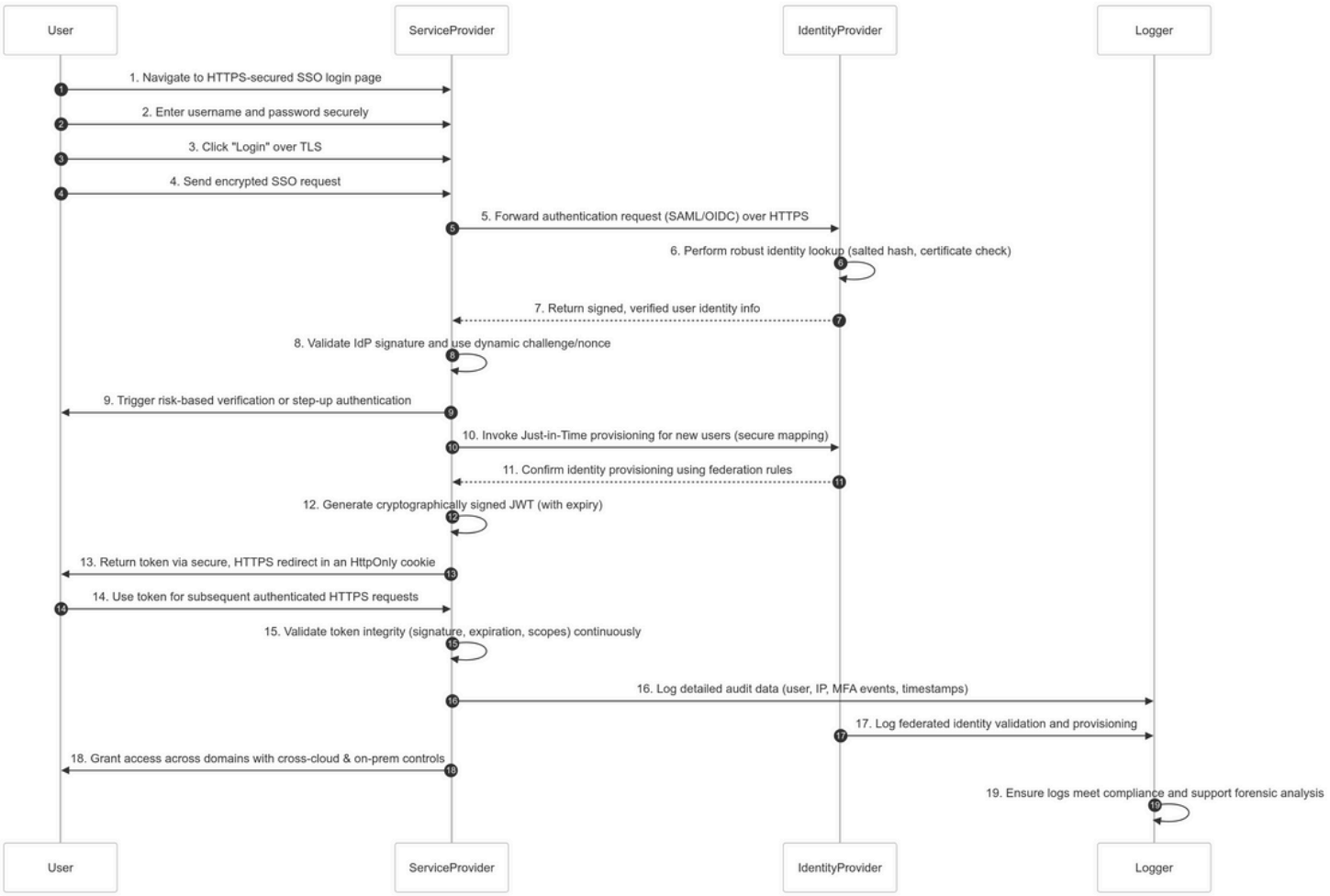


Diagram B: Secure SSO / Identity Federation Workflow (Defense in Depth)



Cheatsheet Table: Comparison per Stage

Stage	Insecure Implementation	Secure Implementation	Best Practice / Remediation
1. Login Request Submission	Uses HTTP; credentials sent in plaintext; no encryption or risk-based controls.	HTTPS-secured login page with TLS; credentials transmitted securely over encryption; risk-based or adaptive authentication enabled.	Always enforce HTTPS/TLS; use secure forms with adaptive authentication to mitigate credential interception.
2. Federation & Identity Retrieval	Forwards SSO request over unprotected channel; identity lookup returns unsalted, plaintext data; no signature verification.	Forwards authentication via SAML/OIDC; identity provider validates user using salted hashes and certificates; returns signed identity info.	Ensure secure federation protocols (SAML, OAuth, OIDC); implement certificate-based signing and dynamic nonce challenges to prevent replay and injection attacks.
3. Token Generation & JIT Provisioning	Legacy JIT without dynamic checks; static, unsigned token issuance; insecure token passed via HTTP.	Implements JIT provisioning with dynamic checks; generates cryptographically signed JWT with expiry; token delivered securely (HttpOnly cookie).	Use dynamic provisioning; issue signed tokens with expiration and claims; secure tokens with HttpOnly and secure flags; ensure tokens follow least privilege access controls.
4. Logging, Cross-Domain Control & Monitoring	Performs minimal logging with sparse details; no robust cross-domain controls; forensic gaps present.	Logs detailed audit data (user, IP, MFA events, timestamps); cross-domain access controlled for multi-cloud and on-prem configurations; compliant logging.	Use detailed, structured logging integrated with SIEM; enforce cross-domain access with stringent controls; ensure logs meet regulatory standards and provide full forensic traceability.

OAuth 2.0 & OpenID Connect (OIDC) for Secure API Authentication

- Enforcing **client authentication with PKCE and mutual TLS (mTLS)**.
- Using **JWT access tokens** securely (signature validation, expiration, scopes).
- Protecting **OAuth refresh tokens** to prevent token theft attacks.

Diagram A: Insecure OAuth 2.0/OIDC Workflow

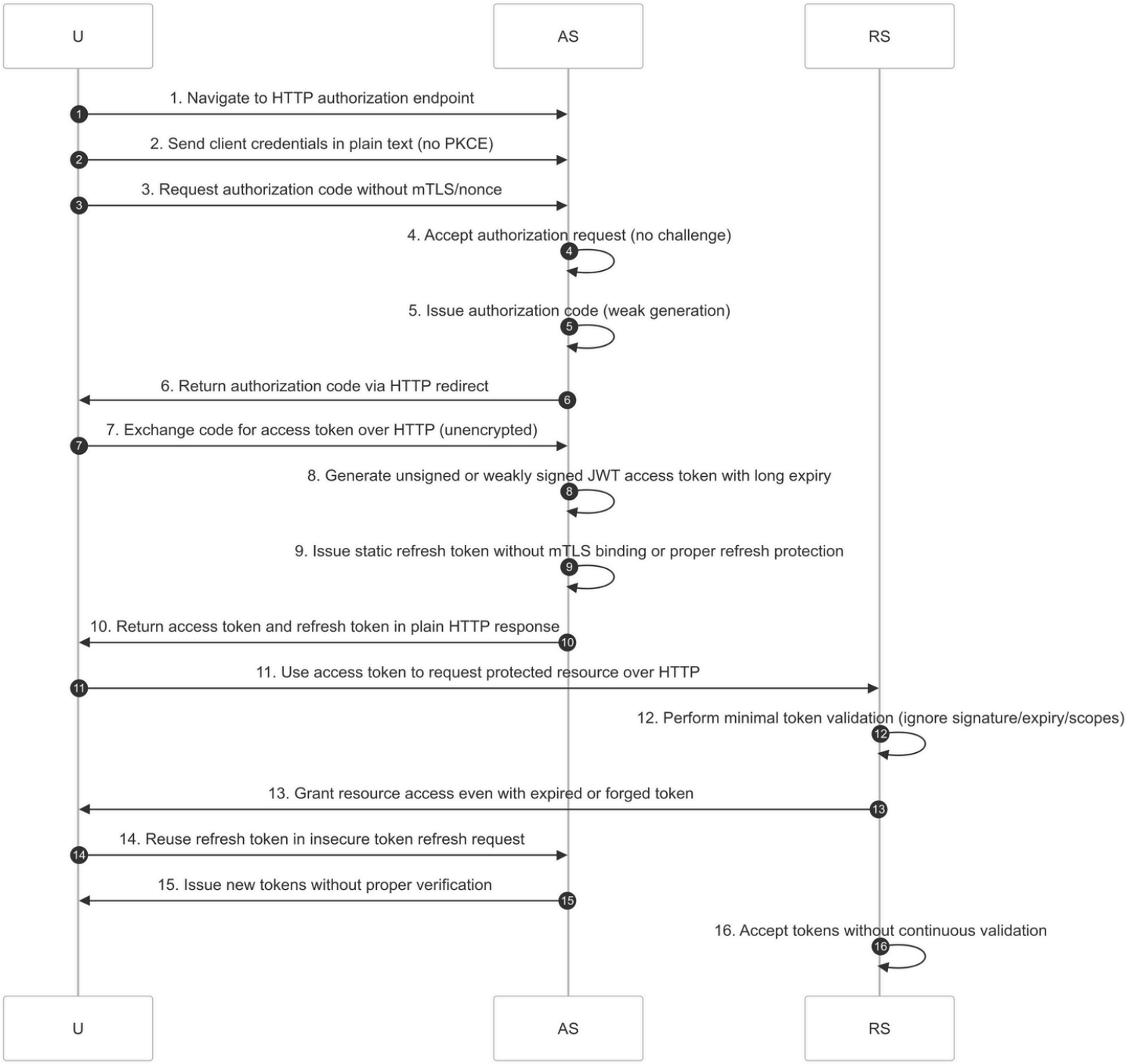
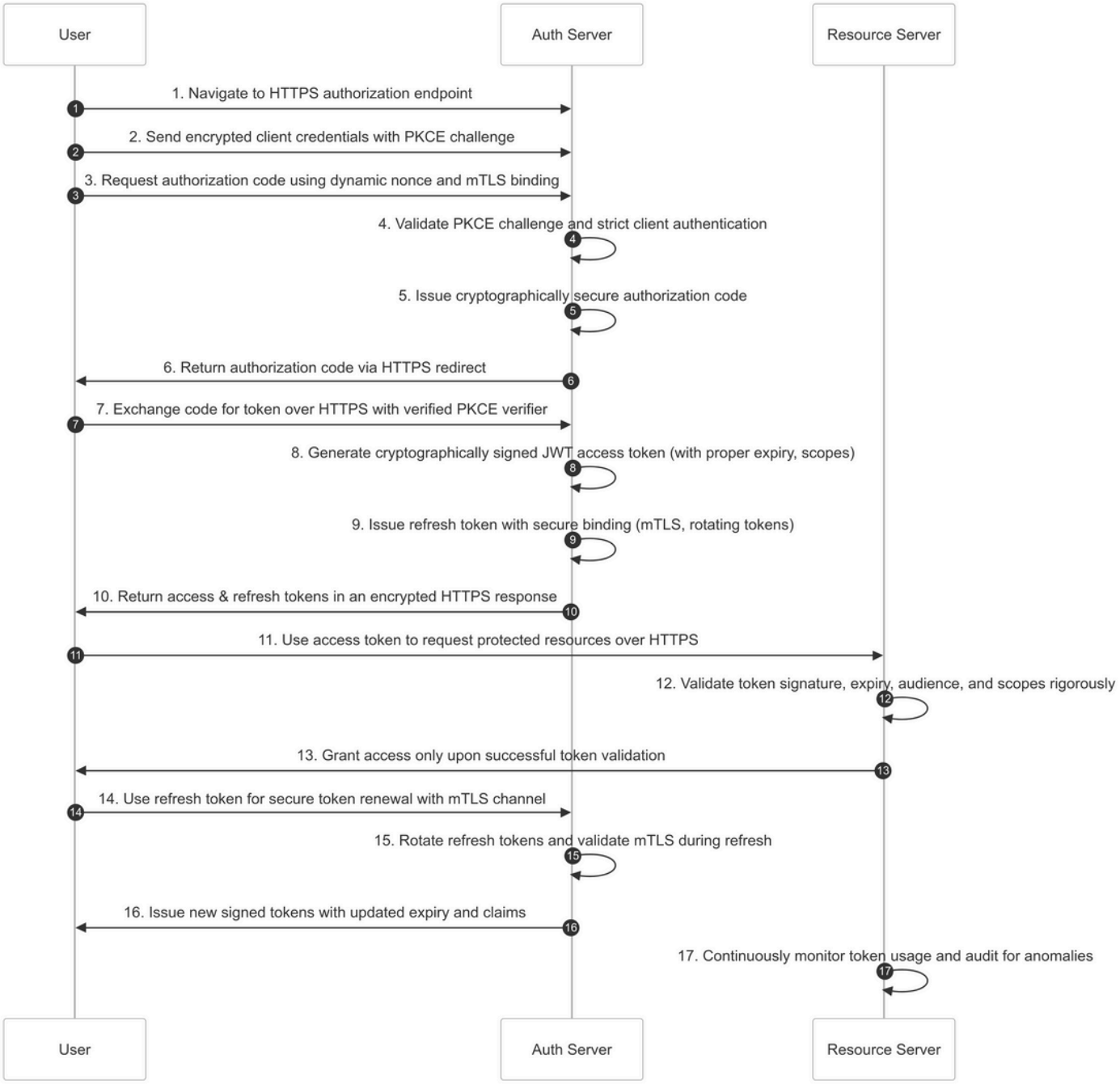


