SHERLOCK

Security Review For unflat



Collaborative Audit Prepared For: **unflat**
Lead Security Expert(s): **0x69**
Raiders
Date Audited: **July 9 - July 13, 2026**

Introduction

unflat is a European savings app built on a simple idea: your savings deserve more. You deposit stablecoins and earn a stable yield generated through overcollateralized lending on Morpho Protocol, with no lockups and the freedom to withdraw anytime. It is meant to feel like a savings account: you never need to understand what happens underneath. Everything, though, stays verifiable on chain, at any moment. unflat is non custodial by design. Your funds remain in your own wallet, and we charge a fee only on the yield you actually earn, never on your capital. We are built for euro minded savers, currently running on USDC, and working toward a fully euro native experience as EURC liquidity grows.

Why this audit

Our brand promise is uncomfortable on purpose: please don't just trust us, verify what we tell you. An independent security review is that promise put into practice. We asked Sherlock to look closely at the parts of the app that handle real money: how deposits and withdrawals work, how yield and fees are calculated, how funds move between the app and the outside world, and how accounts are protected.

The goal was not a badge, but an honest external stress test, so that the people trusting unflat with their savings can do so with clear eyes.

Scope

Repository: xquo-exchange/x-yield-mobile

Audited Commit: a293673947a9a9176223af9d7a00a892612bab97

Final Commit: e0479565598e7f659d7eef13d802a32c59a2a5af

Files:

- src/constants/contracts.ts
- src/hooks/useAutoBridge.ts
- src/hooks/useBridgeBalances.ts
- src/hooks/useCoveredVault.ts
- src/hooks/useCoveredWithdraw.ts
- src/hooks/useOfframpTransfer.ts
- src/hooks/usePositions.ts
- src/hooks/useVaultApy.ts
- src/services/autoBridgeService.ts
- src/services/blockchain.ts
- src/services/bridgeBalances.ts

- src/services/bridgeExecution.ts
- src/services/bridgeHistory.ts
- src/services/bridgeService.ts
- src/services/coinbaseOfframp.ts
- src/services/coinbaseOnramp.ts
- src/services/coveredDepositIntent.ts
- src/services/coveredVaultApi.ts
- src/services/coveredVaultExecution.ts
- src/services/depositPreview.ts
- src/services/depositTracker.ts
- src/services/depositVerification.ts
- src/services/morphoApi.ts
- src/services/publicClient.ts
- src/services/rpc.ts
- src/services/secureStorage.ts
- src/services/strategyExecution.ts
- src/services/transactionBuilder.ts
- src/services/vaultRegistry.ts
- src/services/withdrawOrchestration.ts
- src/services/withdrawVerification.ts
- supabase/functions/delete-account/handler.ts
- supabase/functions/delete-account/index.ts
- supabase/functions/_shared/address.ts
- supabase/functions/_shared/auth/privy-jwks-verifier.ts
- supabase/functions/_shared/auth/resolve-privy-user-id.ts
- supabase/functions/_shared/auth/verify-wallet-ownership.ts
- supabase/functions/_shared/errors/redact-pg-error.ts
- supabase/functions/_shared/supabase-admin.ts

Final Commit Hash

e0479565598e7f659d7eef13d802a32c59a2a5af

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
2	11	13

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Covered withdraw charges the 15% fee on principal when the covered cost-basis scan fails [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/8

Summary

The covered-withdraw fee should be 15% of yield, where yield is the withdrawn value minus the covered cost basis. The covered basis is `deposited - depositedRaw`, and the raw/covered split comes from a Blockscout scan of the covered vault's bbqUSDC token (`fetchCoveredDepositTxHashes`). If that scan fails it returns an empty set, so the covered deposit is counted as raw, `depositedRaw` rises to equal `deposited`, and the covered basis becomes 0. The failure never flips the `basis` flag to `unknown` (that flag tracks only the USDC fetch), so `resolveFeeBasis` charges instead of blocking, and `computeCoveredFeeRaw` bills 15% of the whole withdrawal instead of 15% of the yield.

