ChatGPT

Security & Compliance Framework for a Financial App

Designing a robust security and compliance framework for a financial application requires addressing authentication, data protection, regulatory compliance, threat modeling, and governance. Below we outline best practices and controls in each area, tailored for a web-based personal/business finance app (with future iOS mobile support) that handles banking, payments, and tax data.

1. Authentication & OAuth Integration

Secure OAuth Flows: The app will integrate with external services (Open Finance APIs for banks, Stripe, Gmail, Mercado Pago) via OAuth 2.0. Use the Authorization Code flow (with PKCE for mobile) so that users authenticate directly with the provider, and our app receives an access token – **never store or ask for the user's third-party passwords**. Ensure OAuth scopes are limited to the minimum data needed (principle of least privilege for API access) [1]. For example, use read-only scopes for bank balance/transaction data and limit payment scopes on Stripe/Mercado Pago to necessary operations.

Token Management & KMS: All access tokens, refresh tokens, API keys, and other credentials should be encrypted at rest using a Key Management Service (KMS). A best practice is to use **envelope encryption**: generate a data-encryption key via KMS and use it to encrypt the token, storing only the encrypted token and the encrypted data-key in the database [2] [3]. This way, the plaintext token is only in memory briefly and requires a call to KMS (with proper IAM permissions) to decrypt when needed. For example, on token storage:

```
# Pseudocode for token encryption using AWS KMS
data_key = KMS.generate_data_key(key_alias="alias/AppMasterKey")
encrypted_token = AES256_encrypt(token, data_key.plaintext)
store(db, encrypted_token, data_key.encrypted)
# (Discard plaintext key/token from memory)
```

On retrieval, the service decrypts the data_key via KMS, then uses it to decrypt the token. This ensures that even if the DB is compromised, tokens remain unusable without KMS access [2]. **Never hard-code secrets** – use secure storage (cloud Secrets Manager or environment variables) for API keys and client secrets, and **enable automatic key rotation** for KMS master keys to limit exposure [2].

User Authentication & MFA: For the app's own login, support secure password-based auth (with bcrypt hashing) and offer SSO via Google (OAuth) if possible. Enforce strong passwords or passphrases and implement **multi-factor authentication (MFA)** for sensitive accounts (especially business users) [4]. MFA (e.g. one-time codes or authenticator app) mitigates credential stuffing and stolen password reuse. To further protect against **credential stuffing** attacks, implement defenses like CAPTCHAs on login after several failed attempts, IP rate limiting on auth endpoints, and monitoring for unusual login patterns [1].

All login and password reset forms must use HTTPS and be secured against man-in-the-middle; consider using **WebAuthn** or OAuth social login to eliminate passwords.

Session Management: Use secure session management practices to prevent hijacking. If using web sessions, issue **session IDs as HTTP-only, Secure cookies** so they aren't accessible to JavaScript (protects against XSS stealing the token) [5]. Set the `SameSite` attribute (Lax or Strict) to defend against CSRF. On login, create a new session identifier (avoid session fixation by regenerating session IDs upon privilege changes) [5]. Tie sessions to user context where appropriate (e.g. IP address or device fingerprint) so stolen cookies are less useful [6]. Idle session timeouts (e.g. auto-logout after 15 minutes of inactivity) and absolute expiration (e.g. 8-12 hours) reduce risk of unattended sessions being misused [7]. For mobile API usage, prefer short-lived JWT access tokens stored in memory and rotate them using refresh tokens. **Never store JWTs or sensitive tokens in plaintext storage** on the device; on iOS, use the Keychain to securely store credentials or refresh tokens [8]. Example mobile practice: store an OAuth refresh token in iOS Keychain and keep the access token in-memory – if the app is closed, the token is gone, and the refresh token in Keychain can be used (with user re-auth if needed) to get a new one [9].

OAuth Security Best Practices: Follow financial-grade security profiles (such as FAPI) for Open Banking integrations [10] [11]. This includes requiring PKCE for auth flows, using state parameters to prevent CSRF, and verifying JWT signatures for ID/Access tokens from providers. Where possible, use **mutual TLS** or signed requests when calling bank APIs to prevent phishing and ensure our app is an identified client. Regularly rotate client secrets for OAuth apps and use separate OAuth credentials per environment (dev, prod). Implement token revocation handling – if a user disconnects an integration, securely delete the stored tokens and revoke them at the provider.

2. Data Protection & Privacy

Encryption of Sensitive Data: All sensitive customer data should be encrypted both in transit and at rest. Enable TLS 1.2/1.3 for **all network connections** (between client and server, and server to any third-party APIs) to protect data in transit [12] [13]. For data at rest, utilize database encryption (e.g. transparent disk encryption for MySQL/Postgres, encrypted volumes for VMs) and **field-level encryption** for particularly sensitive fields. This means encrypting data like account numbers, bank transaction details, payment card PANs, or personal identifiers at the application layer using strong algorithms (AES-256-GCM for symmetric encryption) [14]. As discussed, use KMS-managed keys or an **encryption SDK** to perform envelope encryption so that each record/field can have a unique data key encrypted by a master key [15] [16]. For example, a bank account number or tax ID can be encrypted on write, and decrypted only when needed in memory – the database will only see ciphertext. Proper key management is crucial: restrict which services or roles can decrypt keys (e.g., only backend server IAM role can call KMS.decrypt) [14]. Also consider **database field-level encryption features** if using a supporting DB (e.g. MongoDB's client-side Field Level Encryption) to ensure even DB admins cannot see plaintext [17].