Diagram B: Secure OAuth 2.0/OIDC Workflow (Defense in Depth)



Cheatsheet Table: Insecure vs. Secure Workflow per Stage

Stage	Insecure Implementation	Secure Implementation	Best Practice / Remediation
1. Token Request	- HTTP endpoint (unencrypted) - Client credentials sent in plain text - No PKCE or nonce used	- HTTPS endpoint; fully encrypted - Credentials sent with PKCE challenge - Dynamic nonce and mTLS binding for client authentication	Use TLS encryption; enforce PKCE and dynamic nonces; implement mTLS to prevent interception and replay attacks.
2. Token Issuance & Refresh	- Authorization code and tokens generated with weak/random methods - Unsigned or weakly signed JWT - Static refresh token issued without binding	- Authorization code issued securely - Cryptographically signed JWT access token with explicit expiry, scopes, and audience - Refresh tokens securely bound (rotated via mTLS)	Implement cryptographic token signing; enforce expiration and token scopes; secure refresh tokens using binding (mTLS) and rotation.
3. API Access & Session Handling	- Access tokens used over HTTP - Minimal token validation (signature/expiry not checked) - No continuous monitoring	- Access tokens transmitted via HTTPS - Rigorous token validation (signature, expiry, audience, and scopes) - Continuous monitoring and secure token renewal via mTLS	Always transmit tokens over secure channels; validate tokens thoroughly on every request; use continuous monitoring and refresh controls.

Web Service and API

JWT Lifecycle Management and Signature Validation

- **Architect:** Deploy robust JWT mechanisms—create tokens with embedded claims, short expiration, and cryptographic signatures, with regular token reissuance and revocation.
- **Attacks/TTPs:** Token hijacking, forgery, replay attacks.
- **Standards & Best Practices:** NIST SP 800-63, JWT Best Practices, automated vulnerability scanning.
- **Roles and Users:** Application developers, security analysts, operations teams; end-user sessions across web/mobile apps.

Diagram A – Insecure JWT Lifecycle Management

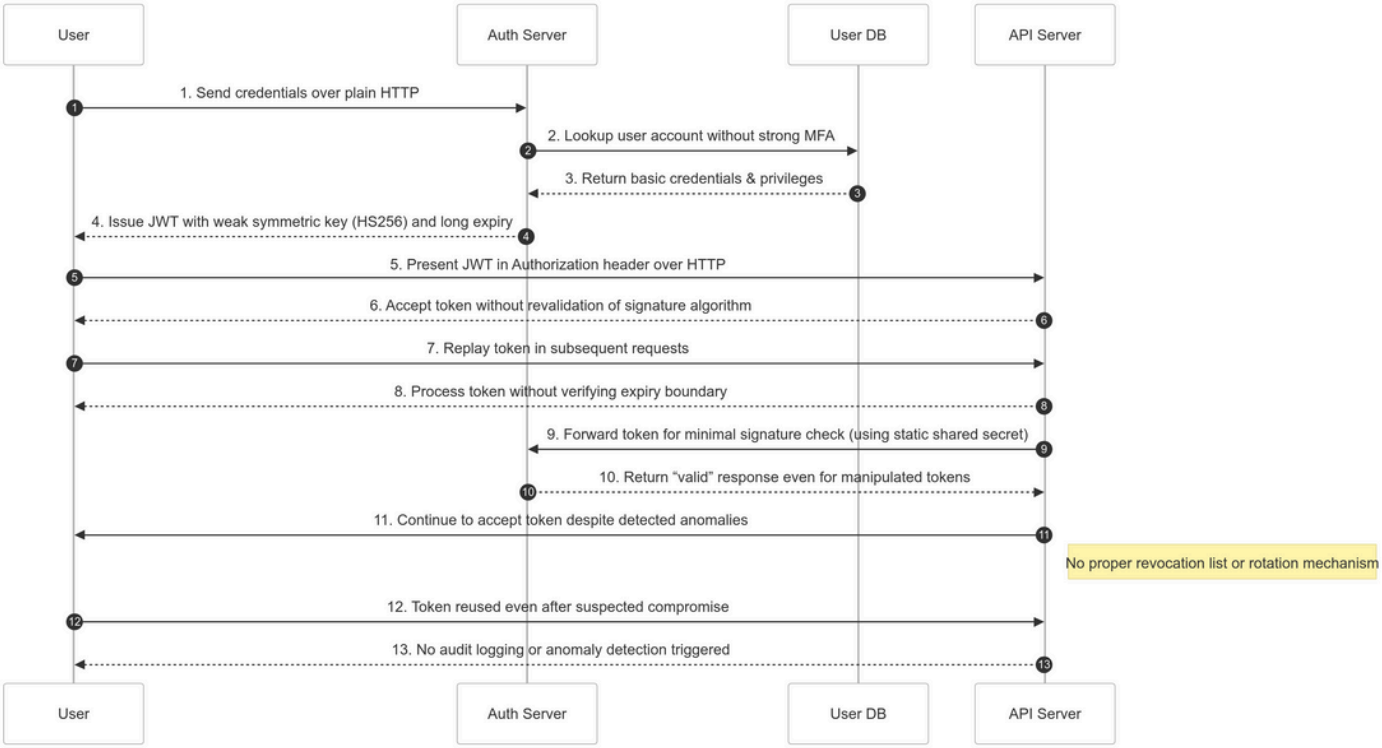
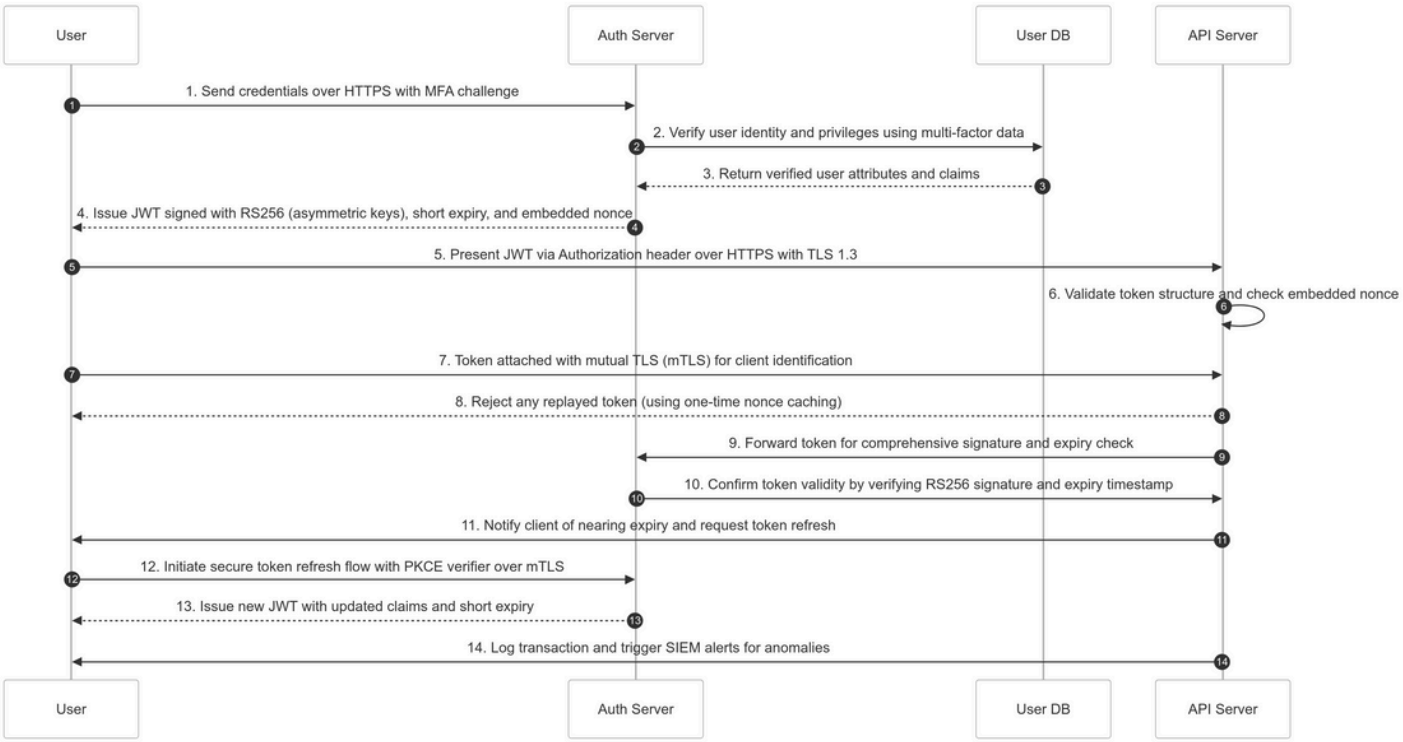


Diagram B – Secure JWT Lifecycle Management



Cheatsheet Table: Comparison Per Stage (Insecure vs. Secure)

Stage	Insecure Implementation	Secure Implementation	Best Practice / Remediation
Token Issuance	• Credentials sent over plain HTTP • No MFA; weak user checks • Uses HS256 with static secret • Long-lived tokens, no nonce	• Credentials exchanged over HTTPS with MFA • User identity verified after multi-factor checks • Uses RS256 with asymmetric keys • Short token lifetime with embedded nonce	Use TLS, enforce MFA and additional identity checks; use asymmetric signing (RS256); enforce short expiry and nonces
Token Propagation	• Tokens sent in clear text over HTTP • No replay prevention or binding to TLS • No mTLS; vulnerable to interception • No transmission integrity checks	• JWT transmitted over HTTPS with TLS 1.3 • mTLS to bind client identity • Replay prevention using one-time nonce caching • Integrity guaranteed with secure transport protocols	Use TLS and mTLS to secure token flow; incorporate nonce-based replay prevention; enforce transmission integrity
Token Validation	• Minimal signature check using weak static secret • No thorough expiry verification or claim checking • Vulnerable to token manipulation	• Comprehensive signature verification with RS256 • Checks expiry timestamp and nonce • Token validated against current claims and client identity	Validate every claim, signature, and expiry; use strong cryptographic verification; use centralized validation services
Revocation/Rotation	• No token revocation mechanism • Tokens reused after compromise • No audit logging • No automated rotation	• Automated token refresh flow with PKCE verifier • Revocation & issuance follows secure rotation processes • Integrated logging with SIEM and audit trails • Alerts set on anomaly detection	Implement secure refresh and revocation flows; automate token rotation; integrate logging and alerting mechanisms for rapid anomaly response

Dynamic API Gateway with Integrated WAF

• **Architect:** Utilize an API gateway that integrates a Web Application Firewall (WAF) for real-time traffic inspection, anomaly detection, and responsive policy enforcement.
• **Attacks/TTPs:** OWASP Top 10 vulnerabilities, API-specific exploits, botnet attacks.
• **Standards & Best Practices:** OWASP API Security Top 10, NIST SP 800-95, continuous security monitoring.
• **Roles and Users:** API gateway administrators, network security, security operations center (SOC); internal and external user access.

