Cryptographic Security in Online Banking Login: Detailed Step-by-Step Analysis

Aniket Basak

1 Question:

Suppose we are logging through SBI bank account using browser. Then it is going through which port? How the communication get established? What is the public key encryption algorithm? What is private key algorithm also? How after putting the username and password , security is preserved? Why password looks black dots, what is happening in background, how verification is done?

1. **DNS & address resolution:**

   If we type https://sbi.co.in (or click a bookmark). The browser resolves the hostname to IP via DNS. DNS may be plain UDP/UDP+TCP, or DoH/DoT (encrypted DNS); if DNS is poisoned it's an initial attack vector, but TLS certificate checks protect us from a wrong host responding unless the attacker also has a valid cert.

2. **TCP connection (port):**

   Browser opens a TCP connection to the bank's server at the server IP on TCP port 443 (standard HTTPS). If a proxy/load-balancer sits in front, our TCP actually terminates at that device.

   TCP 3-way handshake: SYN → SYN/ACK → ACK.

3. **TLS handshake (server authentication + key exchange):**

   This is the critical cryptographic step. Modern banks use TLS (ideally TLS 1.3). Conceptual flow (TLS 1.3 flavor shown because it's current):

   - **ClientHello**—Browser sends supported TLS versions, supported cipher suites (e.g., TLS_AES_128_GCM_SHA256), signature algorithms, and a key_share (an ephemeral public key for ECDHE like X25519 or P-256).
   - **ServerHello**—Server picks a cipher suite and sends its ephemeral key share.
   - **Certificate**—Server sends its certificate chain (leaf certificate for *.sbi.co.in signed by a CA). That certificate contains the server's public key (either RSA or an EC public key).
   - Public key algorithm used in the certificate is typically RSA (2048/3072 bits) or ECDSA (P-256/P-384) using ECC (secp256r1).
   - **CertificateVerify**—Server proves it holds the private key corresponding to the cert by signing parts of the handshake.
   - Both sides use the ephemeral ECDHE shared secret plus the transcript to derive symmetric keys via a KDF (HKDF in TLS 1.3). Ephemeral keys provide forward secrecy.
   - Finished messages finalize the handshake; now both have identical symmetric AEAD keys for encryption/MAC.

   Note: The public-key algorithm purpose is certificate signatures (server identity). The key-exchange algorithm (ECDHE) purpose is to create a shared symmetric key with forward secrecy. The private key is the server's certificate private key (RSA or EC) usually stored in an HSM (Hardware Security Module) and used to sign (prove identity) not to encrypt bulk traffic.

   After the handshake all HTTP traffic runs inside TLS using AEAD symmetric ciphers (AES-GCM / ChaCha20-Poly1305).

4. **Certificate validation & anti-MitM checks** The browser validates the cert chain:
   - Checks that the cert is issued by a trusted CA.
   - Verifies validity period (not expired).
   - Checks domain match (SAN/CN covers the hostname).
   - Performs OCSP/CRL/OCSP-stapling checks where possible.

   If validation fails, the browser warns you (lock icon missing or warning shown).
5. **HTTPS request for login page (encrypted)** Browser sends `GET /login` inside TLS. Server responds with HTML/JS/CSP headers and usually sets cookies (CSRF token, tracking tokens, etc.). All of this is encrypted on the wire.
6. **Login page content & client-side protections** The login page often includes:
   - CSRF tokens embedded in HTML forms.
   - JavaScript for client-side checks, UI, or sometimes client-side crypto.
   - Masked `<input type="password">` for UI.
7. **We type username + password (UI only)** The password field masks characters visually but the actual password string exists in browser memory/DOM.
8. **Submit: HTTPS POST over the established TLS session** When we hit "Login" the browser sends an HTTPS POST with username/password inside the TLS tunnel.
9. **Server receives credentials — server side checks**
   - Validates CSRF token, request sanity, rate limits, IP behavior.
   - Password verification: retrieves stored salted slow hash (Argon2/bcrypt/PBKDF2).
   - Computes hash(candidate_password, salt, parameters).
   - Compares in constant-time to stored hash.
10. **Session creation & cookies** If authentication succeeds:
    - Server creates a session object + random session token.
    - Returns `Set-Cookie: session=<opaque-token>; Secure; HttpOnly; SameSite=Strict`.
11. **Post-login protections & MFA**
    - MFA: OTPs, authenticator apps, hardware tokens.
    - Device fingerprinting and risk engines.
    - Transaction authentication and re-authentication for sensitive ops.
12. **How OTPs and transaction verification work**
    - **SMS OTP**: server generates CSPRNG code, stores hash, sends via SMS.
    - **TOTP**: shared secret + time (HMAC-SHA1/256).
    - **Transaction signing**: challenge-response using tokens or soft tokens.
13. **How TLS defends against network attackers** TLS ensures confidentiality, integrity, and authentication. Forward secrecy: ECDHE. Session resumption: banks may disable 0-RTT to avoid replay risks.
14. **Private key management (server side)** Private key stored in HSM:
    - Prevents key extraction.
    - Performs signing inside hardware.
    - Controlled access and audit.