Tokenization and Masking: For payment card data, avoid storing raw PANs if possible – use Stripe or Mercado Pago tokens for cards. If you must handle credit card numbers, store only a **tokenized value or last-4 digits** for display, and let a PCI-compliant provider handle the full PAN. Implement data **masking** in the application for any sensitive data displayed: e.g. show only the last 4 digits of an account or card number, mask SSN/tax IDs as XXX-XX-1234, and do not reveal full values on UI or logs [18]. This ensures that even authorized views minimize exposure of PII.

PII Minimization & Anonymization: Align with **privacy by design** principles (required by LGPD/GDPR). Collect and retain only data that is necessary for the app's function, and document all personal data fields in a data inventory. Implement processes to **anonymize or pseudonymize data** where full precision is not needed. For instance, for analytics or error logging, strip or hash any personal identifiers (names, emails) so that individuals are not identifiable in analytics datasets. If sharing data with third-party analytics or bug tracking services, **scrub out PII** beforehand. Additionally, use aggregate reporting for any business metrics so individual user data isn't exposed.

Secure Logging Practices: Logging is important for audit and troubleshooting, but must be done securely. **Never log sensitive personal data or secrets in plaintext**. This includes: passwords (never log), auth tokens, full credit card numbers, CVV, or personal identifiers like full CPF (Brazilian ID) or SSN. Where logging user events, use unique user IDs or hashes instead of names or emails to protect identity (you can map IDs back internally if needed). If an error must include some data, mask it (e.g., `****1234` for an account number). **Audit logs** (security-focused logs) should record key events – logins, logout, password changes, MFA challenges, data exports, administrative actions, etc. – but again without sensitive content. Ensure audit logs capture who did what and when, with timestamps, user/account IDs, action types, and success/failure status [19]. All logs should include a synchronized timestamp (consider using ISO 8601 in UTC and synchronize server clocks via NTP) [20].

Centralized Log Management: Transmit application logs securely to a centralized logging system or SIEM (Security Information and Event Management). Cloud environments provide services like AWS CloudWatch/CloudTrail, GCP Cloud Logging, or Azure Monitor, which can store logs in a tamper-resistant manner. **Restrict access** to logs – only authorized devops or security personnel should be able to read them, since they may contain sensitive contextual info. Implement log retention policies that meet compliance needs: PCI DSS requires retaining audit trail logs for at least 1 year (with 3+ months immediately available) [20]. Brazilian regulations (e.g. AML law) similarly require retention of transaction records for 5 years or more [21]. Ensure old logs are archived securely (encrypted backups) and disposed of after the retention period expires.

Monitoring & Alerting: Set up real-time alerts on security logs for suspicious events (multiple failed logins, sudden role changes, data export requests, etc.). Deploy an IDS/IPS or cloud WAF to monitor for injection attacks or anomalous traffic. A SIEM solution can correlate logs to detect account takeover, data exfiltration attempts, or other threats [22]. All access to sensitive data in the app (e.g. viewing a bank account balance) should ideally generate an audit log entry – this can help in detecting unauthorized access and also is needed for compliance (user access reports).

Data Backup and Recovery: Maintain encrypted backups of critical data (databases, audit logs) in case of data loss or ransomware. Backups should also be protected under encryption (use KMS for backup encryption keys). Test restoration regularly as part of the incident response plan. From a compliance standpoint, ensure backup retention aligns with data retention policies – e.g., if a user invokes the right to deletion, you may need to delete or encrypt their data in backups as well, or have a process to exclude it when restoring.

3. Compliance Requirements (LGPD, PCI DSS, Tax)

This section outlines how to meet relevant compliance standards, including Brazil's LGPD data privacy law, Payment Card Industry (PCI DSS) requirements for handling card data, and Brazilian tax and financial regulations.

LGPD: Brazilian Data Protection Compliance

Brazil's **Lei Geral de Proteção de Dados (LGPD)** is a comprehensive data protection law similar to the EU's GDPR. To comply, our app must implement mechanisms to honor user data rights and protect personal data [23] [24] :

- **User Data Rights:** Implement self-service or support processes for the *data subject rights* defined in LGPD Article 18. Users have the right to confirm whether their data is processed, access a copy of their data, correct inaccurate data, and **delete personal data** that was processed with consent [24] . They also have the right to **data portability** – allow users to export their data in a commonly used format (e.g. CSV or JSON) so they can move to a competitor [24] [25] . The app should provide an "Export My Data" feature (e.g., emailing the user a ZIP of their transactions, accounts, receipts) and a "Delete My Account" feature that removes or anonymizes their personal data. When deleting, ensure all backups, logs, and integrations either delete the user's data or at least disassociate it (e.g., replace user ID with an anonymous ID in logs) within a reasonable time.

- **Consent & Purpose Limitation:** Clearly disclose and obtain consent for data processing that isn't strictly necessary for service. For example, if the app uses personal data for marketing or shares data with third parties, explicit informed consent (opt-in) is required [26] [27] . Maintain a record of consents given by users (what they consented to and when) and allow them to revoke consent at any time [28] . If a user revokes consent or denies certain data access, ensure the app can still function in a limited mode or inform the user of any features that will be disabled (transparency about consequences of denying consent is also an LGPD requirement) [28] .

- **Data Minimization & Access Control:** Under LGPD, personal data processing must be adequate and limited to the needs of the purpose. This means avoid over-collecting data (don't request unnecessary PII). Enforce strict **role-based access control** internally so that only authorized staff or services can access personal data (for example, customer support can see user profile but maybe not full bank account numbers). This ties into LGPD's principle of *security and prevention*: adopt technical measures to protect data from unauthorized access, alteration, or loss [23] . Techniques already discussed – encryption, access control, anonymization – all support compliance here. Also, implement the **principle of least privilege** for any internal accounts: employees should only access data absolutely required for their job.