Vulnerability Detail

The two lookups have different failure modes. The USDC fetch tries the CDP cache first; the covered scan goes straight to Blockscout. A Blockscout outage can therefore break the covered scan while the USDC basis is still served from cache (`src/services/depositTracker.ts:460`):

```
const [transferResult, coveredTxHashes] = await Promise.all([
  fetchTokenTransfersForDeposits(address, TOKENS.USDC),   // CDP cache first
  fetchCoveredDepositTxHashes(address),                   // Blockscout direct
]);
```

On a Blockscout error `fetchBlockscoutTokenTransfers` returns an empty list with `status: 'error'` instead of throwing. `fetchCoveredDepositTxHashes` discards that `status` (it destructures only `transfers` at line 358), so the empty list produces an empty set with no covered hashes; a genuinely thrown error is also swallowed by the catch at line 407 (`depositTracker.ts:350-414`):

```
const { transfers } = await fetchBlockscoutTokenTransfers(address,
↪  config.morphoVaultAddress); // :358 drops status
// ...
} catch (error) {
  debugLog('[DepositTracker] covered tx scan failed for', config.id, error); //
  ↪  :407 swallowed
}
return hashes; // empty on failure
```

An empty set means the covered deposit's transfer is not recognized, so its amount lands in `currentDepositedRaw` (`depositTracker.ts:523-534`):

```
if (isFromWallet && isToVault) {
  currentDeposited += amount;
  const isCovered = coveredTxHashes.has((transfer.hash || '').toLowerCase());
  if (!isCovered) currentDepositedRaw += amount; // covered principal counted as raw
}
```

`basis` is set from the USDC fetch alone, so `calculateDepositedFromChain` returns `'confirmed'` and caches the bad split for the 5-minute TTL (`depositTracker.ts:559-573`). `resolveFeeBasis` only blocks `'unknown'` (`depositTracker.ts:752-766`):

```
if (result.basis === 'unknown') {
  return { action: 'block', basis: 'unknown' }; // does not fire
}
return { action: 'charge', deposited: result.deposited, depositedRaw:
↪  result.depositedRaw, basis: result.basis };
```

The screen derives the covered basis from the split, so it is 0, and passes it on (`src/screens/StrategiesScreen.tsx:1146,1177`):

```
const coveredDeposited = Math.max(0, deposited - depositedRaw); // 100 - 100 = 0
setWithdrawConfirmData({ /* ... */ coveredCostBasisUsdc: coveredPos ?
↪  coveredDeposited : undefined }); // passes 0
```

`0 !== undefined`, so `executeUnifiedWithdraw` still builds the fee leg, with a 0 basis (`src/services/withdrawOrchestration.ts:154-163`), and the fee math treats the whole withdrawal as yield (`src/services/coveredVaultExecution.ts:169-182`):

```
const dOut = new Decimal(previewedUsdcOutRaw.toString()).div(1_000_000);       //
↪  105
const dBasis = new Decimal(params.costBasisUsdcRaw.toString()).div(1_000_000); // 0
const dYield = dOut.minus(dBasis);                                             //
↪  105, not 5
const dFee = dYield.times(params.feePercent).div(100);                         //
↪  15.75
```

The fee is appended as a USDC transfer to the treasury and sent on-chain in the redeem batch (`coveredVaultExecution.ts:1191-1209`).

The USDC path handles its own failure exactly this way: it sets `basis: 'unknown'` and the withdraw is blocked (finding C-3). The covered split has no equivalent signal.

Impact

On a covered position of $100 that has grown to $105, the correct fee is 15% of the $5 yield = 0.75, andtheuserreceives 104.25. In the degraded state the fee is 15% of the full $105 =

15.75andtheuserreceives 89.25. The user loses ~15% of principal, and the difference goes to the treasury (`UNFLAT_TREASURY_ADDRESS`). The trigger is an ordinary Blockscout error on the covered-token scan, with no attacker or special input. The bad split is cached for ~5 minutes, so repeated withdraws in that window are all overcharged.

Proof of Concept

Preconditions: the wallet holds a covered ("Protected") position, and at withdraw time the covered-token Blockscout scan is degraded while the USDC basis is served from cache.

Operational reproduction:

1. Make a covered deposit of 100andletitaccrueyieldto 105.
2. Warm the USDC transfer cache so the USDC basis resolves without Blockscout, i.e. ensure `cdp_transfers_<wallet>` is populated (written by the transaction-history fetch), or keep the USDC Blockscout query healthy.
3. Degrade Blockscout for the covered vault's bbqUSDC token (`config.morphoVaultAddress`) so `fetchBlockscoutTokenTransfers` returns 5xx/429/timeout across all retries. `BLOCKSCOUT_API_URL` is `https://base.blockscout.com/api`.
4. Tap Withdraw. The confirmation sheet shows a fee equal to 15% of the full position value rather than 15% of yield.
5. Confirm. The batched redeem sends a USDC transfer to `UNFLAT_TREASURY_ADDRESS` equal to 15% of the full withdrawal.