15. **Additional HTTP/Security headers & defenses**
    - CSP (mitigates XSS).
    - X-Frame-Options / frame-ancestors (prevent clickjacking).
    - HSTS (force HTTPS).
    - X-Content-Type-Options: nosniff.
16. **Protecting sessions against theft and replay**
    - HttpOnly + Secure cookies.
    - Session timeouts, inactivity timers.
    - Device binding.
    - Anti-replay tokens for sensitive actions.
17. **Password field (black dots)** `<input type="password">` masks characters visually. Password string still exists in browser memory. Masking prevents shoulder-surfing but not malware/keyloggers.
18. **Verification done — full picture**
    - Client authenticates server via cert.
    - Symmetric keys derived via ECDHE + KDF.
    - Credentials sent inside TLS.
    - Server verifies credentials.
    - Session token issued + MFA if needed.
    - Transaction requests protected by OTP/signature.
19. **Common cryptographic primitives & their roles**
    - Port: 443 (HTTPS).
    - Server identity: RSA/ECDSA key.
    - Key exchange: ECDHE (X25519/P-256) using ECC (secp256r1).
    - Bulk encryption: AES-GCM / ChaCha20-Poly1305.
    - Hash/KDF: SHA-256, HKDF.
    - Password hashing: bcrypt/scrypt/Argon2.
    - Randomness: CSPRNG.
    - Signing: RSA/ECDSA.
    - Revocation: OCSP/CRL.
20. **Attack vectors and mitigations**
    - Phishing → education, monitoring.
    - MitM → TLS, HSTS, pinning.
    - Malware/keyloggers → OTPs, device fingerprinting.
    - Session hijack/XSS/CSRF → HttpOnly cookies, CSP, CSRF tokens.
    - Database leaks → salted slow hashes.
    - Private key theft → HSMs, restricted access, rotation.

2 Question: Who Provides the Certificate?

In the SBI login flow, the TLS/SSL certificate is issued by a **trusted Certificate Authority (CA)**. The process works as follows:

1. **Key Generation by SBI:** SBI generates a cryptographic key pair:
   - The **private key** is stored securely on the bank's servers, usually inside a Hardware Security Module (HSM).
   - The **public key** is included in a Certificate Signing Request (CSR).
2. **Certificate Signing Request (CSR):** SBI submits the CSR to a public CA (such as DigiCert, Entrust, GlobalSign, etc.). The CSR contains:
   - The domain name (e.g., `*.sbi.co.in`).
   - The bank's public key.
   - Organization details required for identity verification.
3. **Validation by the CA:** The CA verifies SBI's identity and domain ownership. For banks, this involves extended validation checks (legal entity verification, domain control, etc.).
4. **Certificate Issuance:** After successful validation, the CA issues a certificate that:
   - Binds SBI's domain name (`sbi.co.in`) to its public key.
   - Is digitally signed by the CA's private key.
5. **Trust by Browsers:** Web browsers and operating systems trust this certificate because the CA's root certificate (or intermediate certificates) is already present in their **trust store**.

Thus, the certificate is:

- **Issued by:** a public Certificate Authority (CA).
- **Presented by:** SBI's servers during the TLS handshake.

References

1. "Cryptography Engineering" by Niels Ferguson, Bruce Schneier, Tadayoshi Kohno.
2. ChatGPT and DeepSeek.