- **DPO and Governance:** LGPD recommends (and effectively requires in most cases) appointing a **Data Protection Officer (DPO)** [29] . The DPO will be responsible for guiding compliance, training the team on privacy, and serving as a point of contact with Brazil's National Data Protection Authority (ANPD). We should designate a DPO and publish their contact (e.g. in the privacy policy) [30] . The DPO should also maintain a **record of processing activities** documenting what personal data we store, for what purpose, retention time, and who it's shared with [31] . This documentation helps during compliance audits or if users inquire.

- **Breach Notification:** Prepare a procedure for data breach response in line with LGPD. While LGPD does not mandate a specific 72-hour window like GDPR, it requires that breaches be reported to the ANPD and affected users in a "reasonable" time frame [32]. Our incident response plan (see Governance section) should include notifying the ANPD and users with details and mitigation steps if a breach of personal data occurs. We should also have templates for such notifications in Portuguese, as the authorities and users are local.

- **Penalties:** Non-compliance can lead to hefty fines up to 2% of Brazilian revenue (max 50 million BRL per infraction) [33]. Thus, compliance isn't just legal checkbox but critical to avoid business-ending fines. Regularly audit our practices against LGPD requirements and consider an external assessment or certification for privacy (if available).

*(Nota Bene: Many LGPD terms align with GDPR. For example, users' "right to be forgotten" (*direito à eliminação*) and "right to data portability" (*direito à portabilidade*) are explicitly in LGPD [24]. All privacy policies, consent forms, and user-facing content should be available in Brazilian Portuguese to ensure clarity to local users.)*

PCI DSS: Cardholder Data Security

If the app handles payment card information (e.g., users linking credit cards or processing payments), it **must adhere to PCI DSS** (Payment Card Industry Data Security Standard) requirements. The simplest approach is to **minimize direct handling of card data** by using tokenization services: for instance, when integrating Stripe or Mercado Pago, use client-side tokenization (Stripe Elements or MercadoPago SDK) so that card data is sent straight to the provider and we receive a token. This keeps our systems out of PCI scope for storing card numbers.

If any cardholder data is stored or transmitted by our servers, the following PCI controls are essential:

- **Protect Cardholder Data:** Do not store sensitive authentication data (full magnetic stripe, card verification code/CVV, or PINs) after authorization. If storing Primary Account Numbers (PANs), **encrypt them with strong cryptography** (AES-256) and keep keys in a secure KMS or Hardware Security Module [14]. Store only the minimal info needed – for example, the app might store the last4 and cardholder name for display and use a token for actual charges. Follow PCI DSS Requirement 3: maintain a detailed inventory of where card data resides and ensure all such data is encrypted, masked when displayed (show at most first 6/last 4 digits), and retained only as long as necessary [34] [35]. Encryption keys must be managed with dual control and split knowledge (no single person can access plaintext keys), and rotated periodically [2].

- **Secure Network and Systems:** Follow standard PCI network security practices: use a **firewall** to segment any cardholder data environment from the rest of the network, change default passwords on any network or server devices, and disable unnecessary services. Our cloud infrastructure should isolate the database or microservices that handle financial data in private subnets. Apply the latest security patches to all systems (PCI Requirement 6) – keep servers and libraries up to date, and use vulnerability scanning tools. Deploy anti-malware/EDR on servers (PCI Requirement 5) to detect any malware, though if using a cloud serverless stack this may not apply directly [36].

- **Access Control:** Enforce strict access control to systems and data (PCI Requirements 7 and 8). Only developers/admins with a legitimate need should be able to access the production database or

decrypt card data, and this access should require MFA. Every individual with access gets a unique ID – no shared accounts [37]. Use role-based permissions so, for example, a customer support role cannot query raw payment details. Administrators of the cardholder environment must use multi-factor auth for login (PCI requires 2FA for admin access to card data systems) [37]. Physical access to servers (if any) should also be controlled (if cloud, ensure cloud provider meets physical security standards) [38].

- **Monitoring and Testing:** PCI Requirement 10 and 11 mandate robust logging and monitoring. Ensure that **all access to cardholder data and system components is logged** – this overlaps with our audit logging. Logs should record user IDs, timestamps, event types, and be kept for at least 1 year with regular review [19]. Set up automated alerts for critical events (e.g., admin login, changes to configuration) to enable prompt detection of potential compromise. Conduct quarterly vulnerability scans and at least annual penetration tests on the application and network (or after significant changes) [39]. Use intrusion detection/prevention systems to monitor for malicious traffic in the cloud environment [40].

- **PCI Compliance Processes:** Maintain a written information security policy addressing all PCI controls (Requirement 12) and train employees on security procedures [41]. If our volume of cards is low, we might qualify to do an annual self-assessment questionnaire (SAQ) to attest PCI compliance; larger volume or if we store card data might require a yearly on-site audit by a Qualified Security Assessor. We should prepare a **PCI compliance checklist** (see summary checklist below) and update it as PCI DSS evolves (e.g., PCI DSS 4.0 brings new requirements around continuous testing and phishing training, etc.). Non-compliance can lead to fines or loss of ability to process cards [42] [43], so even if using Stripe (which offloads most compliance), we ensure any part we handle (like displaying the last4 or storing customer info) is done securely.