Diagram A – Insecure Implementation

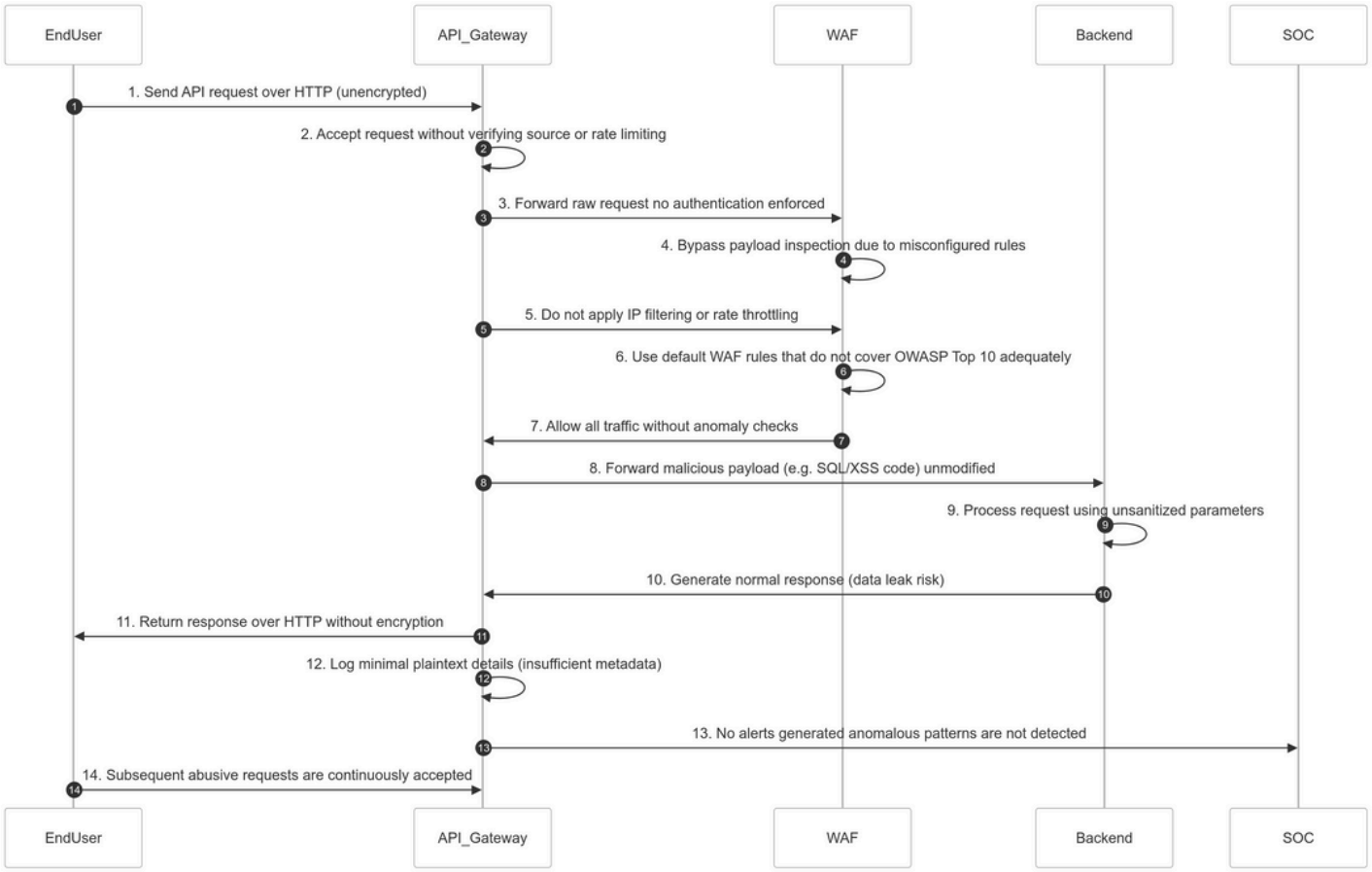
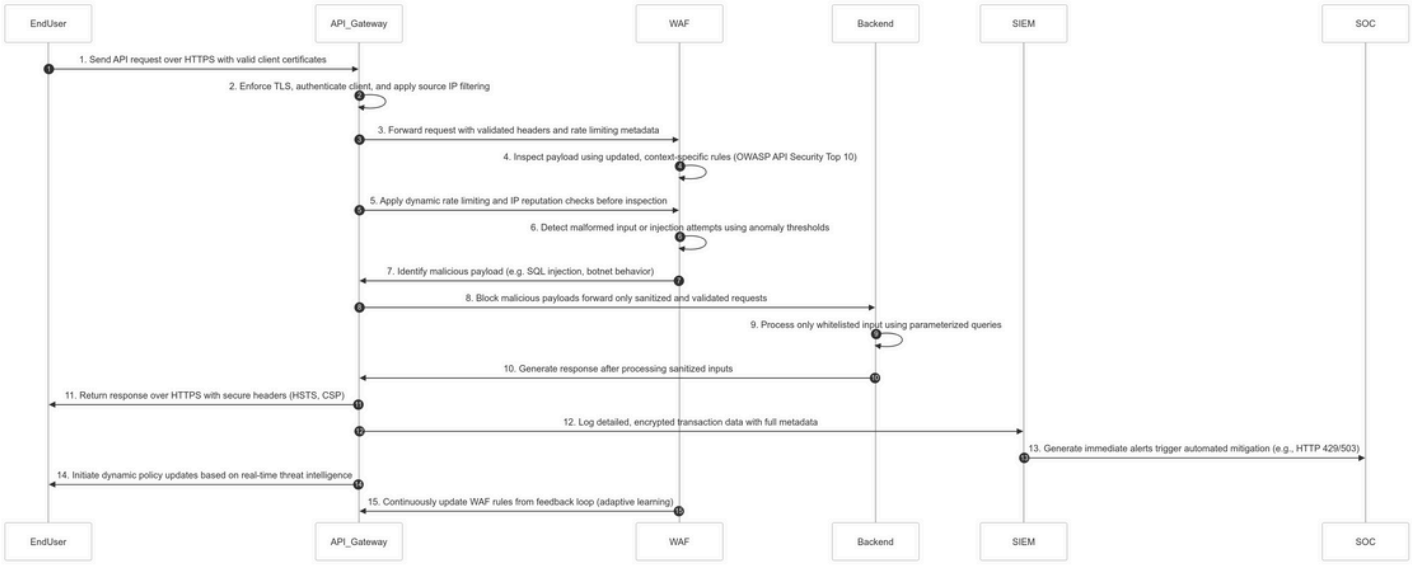


Diagram B – Secure Implementation

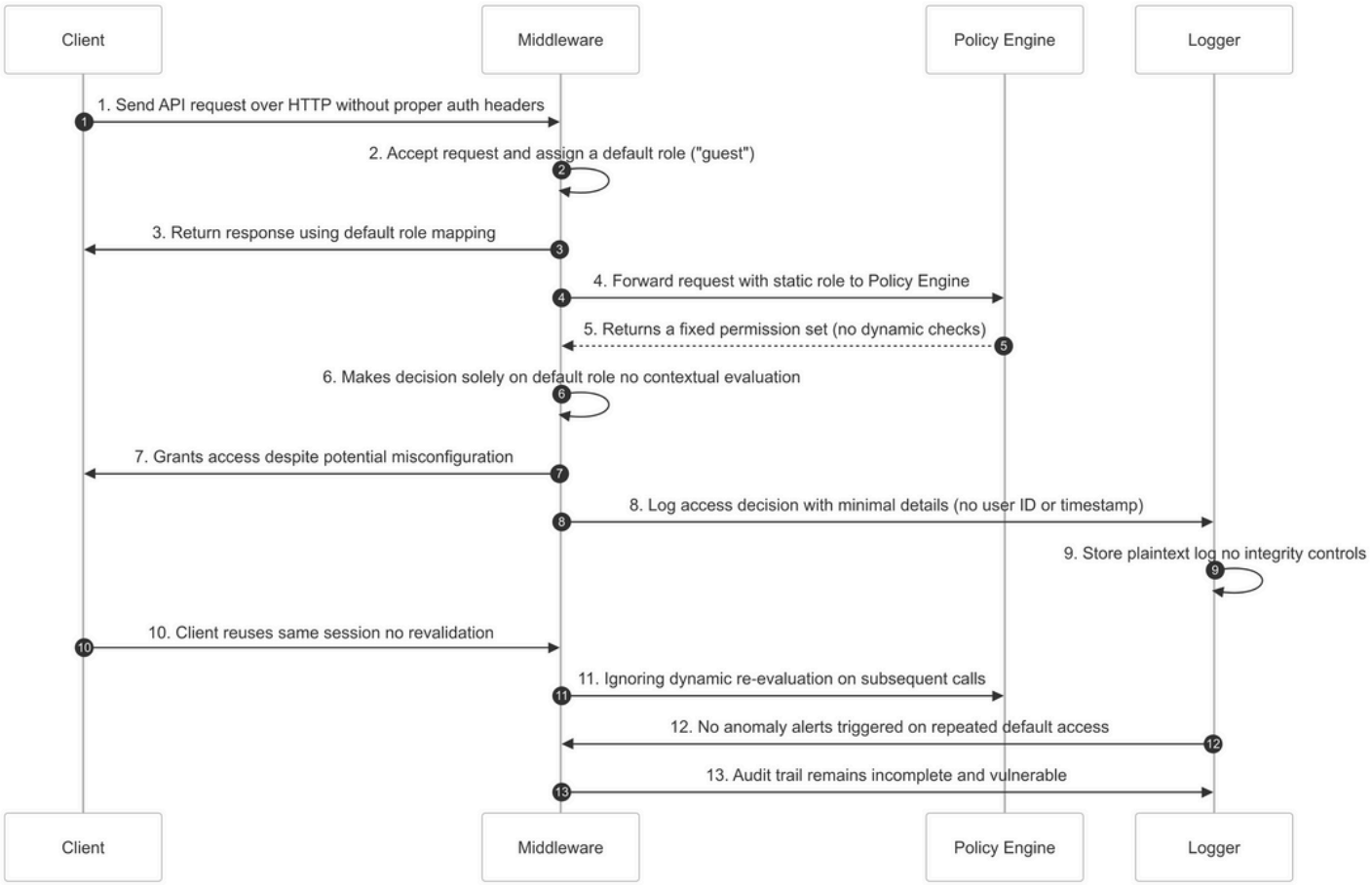


Secure Access Control/Middleware

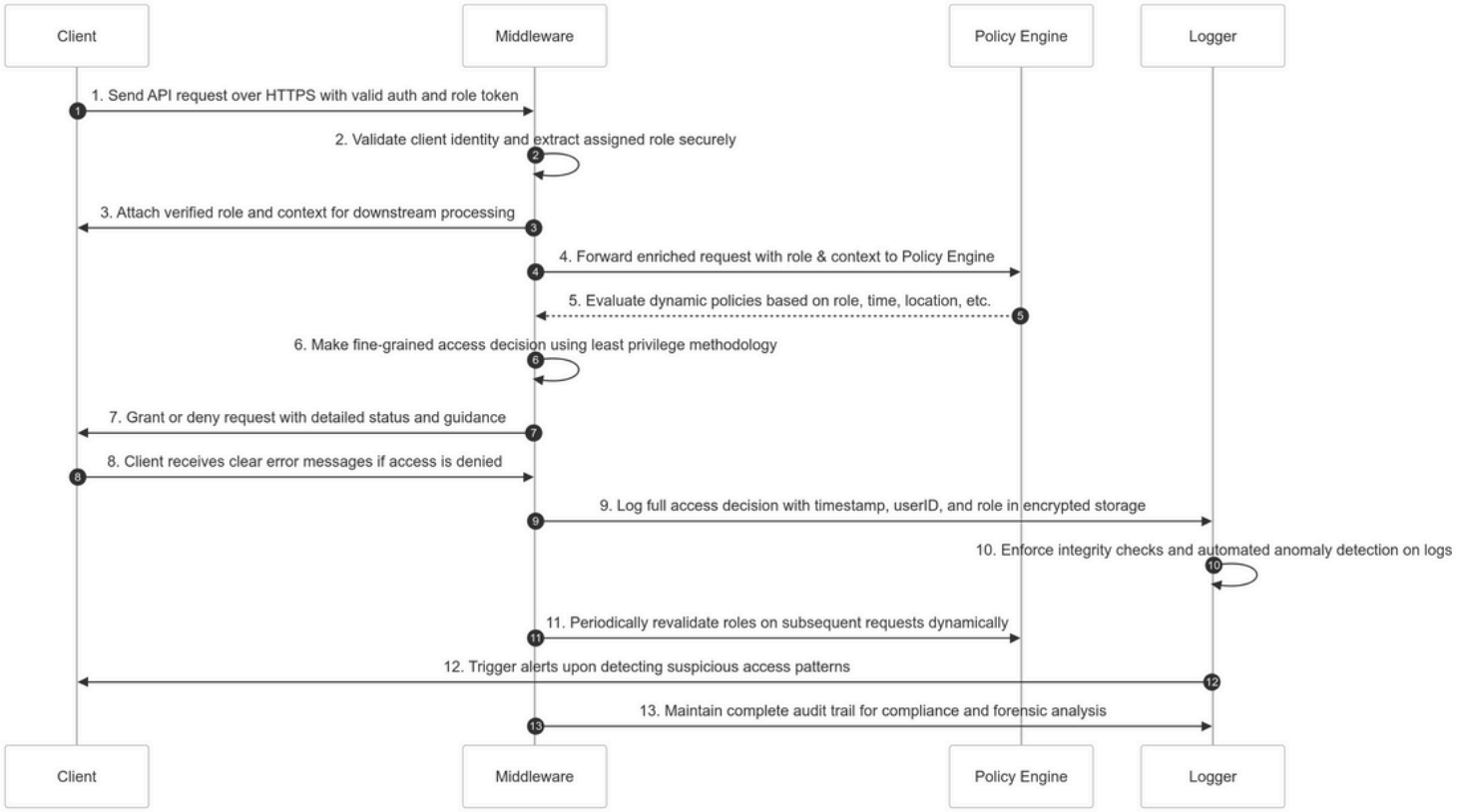
Role-Based Access Control (RBAC) Middleware

• **Architect:** Implement middleware that enforces RBAC across all incoming API and service requests. The solution dynamically maps users' roles to permissions at runtime and logs every access decision through centralized policy engines.
• **Attacks/TTPs:** Privilege escalation, unauthorized access via role misconfiguration, insider threats.
• **Standards for Security Testing and Best Practices:** NIST SP 800-162, OWASP Access Control, periodic access reviews and penetration testing of the middleware.
• **Roles and Users:** IAM specialists, security architects, administrators; end-users, API consumers.