`computeCoveredFeeRaw` is the exact point where the fee is decided. This test drives it directly with a $100 covered position worth $105, once with the correct basis and once with the collapsed basis, and asserts the resulting percentages.

```
// src/__tests__/coveredWithdrawFeeOnPrincipal.test.ts
import { computeCoveredFeeRaw } from '../services/coveredVaultExecution';

describe('covered withdraw fee on principal (covered-basis collapse)', () => {
  const USDC = 1_000_000n;                 // 1 USDC in 6-decimal raw units
  const feePercent = 15;

  const principal = 100n * USDC;           // $100 deposited
  const withdrawalValue = 105n * USDC;     // $105 previewRedeem output
  const yieldRaw = withdrawalValue - principal; // $5 yield
  const correctFee = 750_000n;             // $0.75

  it('correct basis: fee is 15% of yield ($0.75)', () => {
    const fee = computeCoveredFeeRaw(withdrawalValue, { costBasisUsdcRaw:
    ↪  principal, feePercent });
    expect(fee).toBe(correctFee);                                          // $0.75
    expect(Number(fee) / Number(yieldRaw)).toBeCloseTo(0.15, 5);           // 15%
    ↪  of yield
```

```
    expect((Number(fee) / Number(principal)) * 100).toBeCloseTo(0.75, 2);  // 0.75%
    ↪ of principal
  });

  it('collapsed basis (bug): fee is 15% of the full withdrawal ($15.75)', () => {
    const fee = computeCoveredFeeRaw(withdrawalValue, { costBasisUsdcRaw: 0n,
    ↪ feePercent });
    expect(fee).toBe(15_750_000n);                                          //
    ↪ $15.75, not $0.75
    expect((Number(fee) / Number(principal)) * 100).toBeCloseTo(15.75, 2); //
    ↪ 15.75% of principal
    expect(Number(fee) / Number(correctFee)).toBeCloseTo(21, 5);           // 21x
    ↪ the correct fee
  });
});
```

The collapsed basis charges 15.75% of principal instead of 0.75%, which is 21x the correct fee, and the difference goes to the treasury.

Code Snippet

```
// src/services/depositTracker.ts:428  (calculateDepositedFromChain)
const [transferResult, coveredTxHashes] = await Promise.all([
  fetchTokenTransfersForDeposits(address, TOKENS.USDC),
  fetchCoveredDepositTxHashes(address), // empty set on Blockscout failure
]);
// ...
if (isFromWallet && isToVault) {
  currentDeposited += amount;
  const isCovered = coveredTxHashes.has((transfer.hash || '').toLowerCase());
  if (!isCovered) currentDepositedRaw += amount; // covered principal folded into
  ↪ raw basis
}
// ...
return { deposited: currentDeposited, depositedRaw: currentDepositedRaw, /* ... */
↪ basis: 'confirmed' };
```

```
// src/services/coveredVaultExecution.ts:169  (computeCoveredFeeRaw)
const dOut = new Decimal(previewedUsdcOutRaw.toString()).div(1_000_000);
const dBasis = new Decimal(params.costBasisUsdcRaw.toString()).div(1_000_000); // 0
↪ in the bug
const dYield = dOut.minus(dBasis);                                          // =
↪ dOut
const dFee = dYield.times(params.feePercent).div(100);                      //
↪ 15% of full value
```

Tool Used

Manual Review

Recommendation

Propagate the covered-scan status into the basis so a failed covered scan is treated like a failed USDC fetch:

- Have `fetchCoveredDepositTxHashes` return `{ hashes, status }` instead of discarding the `status` it already gets from `fetchBlockscoutTokenTransfers`.
- In `calculateDepositedFromChain`, if the covered scan failed and the wallet has or may have a covered position, return `basis: 'unknown'` so `resolveFeeBasis` blocks the withdraw with the same retry prompt the USDC path uses, and do not cache the partial split.

As a backstop, treat a covered cost basis of 0 with a non-zero covered position value as untrusted and block rather than charge, in `StrategiesScreen` or `computeCoveredFeeRaw`.

Discussion

alexanderpi007

Fixed in 4be61c2598a680c2185edb0f4d0556128d94a17a (fixed together with #11 – same file/pattern).

Root cause: the covered cost basis was derived as deposited – depositedRaw and was append-only – covered deposits grew it but covered exits never decremented it (the redeem ratio returned 0 for the covered pool). On a churn-heavy wallet it inflated (e.g. covered basis $355 vs a $5 live covered position). A degraded/collapsed covered basis then feeds the fee math incorrectly, and the split inconsistency also suppressed the dashboard earnings figure.