(Note: If possible, avoid storing any card data by relying entirely on third-party processors' tokens. For instance, Stripe can store customer cards and return a `customer_id` *and* `card_id` *that our app uses for charges. This way, the most sensitive data never touches our servers, greatly reducing PCI scope. We'd still enforce strong security but the risk is reduced.)*

Tax and Financial Regulations (Brazil)

For a personal/business finance app in Brazil, compliance with tax-related regulations focuses on proper record-keeping, reporting capabilities, and supporting users' tax needs:

- **Record Keeping & Retention:** Financial records (transactions, invoices, receipts) must be retained for audit and tax purposes. Brazilian law typically requires companies to keep books and records for **5 years** (the statute of limitations for tax audits) [44] [21]. Our system should not automatically purge financial transaction data before 5 years by default for business accounts. Even for personal users, providing at least 5 years of accessible history is beneficial for tax prep and is aligned with regulatory norms. If the user deletes their account (and thus invokes privacy rights), we may need to exempt financial transaction records that are required to be kept by law – in such cases, we would anonymize them (so they are not directly tied to the user, but can be produced to authorities if needed). This should be explained in the privacy policy (e.g., "if you request deletion, we will remove all personal data except information we are legally obligated to retain for compliance, which we will retain solely for that purpose").

- **Tax Reporting Features:** The application should provide features to assist with **tax compliance reporting** for users:

- For **personal users**, the app can generate an *annual financial summary* to help with Brazil's *Imposto de Renda* (income tax declaration). This might include totals of different income categories, bank interest earned, investment profits, etc., which users need to report. Ensure the app categorizes transactions in a way that aligns with tax categories (e.g., distinguishing salary income vs. business income vs. investment income) and allow exporting these categorized reports.
- For **business users**, support basic accounting outputs: profit/loss statements, expense reports, and if relevant, support the generation of **Notas Fiscais** (invoices) for their transactions or integration with invoicing systems. For example, if the app is used to record sales, it should either integrate with a fiscal invoicing system or store the necessary info for the user's accountant to issue invoices. While our app itself might not be an official bookkeeping system, it should not impede the business from complying with Brazil's bookkeeping mandates (SPED). We can facilitate by allowing data export in formats that accounting software can import, and by preserving all necessary fields (date, amount, customer info, tax category, etc.) for each transaction.

- Ensure **currency and formatting** adhere to Brazilian standards when generating reports (values in BRL, using Portuguese formatting in any documents). For legal documents or data interchange, use Portuguese language where required (e.g., if providing an official report that might be given to authorities or accountants).

- **Compliance with Tax Authorities:** If our app directly handles any regulated financial activity (like managing investments or payments), we must comply with any reporting to regulators. For instance, Brazilian financial institutions must report certain large transactions to COAF (financial intelligence unit) as part of AML rules – while our app as a personal finance tool may not be directly obligated, if we facilitate transactions above certain thresholds, we should be aware of **AML/KYC obligations**. Law 9.613 (AML law) requires preserving KYC data and transaction records for at least 5 years after the relationship ends [21] and reporting any suspicious activity or large cash transactions ($\geq$ R$50k) to COAF within 24 hours [45] [46]. Our app likely won't handle cash, but if it moves money (say via Mercado Pago), ensure the payment provider is doing necessary reporting. Nonetheless, implementing fraud detection and allowing users to flag suspicious account activity is a good practice.

- Also, if offering business financial management, consider integration or partnership with accountants or compliance services so that things like **SPED ECD/ECF** (Brazil's digital bookkeeping submissions) can be more easily prepared. While this may be beyond our app's immediate scope, designing the data schema to capture needed information (like tax classifications for each transaction) will future-proof for such features.

- If the app expands to handle payroll or invoices, ensure compliance with eSocial (labor taxes) and NFe (electronic invoice) systems by design – typically by using certified third-party components for those.

- **Localization & Legal:** Ensure that the app's terms of service and privacy policy address local Brazilian requirements (in Portuguese). For example, inform users about data processing purposes in accordance with LGPD, and any tax-related disclaimer (e.g., "this app is not a substitute for professional accounting advice" to clarify we assist but ultimate tax filing is the user's responsibility).

Provide the ability for users to collect all data needed to substantiate their tax filings (download receipts images, categorize expenses as deductible or not, etc.).

- **Audit and Regulatory Requests:** Have a procedure to handle requests from authorities. For example, if Receita Federal or a court requests a user's financial records (with proper legal process), the data should be retrievable in a comprehensible format. The system should be able to generate an **audit report** for a given user or account – listing all transactions in a date range, any edits or deletions of data, and who accessed the data. This goes hand-in-hand with our audit logging schema. Only authorized senior personnel should execute such data extractions, and each such action should be logged and reviewed (to prevent abuse).

Compliance Checklist (LGPD, PCI, Tax): *Below is a summary checklist of key compliance controls to implement:*

- **LGPD Compliance:**
- Appoint DPO and document data processing activities [29].
- Obtain and record user consent for data sharing or sensitive processing [27].
- Provide user data rights interface: data access/export and deletion (with verification) [24].
- Implement data minimization and privacy by design (encrypt and restrict PII) [23].
- Establish breach response plan including ANPD notification [32].
- Publish privacy policy in Portuguese, detailing data use and rights.

- **PCI DSS Controls:**

- Use strong encryption for any stored card data; ideally tokenize instead [18].
- Restrict access to card data (need-to-know, unique IDs, MFA) [47] [37].
- Secure network: firewalls, no default passwords, regular patches.
- Regularly scan and penetration test the app and infrastructure [48] [39].
- Log and monitor access to card data systems; keep logs ≥12 months [19].
- Maintain security policies, developer training, and incident response for cardholder environment [41] [49].