Insecure Implementation – RBAC Middleware



Secure Implementation – RBAC Middleware

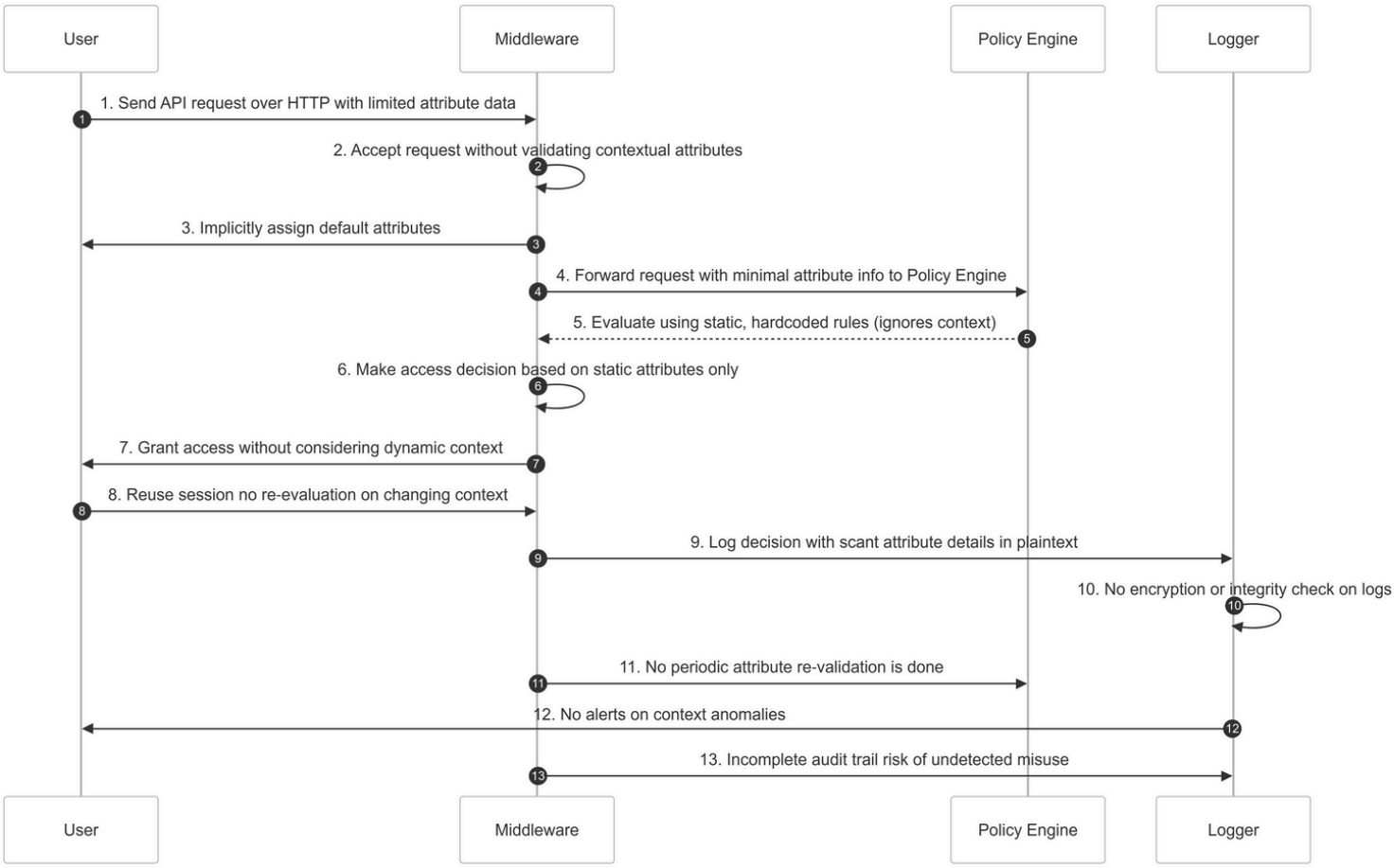


Stage	Insecure	Secure	Best Practice
Request Reception	• Unencrypted request • Default role assignment without auth	• HTTPS with valid auth • Role extracted from validated token	Use TLS; enforce proper authentication and role extraction
Policy Evaluation	• Static permission set • No dynamic or contextual checks	• Dynamically evaluated policies using context (time, location, etc.)	Evaluate policies dynamically based on context and least privilege
Access Decision	• Decision solely based on defaults • No user-specific adjustments	• Fine-grained decision using dynamic input • Clear feedback on access decisions	Enforce least privilege and use clear role-based decision logic
Logging & Audit	• Minimal plaintext logging • No integrity verification	• Encrypted, detailed logging with automated alerts and complete audit trails	Log all access decisions securely; integrate integrity checks and real-time alerting

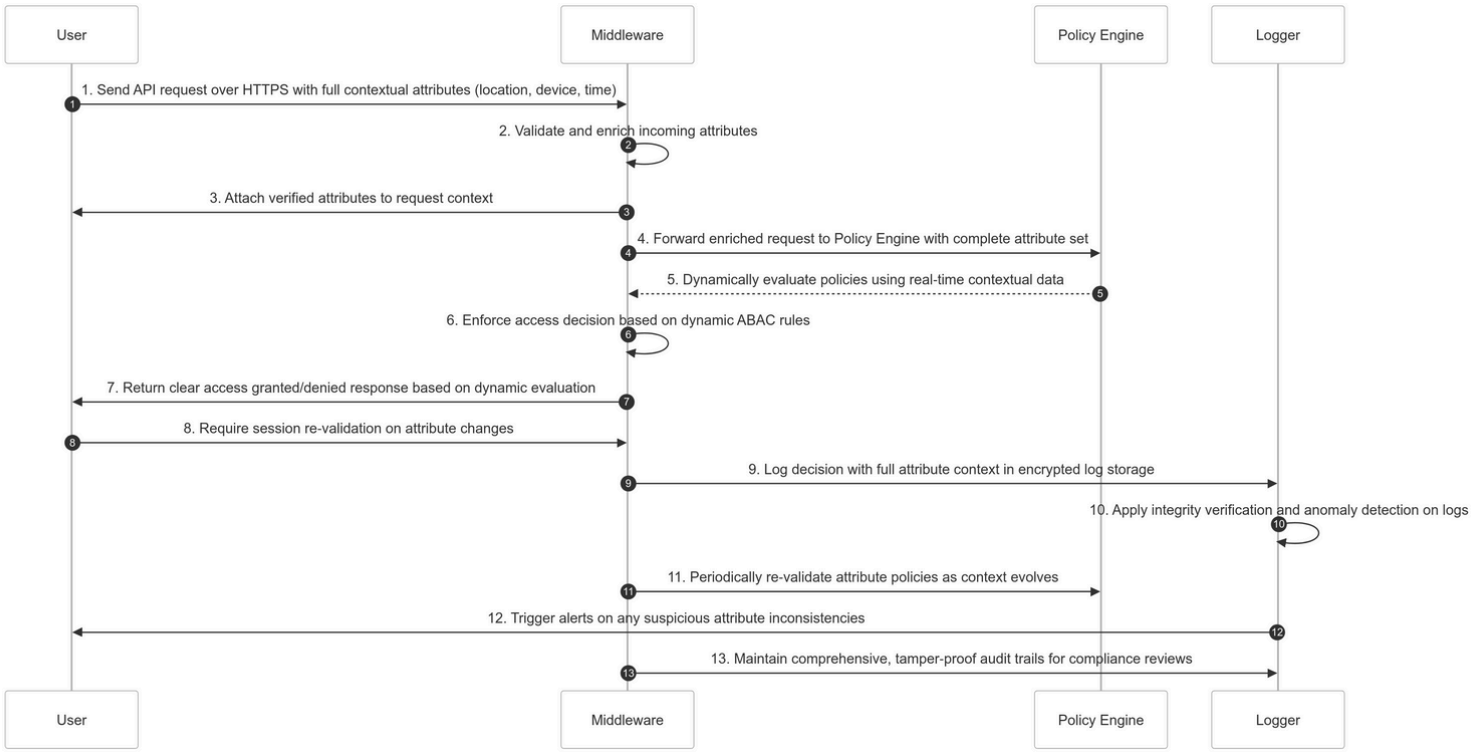
Attribute-Based Access Control (ABAC) Middleware

- **Architect:** Deploy middleware that leverages contextual attributes (user location, time, device type) to enforce access policies dynamically. Policies are managed centrally and evaluated in real time.
- **Attacks/TTPs:** Context spoofing, manipulation of attribute data, insider misuse.
- **Standards for Security Testing and Best Practices:** NIST SP 800-162, XACML policy testing, continuous threat modeling and validation.
- **Roles and Users:** Policy engineers, security analysts, compliance teams; internal employees, SaaS users.