Fix:

- Covered basis is now a first-class, per-vault, decrementing quantity. On a covered exit, coveredRatio = burnedFromCovered / outstanding covered shares (denominator is outstanding shares, not the returned pool, so a partial covered exit reduces pro-rata and never wipes the whole covered basis). Public shape is unchanged, so covered = deposited – depositedRaw stays exact for all consumers.
- Fail-closed: when the covered share ledger can't reconcile (a covered scan failure/timeout or a truncated page – a covered return with no prior stake), we return basis 'unknown' and BLOCK the withdraw with the retry prompt instead of charging 15% on principal. This directly addresses the reported ”covered withdraw charges 15% on principal when the scan fails” path.
- Fixed a SplitGuard false-fire during the empty-positions load window on both screens that evaluate it (Dashboard and ManageFunds).

Tests: churny-wallet mirror (covered $355 in / $350 out → covered basis ≈ $5, split plausible against the live snapshot), partial covered exit reduces pro-rata (not wiped), covered-ledger truncation → 'unknown'/blocked/uncached, plus a split-guard contract test. Full suite 97 suites / 1777 tests green, tsc + ESLint clean. Verified on-device.

Issue H-2: `FULL_WITHDRAWAL_THRESHOLD` resets cost basis from transfer sums, not live equity [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/11

Summary

`calculateDepositedFromChain` treats a full exit as `vaultBalance < $1`. That balance is only a running sum of wallet-to-vault and vault-to-wallet USDC transfers. It is not live Morpho share equity. After yield accrues, a large redeem of roughly the original deposit face value can push the transfer sum below $1 while a large on-chain position remains. The tracker then zeros cost basis. The next app withdraw hits the confirmed-zero path and charges no performance fee on remaining profit; and if the user re-deposits first, the mirror case instead overcharges, taxing the retained principal as yield (user fund loss).

Vulnerability Detail

In `src/services/depositTracker.ts:38-39` (also `src/services/depositTracker.ts:535-546`, `src/services/strategyExecution.ts:456-458`).

On each transfer:

- Wallet to vault increases `currentDeposited` and `vaultBalance`.
- Vault to wallet decreases `vaultBalance` only. Partial exits do not reduce cost basis.
- If `vaultBalance < FULL_WITHDRAWAL_THRESHOLD` (1.0), both bases reset to zero.

Yield never increases `vaultBalance`. Only deposits do. So a $100 deposit that grows to about $200 can redeem about $99.50 of USDC and leave about $100.50 still in vault shares. The tracker computes `100 - 99.50 = 0.50 < 1` and wipes basis.

Project unit tests treat this reset as expected behavior. Downstream, `buildWithdrawBatch` with `totalDeposited === 0` and positive current value charges no fee. Any path that produces matching vault-to-wallet USDC transfers can trigger the reset (app redeem or direct Morpho redeem).

Impact

A user who can produce vault-to-wallet USDC transfers that leave transfer-sum dust under $1 can zero cost basis while a material position remains, then withdraw through the app with no performance fee on residual yield. That is platform revenue loss (fee avoidance), not theft of another user's funds. Severity remains High for reliable fee avoidance without needing a third-party compromise: the user only needs to exit most of the deposit face value while yield has accrued (via partial redeem paths available on a non-custodial vault).

The same reset has an opposite-direction impact that is user fund loss, not just fee avoidance. If the user re-deposits after the reset, the basis becomes nonzero, which bypasses the confirmed-zero no-fee guard at `strategyExecution.ts:456`, so `buildWithdrawBatch` charges 15% on `currentValue - newBasis` and taxes the retained original principal as if it were yield. Example: deposit $1000 -> grows to $1191 -> partial withdraw $1000 (triggers the reset) -> deposit $100 -> withdraw $291 charges 15% * (291 - 100) = 28.65, versusacorrect 4.59, a ~24overchargeequalto15160 of retained principal. So the bug both under-collects (no re-deposit) and over-collects on principal (re-deposit), and the over-collect case is direct user fund loss.

Code Snippet

Tool Used

Manual Review

Recommendation

1. Reset basis only when live equity across known vaults is below the dust threshold and transfer history agrees.
2. On partial exits, reduce cost basis by the redeemed share of principal (or equivalent redeem ratio). Do not wipe basis while live equity remains material.
3. If live equity is far above the transfer-sum residual, treat the basis as unknown and block fee-bearing withdraw until recomputed.