- **Tax/Financial Compliance:**

- Retain financial records for ≥5 years (securely archived) [21].
- Enable data export of transactions and reports for tax filing (BRL currency, local format).
- Support categorizing transactions with tax-relevant labels (e.g., business expense, income type).
- For business accounts: allow recording invoice numbers or tax IDs for clients to aid official bookkeeping.
- If facilitating transactions, ensure receipts/invoices can be generated or attached.
- Implement fraud detection and, if applicable, report suspicious activities in line with AML laws [45].
- Ensure compliance updates: Stay updated on regulatory changes (tax laws, privacy regulations) and update the app/policies accordingly (e.g., if Brazil mandates new open finance standards or invoicing rules, integrate them).

4. Threat Modeling and Mitigations

Proactive threat modeling is used to identify potential attacks and plan mitigations. We analyze threats across the system's architecture, covering client apps, backend, external integrations, and data stores. The diagram below illustrates a high-level architecture with key security components and some threat vectors:

Figure: High-level architecture and threat model of the financial app. The client (web browser or iOS app) communicates over HTTPS to the backend server, which interacts with an encrypted database, a Key Management Service for encryption keys, and external financial APIs (Open Finance banks, Stripe/MercadoPago) via OAuth 2.0. Potential attack vectors (in red) include phishing or malware targeting the client, injection or DDoS attacks on the server, etc. Defense-in-depth measures (TLS, encryption, WAF, throttling, etc.) mitigate these threats.

In threat modeling, we consider STRIDE categories – **Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege** [50] – and design controls for each:

- **Spoofing & Phishing (Impersonation):** An attacker might spoof user identity or trick users into exposing credentials. Mitigations: use strong authentication (MFA) and vigilant session handling as described [4]. Implement anti-phishing measures such as email signing for any communications (so users can verify official emails), and educate users (e.g., warnings in app) to only enter credentials on the official domain (consider using domain enforcement like HSTS preload for your domain to prevent impostor sites). For OAuth flows, register redirect URLs strictly and use PKCE to prevent interception. On the server side, use **OAuth state** parameters to prevent CSRF in the login flow. Also, **monitor for credential stuffing** attacks (many login attempts with different creds) and auto-lock accounts on too many failures [1]. For **spoofing of our server** by an attacker (MITM), use TLS everywhere and pin certificates in the mobile app if possible [51]. The external API calls to banks should use mutual TLS or signed JWT assertions if the Open Finance standard requires, ensuring the bank's API only accepts our app's calls (prevents a man-in-the-middle altering API requests).

- **Tampering (Data Integrity):** This includes attacks like intercepting and altering data in transit or injecting malicious inputs (e.g., SQL injection). Mitigations: End-to-end encryption (TLS) to prevent transit tampering [12]. On the client, use integrity checks such as subresource integrity for CDN scripts to prevent tampering of libraries. On the backend, defend against injection by using **parameterized queries/ORM** for all database access [48]. Employ input validation on all APIs – any data coming from clients or third-party APIs should be treated as untrusted. Use server-side filtering and validation rules for financial fields (amounts should be numeric and within expected ranges, dates valid, etc.). A Web Application Firewall (WAF) or library can detect and block common injection patterns. Also implement **message integrity** in critical workflows: for instance, if our server sends instructions to a third-party (like a payment request to Stripe), use their SDKs which include signing or check returned payload signatures to ensure they weren't forged. The system can also use HMACs or digital signatures internally for any data passed between microservices to detect tampering.

- **Repudiation (Action Denial):** A user or attacker might deny performing an action (like "I didn't make that $500 transfer"). Mitigation: maintain comprehensive **audit trails** of user actions (with timestamps, origin IP/device, etc.) [52]. Ensure that important actions (login, changing an email, initiating a payment) are logged with enough context to prove later that it was requested by the authenticated user. Use measures like email confirmations or 4-eye approvals for high-value actions, so there is a non-repudiable record (e.g., two parties approved an expense). For any financial

transaction, consider generating a unique ID and possibly a digital signature or checksum that ties the transaction to the user session. While not usually exposed to the user, this can technically bind an action to a user's private key if we issue such keys (this is more advanced; at minimum, logs suffice). Implement a mechanism for users to review recent critical actions (an account statement or log of recent logins and transactions) – this improves transparency and early detection if someone tries to repudiate a legitimate action or conversely if an attacker did something, the user can spot it.

- **Information Disclosure (Data Breach):** This threat includes accidental or malicious leakage of sensitive data. We mitigate this through **access control, encryption, and careful data handling**. As detailed in Data Protection, even if an attacker breaches the database, encrypted fields (accounts, tokens, PII) are gibberish without KMS keys [53]. Use **least privilege** for database accounts – the app's DB user should only access necessary tables/stored procedures. Implement query-level access control: e.g., a business user from Company A cannot query data of Company B (enforce tenant isolation by scoping every query to the user's company or user ID). Protect against common web vulnerabilities that lead to data leaks: **XSS** (which could let an attacker steal user data or session if our site had a vulnerability) is countered by output encoding and Content Security Policy headers [54], and by avoiding dangerous mixes of data and code. **CSRF** (which could make a user unknowingly trigger actions) is mitigated by sameSite cookies and CSRF tokens for state-changing requests. Additionally, ensure that **error messages** and API responses do not inadvertently leak data – e.g., an error should not reveal server file paths or stack traces in production, and APIs should only return data the authenticated user is allowed to see. Regularly scan for sensitive data exposure (some tools scan source code and responses to ensure no secrets or PII are exposed).