Insecure Implementation – ABAC Middleware



Secure Implementation – ABAC Middleware

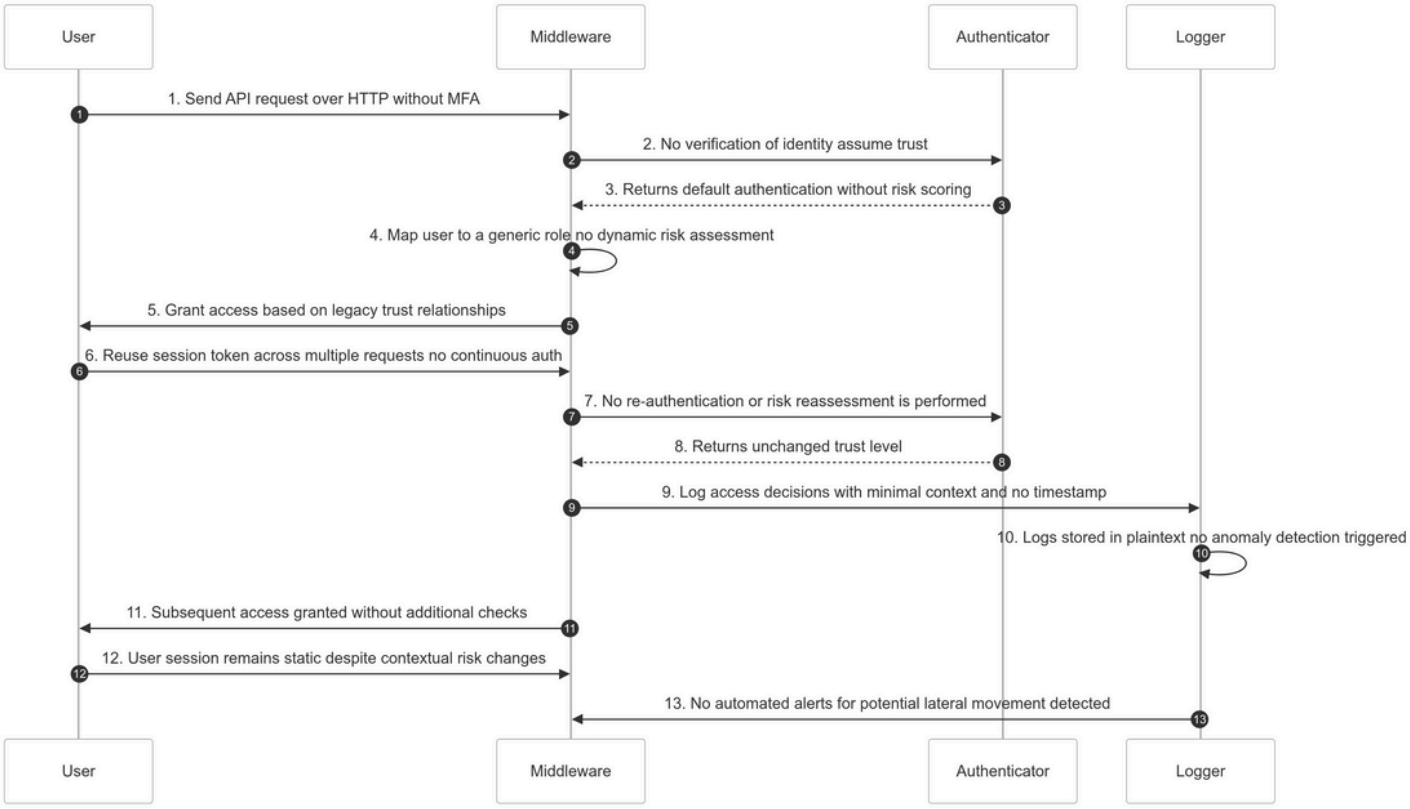


Stage	Insecure	Secure	Best Practice
Request Reception	• Minimal attribute data • No validation/enrichment	• Full HTTPS-based request with complete attributes • Attribute enrichment and validation	Collect and validate full contextual data
Attribute Evaluation	• Static rules; no dynamic context • Hardcoded policy checks	• Dynamic evaluation using real-time contextual data	Use dynamic, context-aware attribute evaluation
Access Decision	• Access decision based on defaults only	• Fine-grained decision incorporating dynamic attributes	Enforce dynamic ABAC; require periodic re-validation
Logging & Audit	• Plaintext logs without context • No alerts on anomalies	• Encrypted, detailed logging with integrity checks and automated anomaly alerts	Log comprehensively and continuously monitor for context anomalies

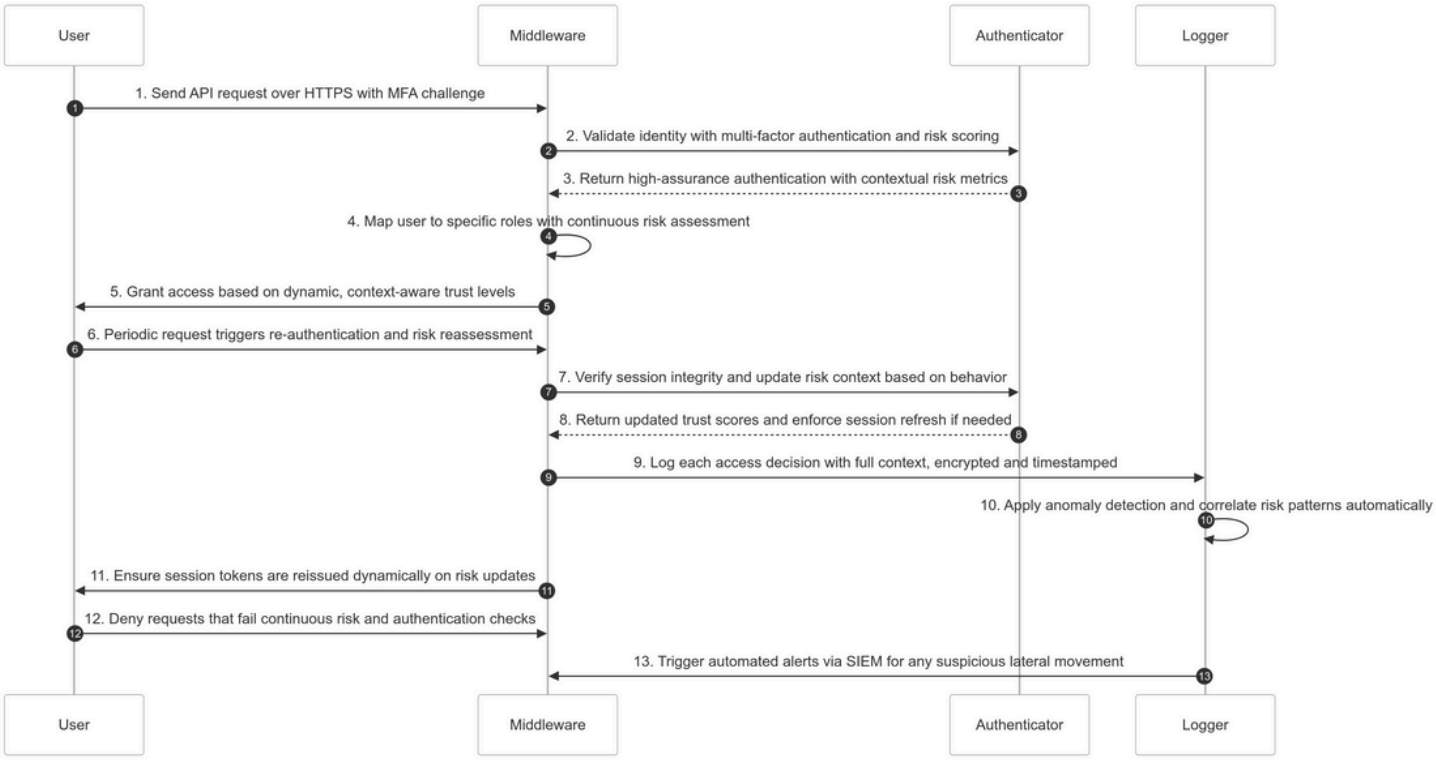
Zero Trust Middleware Enforcement

• **Architect:** Design a middleware layer that implements Zero Trust principles by verifying every request with multi-factor authentication and continuous risk assessment before granting access.
• **Attacks/TTPs:** Lateral movement, session hijacking, zero-day exploitation of trust relationships.
• **Standards for Security Testing and Best Practices:** NIST Zero Trust Architecture, MITRE ATT&CK assessments, continuous authentication testing.
• **Roles and Users:** CISO, network security teams, risk analysts; all organizational users accessing sensitive resources.

Insecure Implementation – Zero Trust Middleware



Secure Implementation – Zero Trust Middleware



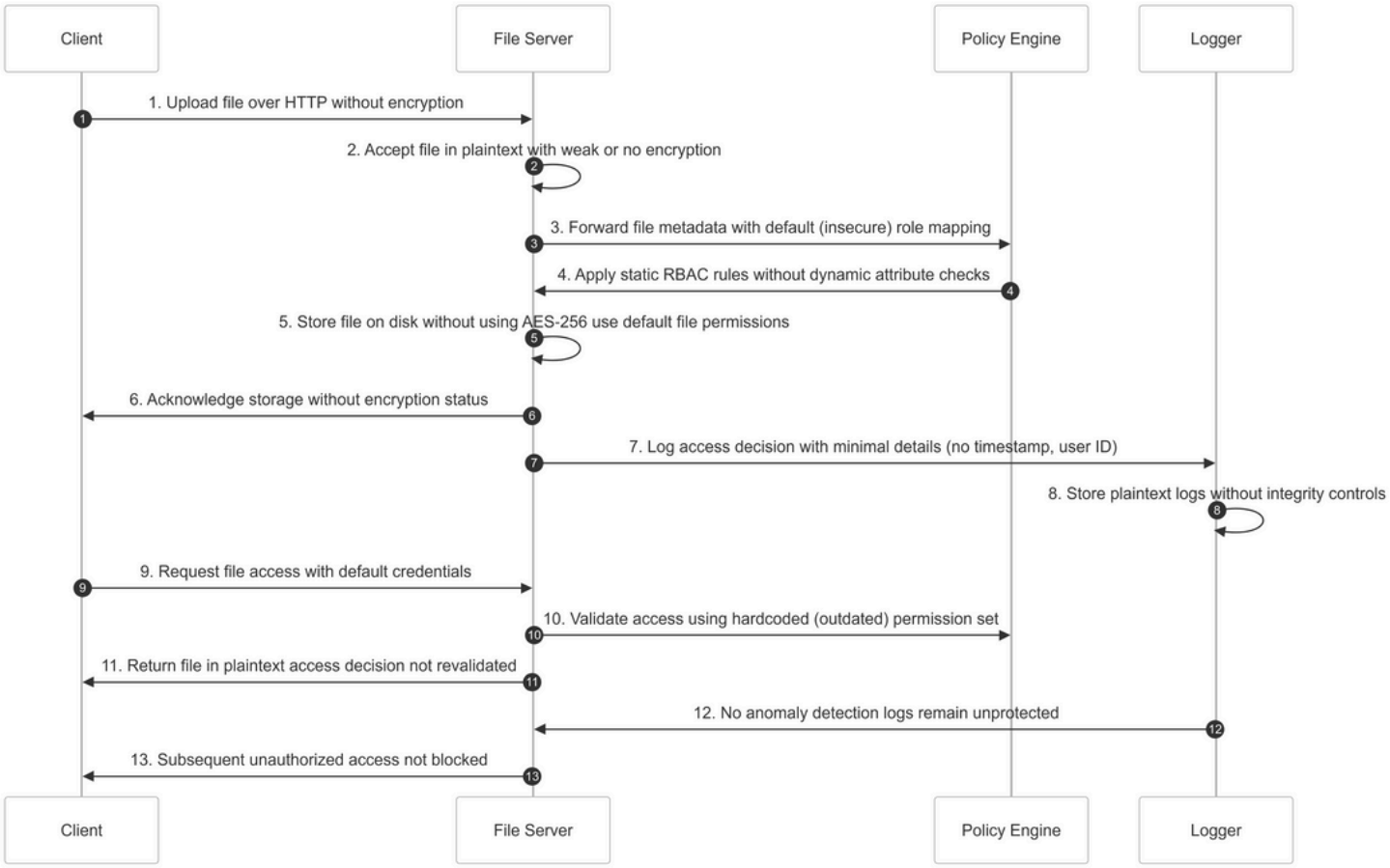
Stage	Insecure	Secure	Best Practice
Request Verification	• No MFA • Default authentication with no risk scoring	• HTTPS with MFA • Dynamic risk scoring with contextual metrics	Enforce MFA and continuous risk-based authentication
Trust Establishment	• Generic role mapping; static trust	• Dynamic, context-aware trust mapping based on updated risk assessments	Continuously assess trust based on behavioral and contextual data
Session Re-Authentication	• Reuse of static session tokens with no re-evaluation	• Periodic re-authentication and session refresh on risk changes	Require continuous authentication and session refresh
Logging & Audit	• Minimal logging; stored in plaintext; no alerts	• Encrypted, detailed logging with anomaly detection and real-time SIEM alerts	Log with full context; integrate with SIEM and trigger automated alerts

Secure Execution/File Management

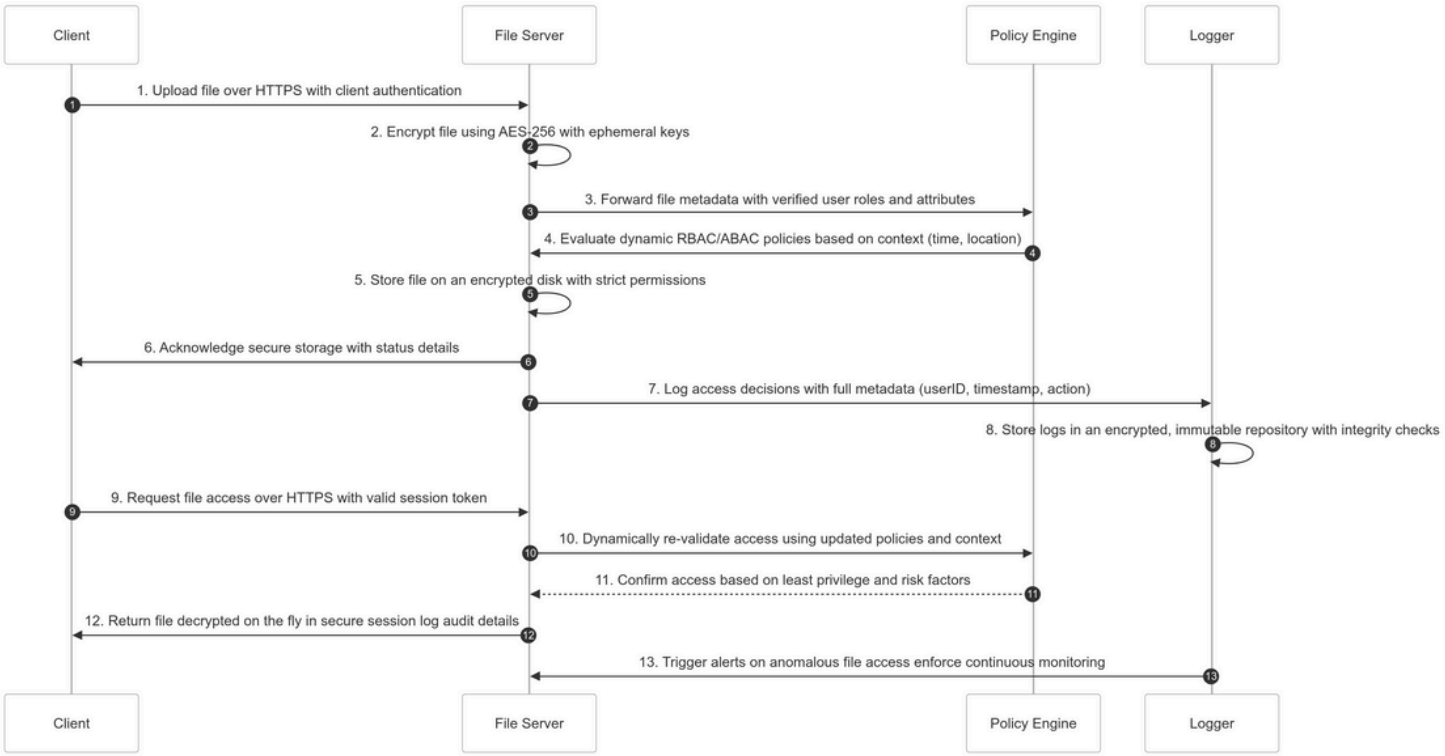
Encrypted File Storage with Granular Access Controls

- **Architect:** Design a multi-layered file storage system that encrypts data at rest using AES-256 and applies RBAC/ABAC policies at the file system level with audit logging.
- **Attacks/TTPs:** Data exfiltration via stolen credentials, unauthorized file access, brute-forcing weak cryptographic keys.
- **Standards for Security Testing and Best Practices:** NIST SP 800-57, FIPS 140-2, regular penetration tests, and automated file access reviews.
- **Roles and Users:** Security architects, file system administrators, compliance officers; end-users, backup operators.

Insecure Implementation – Encrypted File Storage



Secure Implementation – Encrypted File Storage



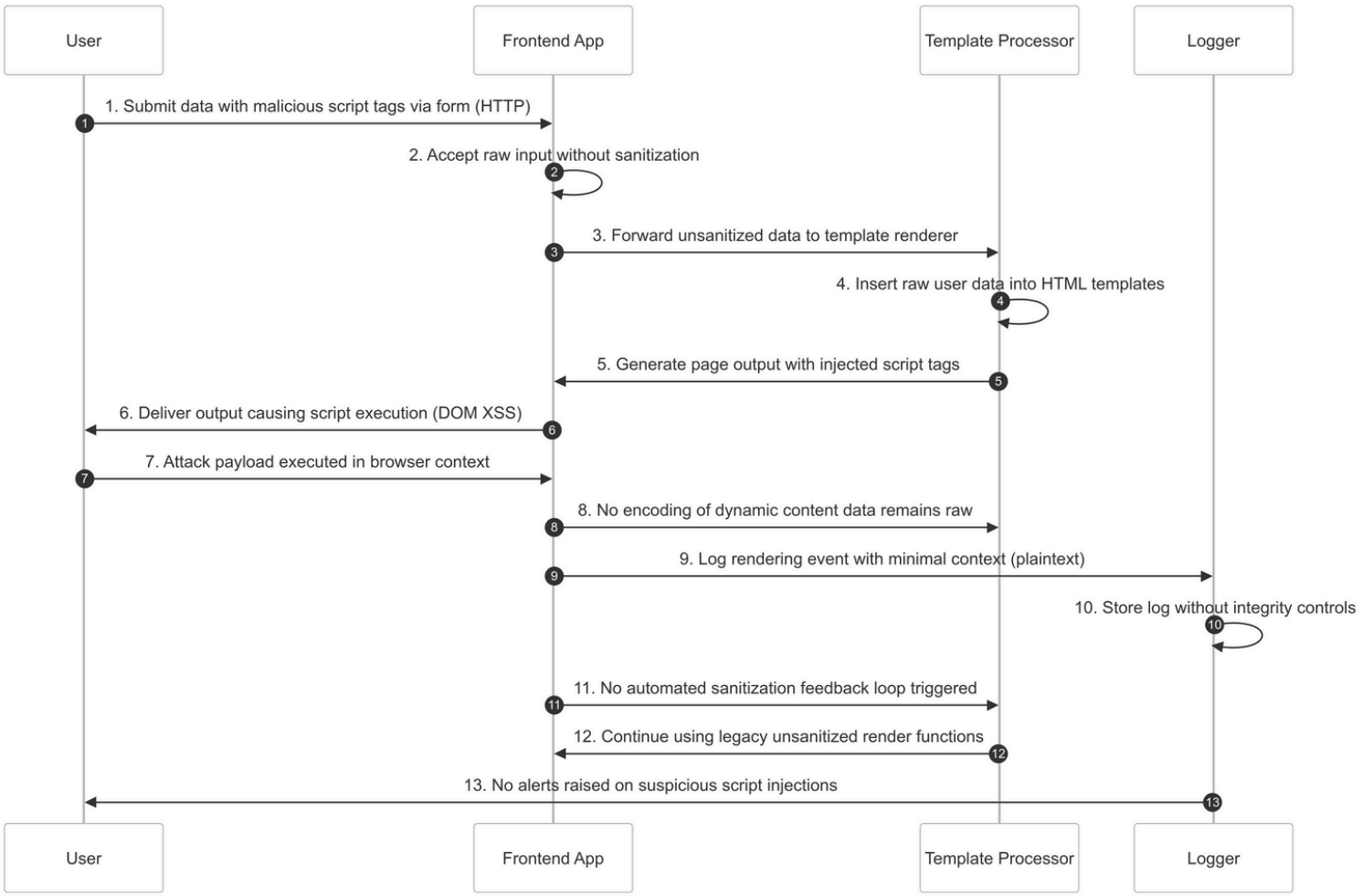
Stage	Insecure Implementation	Secure Implementation	Best Practice
Request Intake	• Unencrypted HTTP upload • No client authentication • Weak/no encryption applied	• HTTPS upload with client authentication • AES-256 file encryption • Verified metadata passed	Enforce TLS/HTTPS; require strong encryption and authentication
Access Evaluation	• Static, hardcoded RBAC • No dynamic ABAC • Default file permissions	• Dynamic RBAC/ABAC based on time, location, etc. • Strict file system ACLs	Evaluate access using dynamic, context-aware policies
Audit Logging	• Minimal details; plaintext logs; no integrity controls	• Detailed, encrypted log entries with full audit trail and integrity checks	Log all access details securely and integrate with SIEM
File Retrieval	• Plaintext file return • No re-validation on each request, allowing unauthorized reuse	• Continuous re-validation of access tokens • Secure, on-the-fly decryption within an active session	Re-validate on each access; enforce short-lived tokens and session monitoring

Front-end

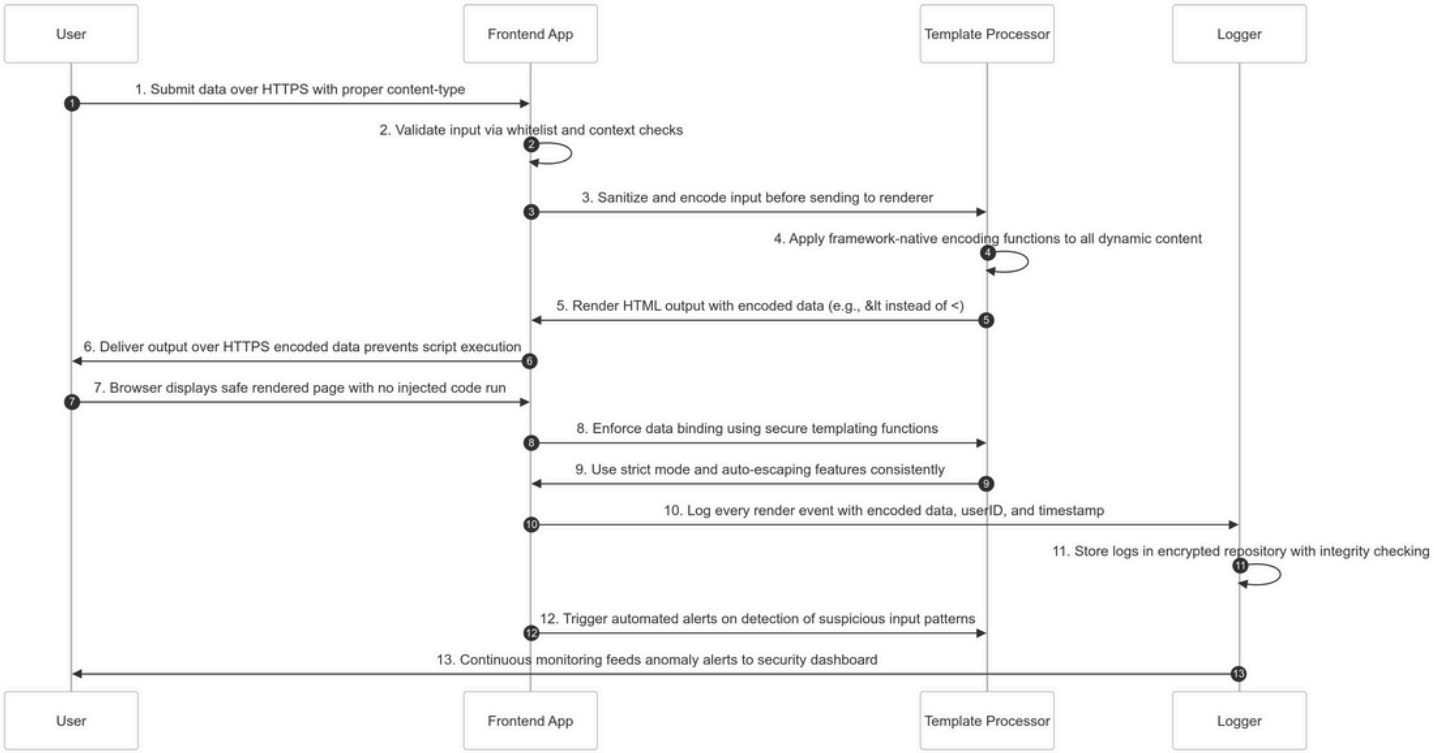
Secure Output Encoding

- **Architect:**
  Design a multi-stage client-side rendering pipeline that automatically encodes output from all user-supplied data in templates and components, leveraging framework-native sanitization libraries.
- **Attacks/TTPs:**
  Cross-Site Scripting (XSS), DOM-based XSS, injection attacks.
- **Standards for Security Testing and Best Practices:**
  OWASP XSS Prevention Cheat Sheet, SANS Top 25, automated SAST scans using ESLint plugins.
- **Roles and Users:**
  Frontend developers, security architect, QA/testers; end-users accessing the web application.