- **Denial of Service (Availability):** The app could be flooded with traffic (DDoS) or heavy usage, disrupting service. To mitigate: employ **rate limiting and throttling** on all APIs [40]. For example, limit login attempts by IP and by account to deter brute force, limit expensive operations (like generating a yearly report) to reasonable frequencies. Use an API gateway or CDN (Cloudflare, AWS CloudFront, etc.) with built-in DDoS protection to absorb large floods of traffic. At the application level, ensure queries are optimized and have timeout limits – so an attacker cannot craft an extremely expensive report query to exhaust the database. Implement circuit breakers in microservices to prevent cascading failures under high load. We will also set up **autoscaling** in the cloud for expected load bursts, but for sudden DDoS, having a traffic scrubber/WAF is key (many cloud providers offer DDoS protection services). Distinguish between trusted and untrusted traffic – for instance, if we have a mobile app, we could issue API keys to the app and throttle unknown clients more strictly. For critical third-party integrations (bank APIs, etc.), handle gracefully if they are down or slow (maybe queue requests) so that our system overload in one part doesn't crash everything (graceful degradation).

- **Elevation of Privilege:** This is where an attacker gains higher access than they should (e.g., a regular user becoming an admin). Mitigations: **rigorous authorization checks** on every endpoint and function [55]. We cannot rely solely on the UI to hide admin features; the backend must verify roles/permissions for each request. Implement a robust RBAC system (as discussed in Governance) and use library/middleware patterns to enforce it (so that every API first checks "does user X have role Y to do this?"). Perform permission audits regularly – e.g., ensure no test admin accounts are left enabled in production, and that roles are assigned correctly. Use the principle of least privilege in code: for example, if a microservice doesn't need to write to a certain database, give it read-only credentials. **Code reviews and testing** are important to catch privilege escalation bugs (like an

endpoint that doesn't correctly validate the owner of a resource). Also monitor logs for suspicious privilege changes (if an account suddenly gains admin rights, trigger an alert). In addition, keep software dependencies updated to avoid known exploits that allow privilege escalation (e.g., stay up to date with framework security patches, since an RCE vulnerability could lead to an attacker running code as the application user).

- **Other Notable Threats:**

- *Scraping Fallback:* If the app ever considered "screen scraping" a bank website as a fallback (when no API is available), this introduces serious security risks (storing user banking passwords, potential legal issues, and being brittle to UI changes). The recommendation is **to avoid scraping in favor of official APIs**. Brazil's Open Finance initiative means most banks provide OAuth APIs – use those. If absolutely required to scrape, do **not store user banking credentials** persistently; if needed, encrypt them in memory, perform the scrape in a secure sandbox, and immediately discard the credentials. Preferably, ask the user to input credentials each time in a controlled UI and *never transmit or store those on our servers* (some apps let the user do it in their own device and just send back data). However, given the sensitivity, it's best to outsource such functionality to an aggregation service that specializes in it (they have their own robust security and user consent flows). In threat modeling, storing third-party credentials is high-risk: it makes us an attractive target. So the strategy is to eliminate that vector by design.
- *Social Engineering & Support Scams:* Attackers may try to impersonate users by tricking customer support (e.g., "I'm locked out, please reset my email"). Mitigation: have strict verification for support requests – e.g., require certain information that only the legitimate user would know, and have policies in place that support staff cannot divulge sensitive info or make major account changes without additional verification. Train support personnel about common scams and require them to log all password reset or data change requests in the audit log.
- *Supply Chain Risks:* Ensure any third-party libraries, especially for fintech (e.g., Open Banking SDKs), are kept updated and from trusted sources. Use hash verification for libraries and consider solutions like Subresource Integrity for web resources. Monitor for any security advisories of tools we integrate (Stripe SDK, etc.). Also, secure our CI/CD pipeline so attackers cannot inject malicious code into our app (use signed commits, protected build infrastructure).
- *Mobile App Hardening:* Since an iOS app is planned, implement anti-tampering in the mobile app (e.g., jailbreak detection, certificate pinning, and not hardcoding secrets in the app). The OAuth client secrets for mobile should not be in the app code – use dynamic registration or a secure backend exchange to get tokens. Assume attackers will analyze the mobile app; do not include sensitive endpoints or debug backdoors in production releases.

By regularly performing **threat modeling exercises** (e.g., before each major feature and periodically), the team can stay ahead of emerging threats. Use frameworks like STRIDE and DREAD to assess risk and prioritize fixes [56] [57]. All high-risk threats should have at least one mitigation in place (often multiple layers of defense). The goal is to achieve *defense in depth*: even if one control fails, others still protect the app [58] [59].

5. Governance, Risk & Compliance Management

Finally, a governance framework is needed to maintain security and compliance over time. This includes defining roles and permissions, incident response planning, and audit and compliance management.