Insecure Implementation – Output Encoding & XSS Prevention



Secure Implementation – Output Encoding & XSS Prevention

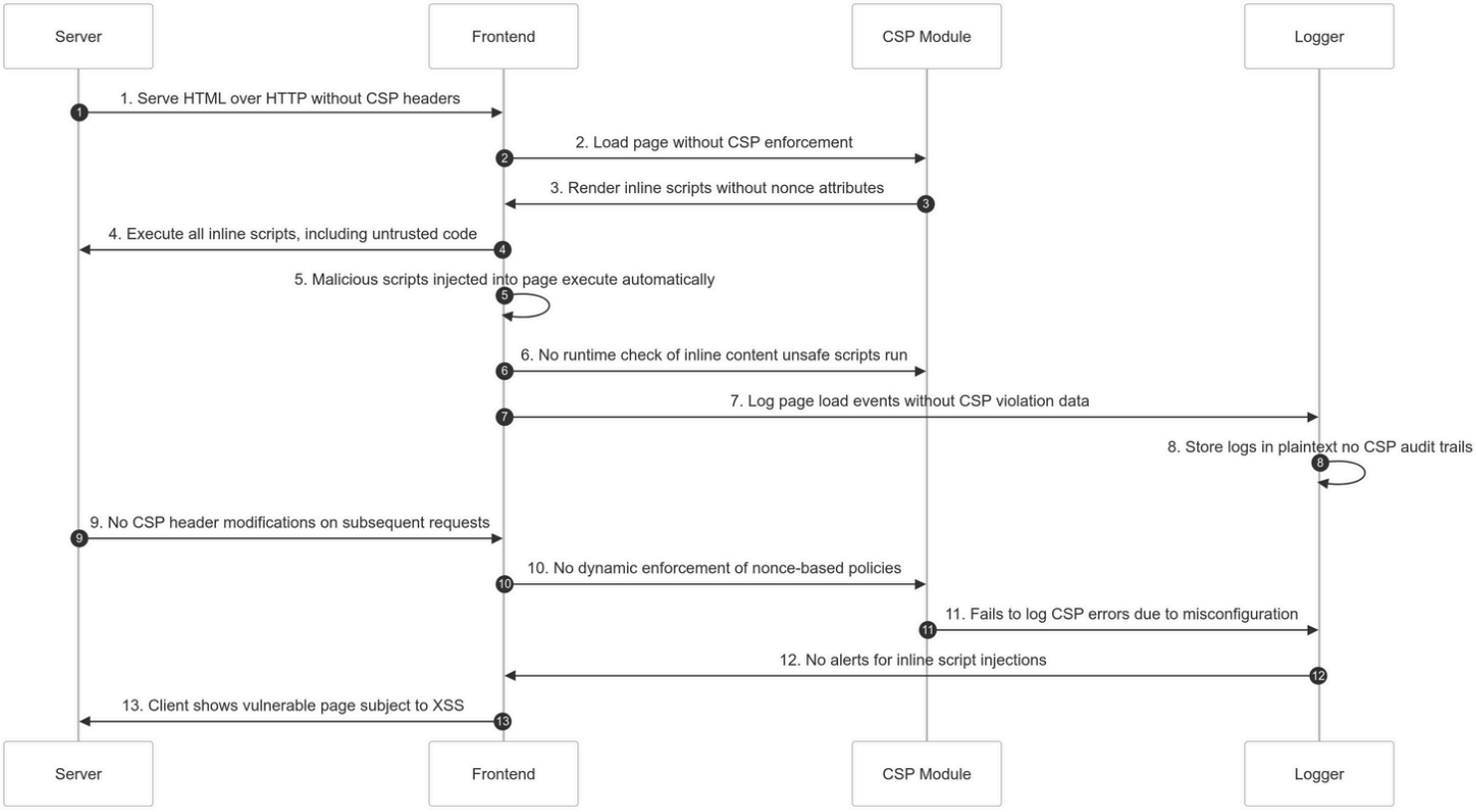


Stage	Insecure Implementation	Secure Implementation	Best Practice
Data Input	• Accepts raw input; no validation • Uses HTTP	• Uses HTTPS; validates data against whitelist; performs encoding	Secure transmission and input validation
Template Rendering	• Inserts raw input directly • No output encoding	• Applies framework-native auto-escaping and context-sensitive encoding	Always encode output appropriately
Browser Rendering	• Delivers output that executes injected scripts	• Delivers fully encoded output preventing XSS	Prevent script execution via proper sanitization
Logging & Monitoring	• Minimal, plaintext logging; no alerts • No monitoring of injections	• Detailed, encrypted logs; automated anomaly alerts integrated into SIEM	Implement comprehensive logging and real-time alerting

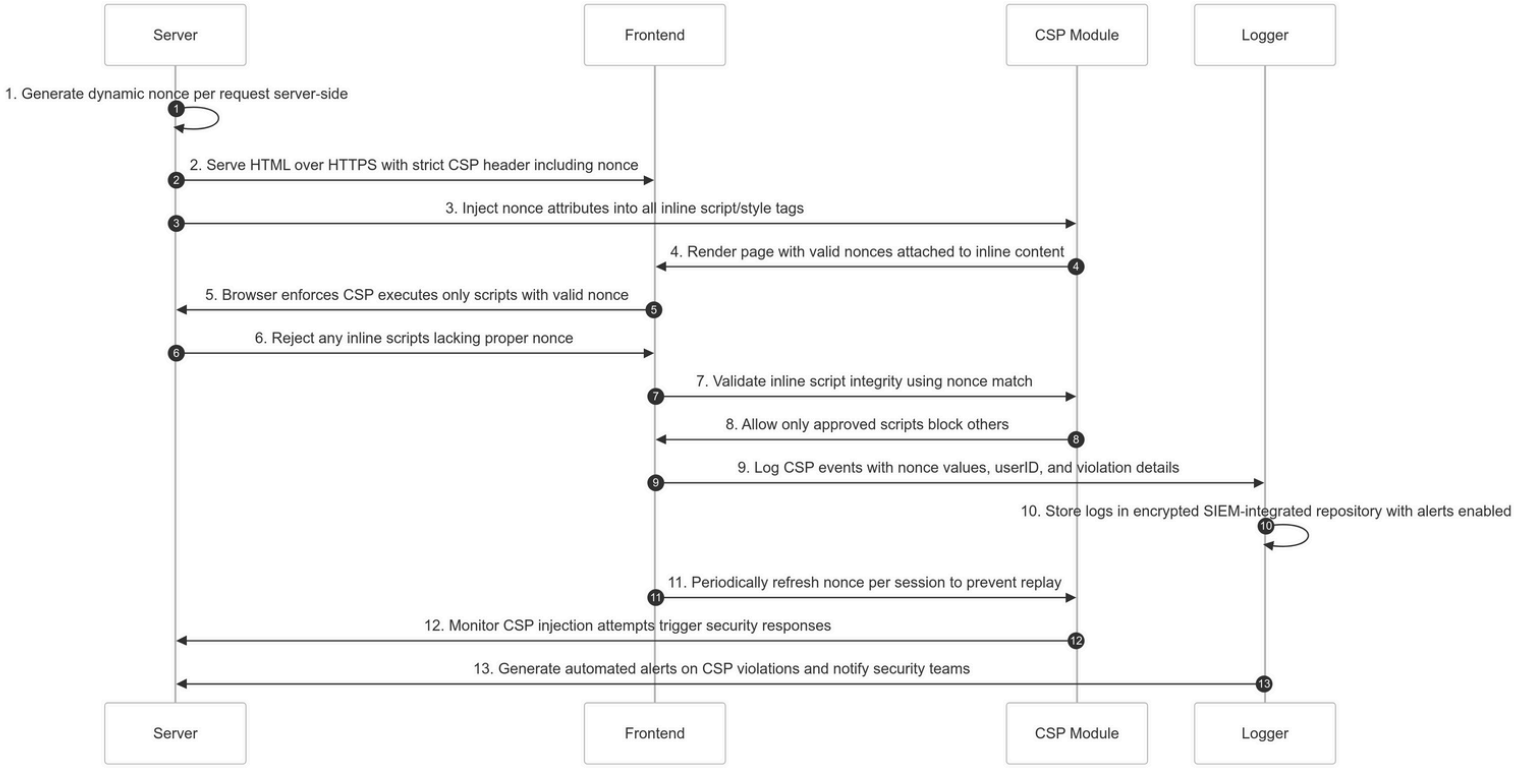
Content Security Policy (CSP) Enforcement with Nonce-based Scripts

- **Architect:**
  Implement a strict CSP with dynamic nonce generation on script tags and inline styles to ensure that only approved scripts are executed.
- **Attacks/TTPs:**
  Inline script injection, code injection, and mixed-content attacks.
- **Standards for Security Testing and Best Practices:**
  OWASP CSP Cheat Sheet, NIST SP 800-53 rev. 4 AC-3, periodic CSP header audits.
- **Roles and Users:**
  Security engineers, DevOps, frontend architects; web users and API clients.

Insecure Implementation – CSP without Nonce Enforcement



Secure Implementation – CSP with Nonce-based Scripts



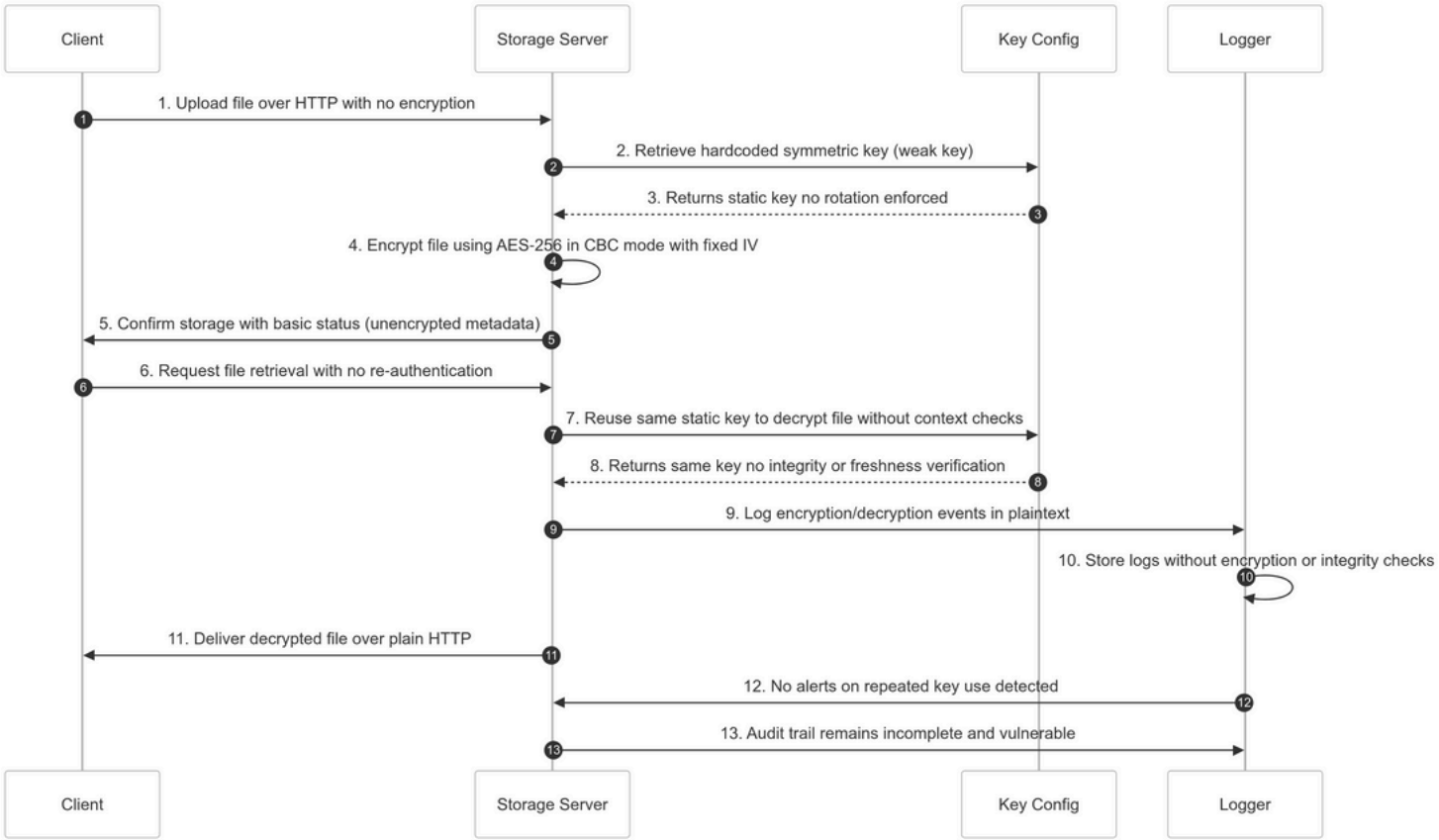
Stage	Insecure Implementation	Secure Implementation	Best Practice
Response Delivery	• No CSP headers • Served over unsecured HTTP	• HTTPS with strict CSP headers including dynamic nonce	Always serve content over HTTPS with strict CSP
Nonce Generation	• No dynamic nonce; inline scripts lack nonce attributes	• Dynamic nonce injected into all inline script/style tags	Use dynamic nonce on inline scripts to prevent injection
CSP Enforcement	• Browser executes all inline scripts	• Browser enforces CSP; only scripts carrying the valid nonce are executed	Enforce CSP at runtime to restrict script execution
Logging & Monitoring	• Minimal logging; no CSP audit trail	• Detailed, encrypted logging of all CSP events with automated alerts	Monitor CSP violations in real time and integrate with SIEM