Role-Based Access Control (RBAC) and Permissions

The application should implement a robust RBAC system both for end-users and internal staff/admins: - **Separation of Consumer/Business Roles:** Since the app serves both individuals and businesses, design tenant isolation and roles. For example, a *consumer user* can only access their personal data, while a *business account* may have multiple user roles: e.g., *Owner* (full access to that company's financial data), *Accountant* (read-only financial reports), *Employee* (maybe limited expense input permission). Ensure that a user belonging to Company A cannot ever access Company B's data – tenant IDs should be checked on every query (multi-tenancy security). Use object-level access control in the code (only return objects that belong to the authenticated user or their organization). - **Admin and Support Roles:** Define internal roles such as *Customer Support*, *Security Admin*, *System Administrator*. These should have access to tools or dashboards to perform specific tasks (e.g., support can assist with account recovery or view a user's settings but should **not** see plaintext sensitive data like passwords or full card numbers). Use the principle of least privilege: even admins should use separate accounts for administrative tasks vs regular use, and their actions should be limited to what's necessary. For any super-admin accounts that can override data, consider requiring MFA and maybe two-person approval (the "four-eyes principle") for dangerous actions like deleting a bunch of user data or accessing someone's financial records [60]. - **Policy Management:** Governance should establish written policies for access control: e.g., an internal policy that outlines who can elevate privileges or how new admins are approved. Perform **access reviews** periodically – e.g., every quarter, have the security team review all accounts with admin roles and confirm that those accesses are still required. Immediately revoke access for departing employees and have a checklist to ensure no orphan accounts. - **Continuous Training:** All team members, especially those with privileged access (developers, admins, support), should receive regular security training. They should be aware of social engineering tactics, safe data handling (e.g., not to copy data to personal devices), and incident reporting procedures. Developers should be trained in secure coding practices (preventing OWASP Top 10 issues, etc.), which governance can track (possibly require each developer to certify or attend an OWASP training annually).

Incident Response & Disaster Recovery

Despite best efforts, incidents can happen. An **Incident Response Plan** (IRP) should be in place and tested. Key components: - **Detection and Reporting:** Ensure everyone knows how to report a suspected security incident (internal email/Slack hotline to security, etc.). Leverage automated detection (from SIEM alerts, CloudWatch alarms, etc.) to catch anomalies. Once an incident is detected, log it in an incident tracking system with time and details. - **Incident Team & Roles:** Form an incident response team (could be a rotation of security-trained engineers). Assign roles in advance: e.g., *Incident Manager* (to coordinate), *Communications lead* (to handle internal/external comms), *Forensics* (to gather evidence from logs), *Remediation lead* (to implement fixes). Have contact info (including after-hours) for all team members. Define thresholds for engaging outside help (e.g., when to involve legal counsel, PR, law enforcement, or third-party forensic consultants). - **Containment:** Define steps to contain different incident types. For example, if an API key is leaked, immediately revoke and rotate it. If a server is compromised, isolate it (take it offline or disable network access) and switch over to backup systems if possible. If user credentials are leaked, force-reset passwords and tokens for those users and possibly suspend account actions until secure. - **Eradication and Recovery:** After containment, eradicate the threat (e.g., remove malware, patch vulnerabilities exploited) and then restore systems to normal operation. Ensure clean backups are used if data was corrupted. For instance, if a database was altered, restore from backup and use binary logs to recover transactions not affected by the breach. Throughout, maintain evidence (log files, any attacker files) in case of investigation – do not immediately wipe everything. - **Communication Plan:** This includes

internal communication (keeping leadership and the team informed) and external. Under LGPD and potentially other laws, if personal data was breached, we must notify the users and regulators. Prepare templates for breach notification in both English and Portuguese for Brazilian users, describing what happened, what data is affected, and what steps users should take (e.g., change passwords, monitor accounts) [32] . Notify the ANPD (Brazil's data authority) with the required info. If payment data was involved, notify card processors/brands according to PCI procedure. Transparent communication is key to maintaining user trust. Also, if law enforcement needs to be notified (e.g., large theft or fraud), the plan should cover when and how to do so. - **Post-Incident Review:** After resolving an incident, do a blameless post-mortem. Analyze root causes and whether detection and response were effective. Update the security controls and IR plan based on lessons learned. For example, if an incident revealed a gap in monitoring, add new log sources or alerts. If response was slow because of unclear on-call, adjust the process.

Test the incident response plan at least annually with drills. This could be a tabletop exercise simulating, say, a ransomware attack or a data breach, to ensure the team knows their roles and to identify any shortcomings in a low-stakes setting.

In addition to security incidents, have a **Disaster Recovery Plan** for non-malicious incidents (outages, data loss from bugs, etc.). Define RPO/RTO (Recovery Point/Time Objectives) for critical services and ensure backups and failover systems meet those. E.g., database failover cluster for immediate continuity, offsite backups in case of major cloud region failure, etc. Regularly test restoring backups and failing over to backup systems.

Audit and Compliance Oversight

Maintaining compliance is not a one-time task. Establish an **audit program**: - **Security Audit Logging Schema:** As part of our logging (discussed in Data Protection), we define a schema for audit logs to facilitate reviews and audits. For instance, an audit log entry might include: - `timestamp` (with timezone or in UTC), - `user_id` (or service ID if internal action), - `action` (e.g., LOGIN_SUCCESS, LOGIN_FAIL, VIEW_ACCOUNT, UPDATE_PROFILE, EXPORT_DATA), - `target` (what record or object was affected, e.g., account ID or report name), - `source_ip` or `device_id` (where the action came from), - `status` (success/fail), - `metadata` (any additional info, like user-agent for login, or number of records exported, etc.).

All such logs should be in a consistent format (JSON or CSV) and easily queryable. Using JSON with fixed field names is convenient for indexing in a SIEM. For example, a login event might log: `{"ts":"2025-09-26T16:00:00Z", "user":"u12345", "action":"LOGIN_SUCCESS", "source_ip":"200.100.X.X", "user_agent":"iOSApp/1.2", "mfa_used":true}`. A data view event might be: `{"ts":"2025-09-26T16:05:00Z", "user":"u12345", "action":"VIEW_TRANSACTION", "transaction_id":"abcde-12345", "source_ip":"200.100.X.X"}`. By logging in detail, we can answer "who accessed this data" or "what did this admin account do" during audits [52] .