Cryptography

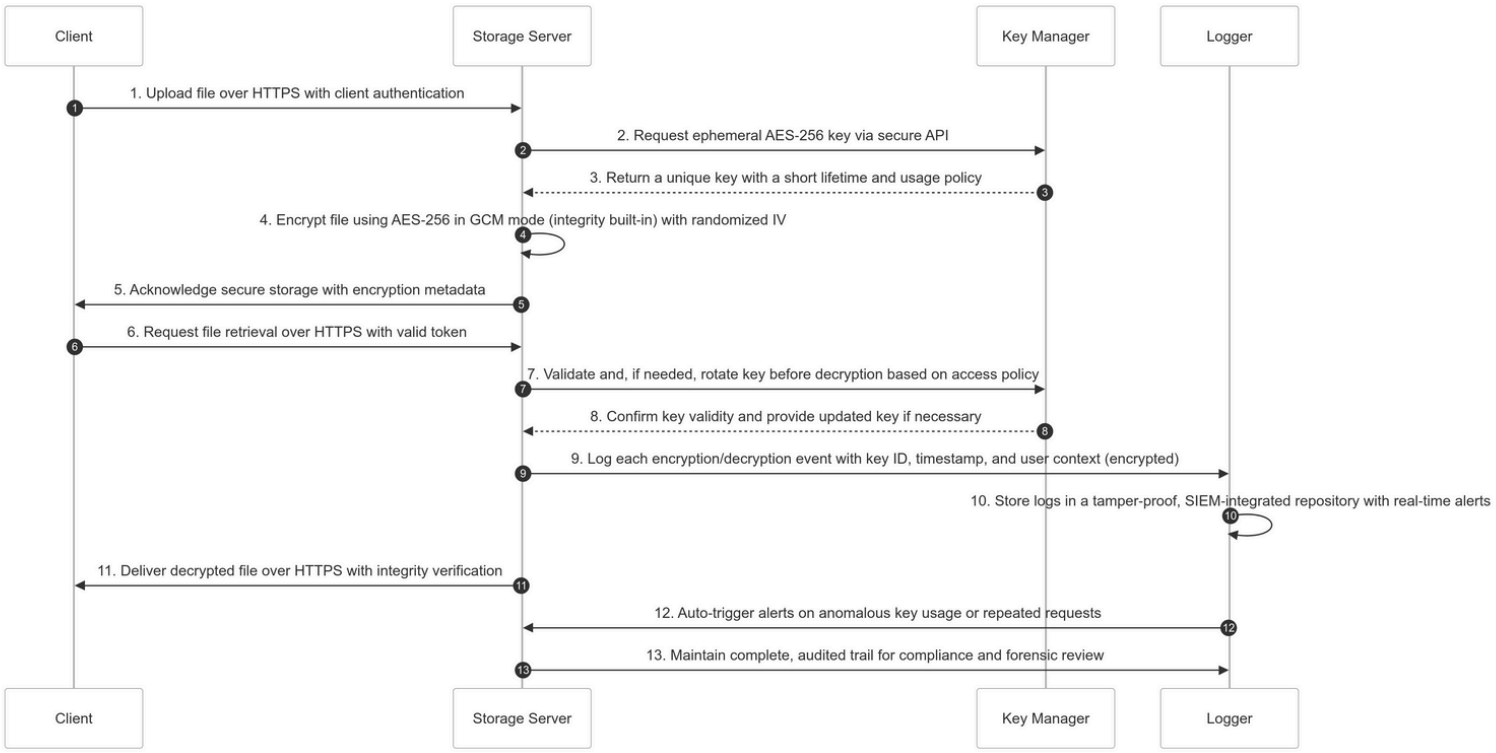
Symmetric Encryption for Data-at-Rest

- **Architect:**
  Utilize AES-256 in GCM or CBC mode for encrypting files and databases. Multi-stage implementation covers encryption during storage, secure key management, and decryption controlled by access policies.
- **Attacks/TTPs:**
  Side-channel attacks, key-reuse vulnerabilities, padding oracle attacks.
- **Standards for Security Testing and Best Practices:**
  NIST SP 800-38, FIPS 197, periodic cryptographic code reviews and penetration tests.
- **Roles and Users:**
  Data protection officers, storage admins; cloud engineers, backup operators.

Insecure Implementation – Symmetric Encryption



Secure Implementation – Symmetric Encryption

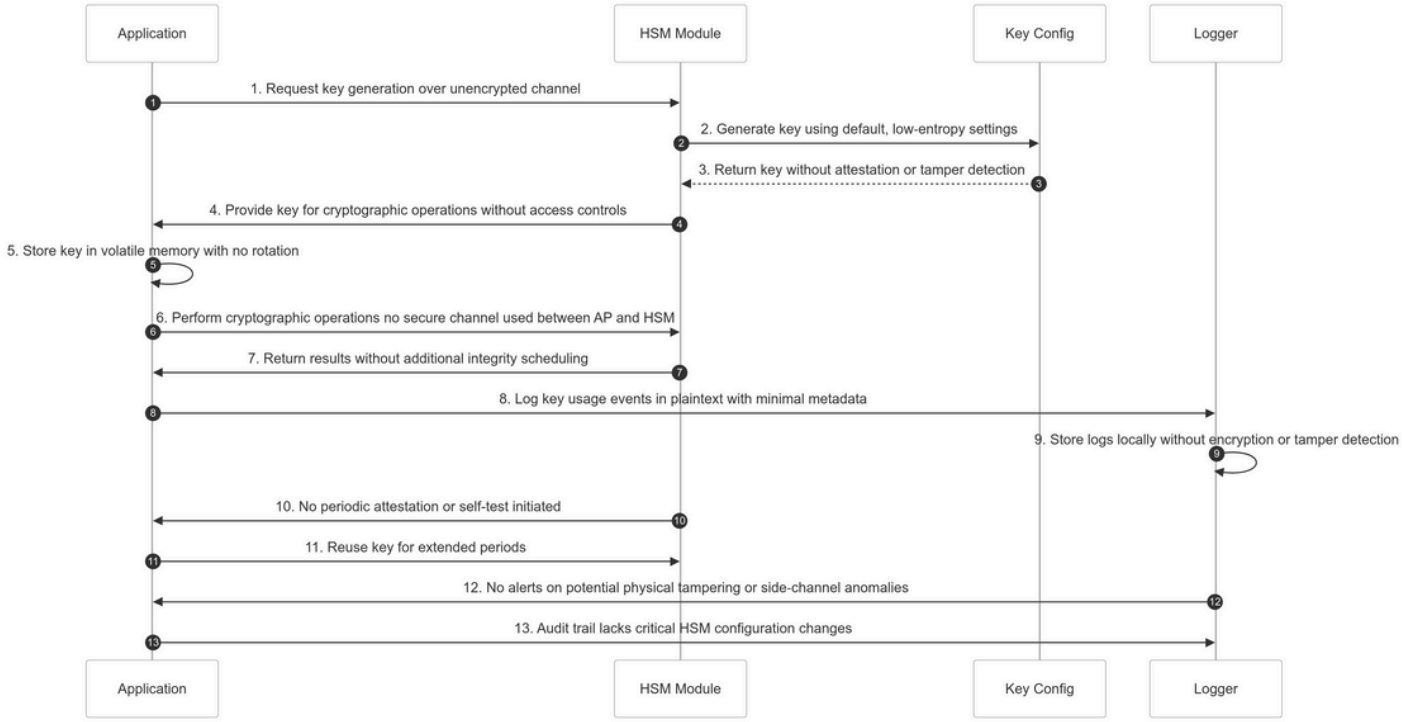


Stage	Insecure Implementation	Secure Implementation	Best Practice
Data Upload & Key Retrieval	• Upload over HTTP; hardcoded static key; no rotation	• HTTPS upload; dynamic ephemeral key fetched via secure API; key lifetime enforced	Use TLS, dynamic key generation, and enforce key rotation
Encryption Process	• Uses AES-256 CBC with fixed IV; no integrity checks	• Uses AES-256 GCM with randomized IV; built-in integrity verification	Use strong modes with random IVs and integrity guarantees
Access & Decryption	• No re-authentication; reused static key for decryption; no context check	• Dynamic re-authentication and key validation before decryption	Re-validate access continuously and rotate keys if needed
Logging & Audit	• Plaintext logs; no integrity; no alerts	• Encrypted, SIEM-integrated logging; automated alerts on anomalies	Secure logging with automated monitoring and complete audit trails

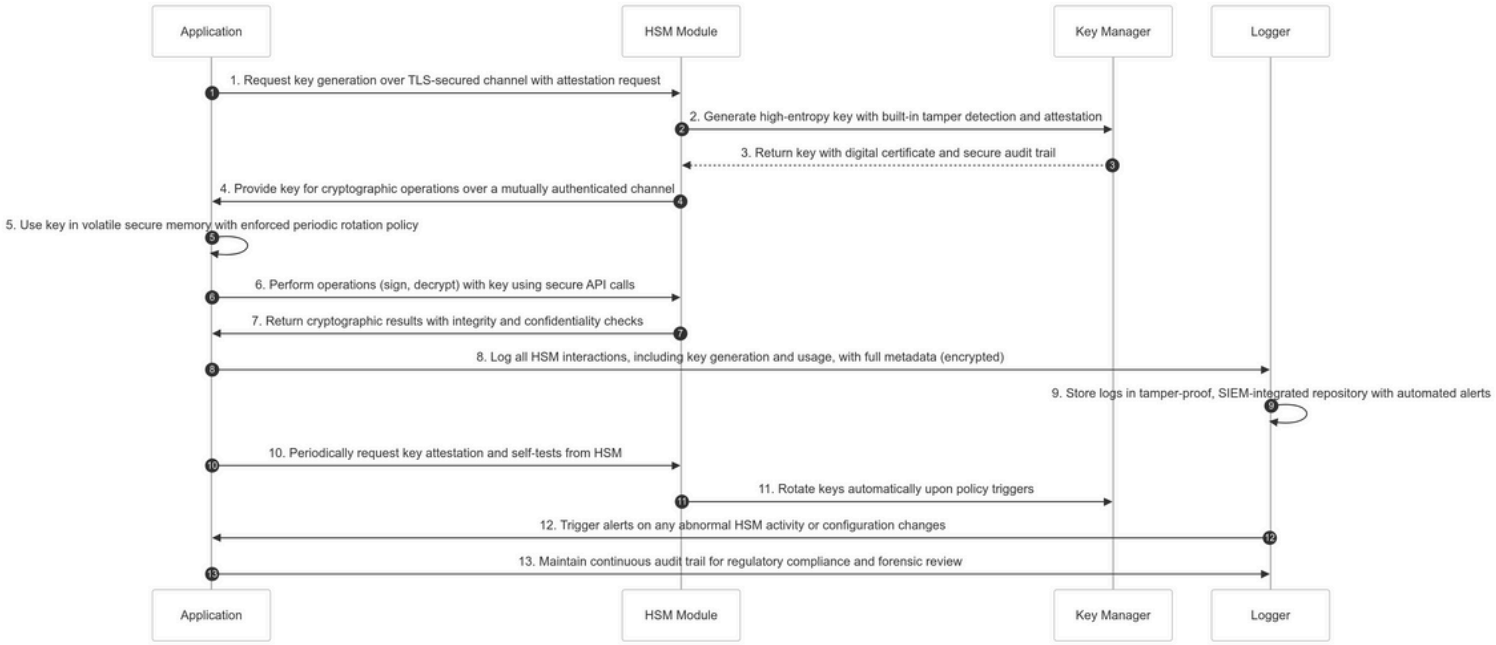
Hardware Security Module (HSM) Integration

- **Architect:**
  Deploy HSMs for secure key generation, storage, and cryptographic operations. Multi-stage integration covers key lifecycle management, access control policies on HSMs, and secure channel communication with applications.
- **Attacks/TTPs:**
  Physical tampering, side-channel leakage, key extraction via hardware exploits.
- **Standards for Security Testing and Best Practices:**
  FIPS 140-2/3, Common Criteria, periodic HSM penetration testing and attestation reviews.
- **Roles and Users:**
  Hardware security engineers, cryptographic key managers; enterprise security operations, compliance teams.

Insecure Implementation – HSM Integration



Secure Implementation – HSM Integration



Stage	Insecure Implementation	Secure Implementation	Best Practice
Key Generation	• Unencrypted requests; low-entropy keys; no attestation	• TLS-secured key requests; high-entropy keys with digital attestation; tamper detection	Use secure channels and require attestation for key generation
Key Storage/Usage	• Key stored insecurely; reused for extended periods; no dynamic rotation	• Keys used only transiently in secure memory; periodic rotation enforced with mutual auth	Enforce strict key lifetime and rotation policies
Cryptographic Operations	• Operations performed over unsecure channels; no integrated integrity checks	• Secure API calls between app and HSM; integrity and confidentiality verified within operations	Use mutually authenticated channels and verified cryptographic operations
Logging & Audit	• Plaintext logs; no SIEM integration; missing tamper alerts	• Encrypted, SIEM-integrated logs with detailed HSM event tracking and automated alerts	Log all HSM events securely and monitor continuously