- **Regular Audits and Reviews:** Internally, perform regular audits:
- **User Access Audit:** As mentioned, review user and admin access rights periodically.
- **Compliance Audit:** At least yearly, do a comprehensive review of compliance against LGPD and PCI requirements (could be a checklist walk-through or even hiring an external auditor to assess our controls). For LGPD, verify that data inventories are up to date, consents are recorded, and no

prohibited data is collected. For PCI, ensure all 12 requirement families are still in compliance (e.g., check firewall rule review happened in last 6 months, penetration test done, etc.).

- **Logging Audit:** Check that logs are being collected as expected and that we can search them. Test the integrity by verifying no logs have been tampered with (if using cloud logs, ensure access is limited and perhaps use write-once storage for critical audit logs). PCI recommends daily log review for cardholder systems [20] – in practice, have automated alerts for critical events and at least a weekly human review of logs for unusual patterns.

- **Third-Party Audit/Certifications:** Ensure that any third-party services (payment processors, cloud providers) have relevant compliance certifications (PCI DSS certification, SOC 2, ISO 27001, etc.). Our compliance checklist should track that we have up-to-date attestation of compliance from these providers, since we rely on them for certain controls (e.g., AWS's PCI Attestation for the underlying infrastructure if we store card tokens).

- **Change Management:** Implement a change management process where security and compliance impacts are considered for any major app update. For example, before deploying a new feature that collects a new type of personal data, involve the DPO or security officer to conduct a privacy impact assessment and update the data processing records. For code changes, use code review to ensure security considerations (at least one reviewer looks at security impacts). Maintain version control and documentation especially for anything related to encryption (so you don't accidentally lose ability to decrypt older data after a change).

- **Governance Meetings:** It's wise to have a security and compliance committee or at least periodic meetings (e.g., monthly) to discuss any new risks, incidents, or regulatory updates. This ensures a cycle of continuous improvement. Given Brazil's regulatory environment, staying updated with ANPD guidelines or Central Bank regulations (for financial data sharing) is important – incorporate those updates into the roadmap.

- **Continuous Monitoring and Testing:** Beyond automated monitoring, periodically engage external experts for **penetration testing** and perhaps bug bounty programs to get fresh eyes on our security. For compliance, consider annual penetration tests as required by PCI, and whenever significant infrastructure changes occur [39]. Also consider **internal drills** like pretending an insider is trying to steal data and see if our controls (logging, alerts, least privilege) would catch or stop it – this can reveal gaps, reinforcing insider threat mitigations [61].

By establishing strong governance – clear roles, rigorous processes for incident and change management, and regular audits – the application's security posture will remain strong over time and adapt to new threats or rules. This holistic approach, combining technical measures with policies and procedures, ensures the financial app can protect sensitive banking and tax data and maintain user trust while meeting all legal obligations.

Sources:

1. OWASP Cheat Sheet – Session Management best practices (secure cookies, timeouts, etc.) [5] [7]
2. Reddit / AWS discussion – KMS envelope encryption approach for database secrets [2]
3. Zonos Guide – Summary of LGPD data subject rights and obligations [24] [30]
4. Exabeam – PCI DSS 12 Requirements (encryption, access control, logging, testing) [62] [19]

5. Medium (S. Saraswat) – Banking app threat modeling (authentication threats, MFA, CAPTCHA, secure cookies) [4] [5] ; data encryption/tokenization and PCI DSS for sensitive data [14] ; mobile security (Keychain, biometrics) [8] ; logging/monitoring and insider threat controls [52] [60]
6. Persona (Fintech compliance) – Data retention and AML requirements in Brazil (5-year record keeping, reporting large transactions) [21] [45] .

---

[1] [4] [5] [6] [8] [12] [13] [14] [18] [22] [40] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] Secure System Architecture & Threat Modeling: A Banking Application Example | by Shivam Saraswat | Medium
https://cybersapien.medium.com/secure-system-architecture-threat-modeling-a-banking-application-example-6ca28eda46e3

[2] [3] [15] [16] [17] KMS to Encrypt Values in DB - Best Practice? : r/aws
https://www.reddit.com/r/aws/comments/cr94za/kms_to_encrypt_values_in_db_best_practice/

[7] [9] Session management best practices
https://stytch.com/blog/session-management-best-practices/

[10] [11] What is Financial-Grade Security? | Curity
https://curity.io/resources/learn/what-is-financial-grade/

[19] [20] [34] [35] [36] [37] [38] [39] [41] [42] [43] [47] [62] The 12 PCI DSS Requirements Explained | Exabeam
https://www.exabeam.com/explainers/pci-compliance/the-12-pci-dss-requirements-explained/

[21] [45] [46] KYC and KYB in Brazil: Compliance Guide for Fintech Companies
https://withpersona.com/blog/understanding-kyc-and-kyb-requirements-in-brazil-for-fintech

[23] [24] [25] [26] [27] [28] [29] [30] [31] [33] Understanding Brazil's LGPD - Protecting Personal Data | Zonos Docs
https://zonos.com/docs/guides/country-guides/brazil/brazil-lgpd

[32] What the Latest Data Privacy Law (LGPD) Means For Your SaaS Data
https://www.owndata.com/blog/what-lgpd-means-for-your-saas-data

[44] Bookkeeping in Brazil: Mandatory Rules for Foreign Companies
https://mybusinessbrazil.com/bookkeeping-in-brazil/