Discussion

Ali-Kalout

Tightened the impact a bit more

Raiders0786

Thanks

alexanderpi007

Fixed in 4be61c2598a680c2185edb0f4d0556128d94a17a (fixed together with #8 – same file/pattern).

Root cause: calculateDepositedFromChain reset the cost basis from the net USDC transfer-sum residual, which yield never increases. A yield-grown position could redeem ~its face value, push the transfer sum under $1, and wrongly wipe the basis – under-charging the fee (no re-deposit) or taxing retained principal as yield after a re-deposit.

Fix:

- Cost basis is now tracked per vault in SHARE terms. Partial exits reduce the basis by the redeem ratio (sharesBurned / sharesHeldBefore), so yield-grown exits keep the retained principal's basis instead of wiping it.
- A reset now requires live Morpho equity (balanceOf → convertToAssets, the same source the withdraw path uses) to agree; if live equity contradicts the wipe, we return basis 'unknown' and block the fee-bearing withdraw (never a silent $0 fee, never a principal tax). 'unknown' is never cached.
- Fail-closed on failed/truncated share fetch, including a deposit□mint cross-check that detects missing history and returns 'unknown' rather than a mis-ratio.

Beyond the recommendation: basis is tracked per-vault (one vault's exit can't wipe another's) and the deposit□mint cross-check closes a silent over-reduction hole under a truncated page.

Tests: audit repro (deposit $1000 → $1191 → withdraw $1000 → re-deposit $100 → withdraw $291 □ fee $4.59, not $28.65), no-re-deposit residual keeps its basis, live-equity contradiction → 'unknown', equity-read failure → 'unknown', full-exit → clean reset. Full suite 97 suites / 1777 tests green, tsc + ESLint clean. Verified on-device.

Issue M-1: Bridge execution sends to unvalidated `transactionRequest.to` from LI.FI API response and approves USDC to the LI.FI diamond [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/1

Summary

`bridgeExecution.ts` sends the bridge transaction to whatever address the LI.FI quote API returns in `transactionRequest.to` without validating it is the expected LI.FI Diamond contract (`LIFI_DIAMOND_ADDRESS = 0x1231DEB6f5749EF6cE6943a275A1D3E7486F4EaE`). If the API response is compromised (MITM, DNS hijack, or LI.FI backend compromise), a malicious `to`/`data` pair could drain the wallet.

Vulnerability Detail

In `src/services/bridgeExecution.ts:184-188`, In src/services/bridgeExecution.ts:72-77 (also src/services/bridgeExecution.ts:147-171).

```
const hash = await client.sendTransaction({
  to: transactionRequest.to as `0x${string}`,
  data: transactionRequest.data as `0x${string}`,
  value: transactionRequest.value ? BigInt(transactionRequest.value) : BigInt(0),
});
```

```
 */
function buildApproveData(): `0x${string}` {
  return encodeFunctionData({
    abi: erc20Abi,
    functionName: 'approve',
    args: [LIFI_DIAMOND_ADDRESS, maxUint256],
  }) as `0x${string}`;
}
```

```
const usdcAddress = quote.action.fromToken.address;
const fromAmount = BigInt(quote.action.fromAmount);

debugLog(`[Bridge] Executing bridge: ${quote.tool}, chain ${sourceChainId}`);
debugLog(`[Bridge] Wallet: ${walletAddress}`);

try {
  // Step 1: Check allowance and approve if needed
  const chain = getBridgeChainById(sourceChainId);
  if (chain) {
    const allowance = await checkAllowance(
```

```
      usdcAddress,
      walletAddress,
      LIFI_DIAMOND_ADDRESS,
      chain.rpcUrl,
    );

    if (allowance < fromAmount) {
      debugLog(`[Bridge] Allowance ${allowance} < ${fromAmount}, approving USDC`);
      try {
        const approveData = buildApproveData();
        await client.sendTransaction({
          to: usdcAddress as `0x${string}`,
          data: approveData,
        });
```

The `LIFI_DIAMOND_ADDRESS` constant is imported and used for the approve step (line 77), but the actual bridge tx destination is never compared against it. The `transactionRequest` object is the raw JSON response from `bridgeService.getQuote()` → `lifiGet()`, which is a plain `fetch()` call to `https://li.quest/v1/quote`.

The approve call grants `maxUint256` allowance to `LIFI_DIAMOND_ADDRESS` (correct), but the bridge tx itself can target any address. The approve spender is correctly hardcoded to the diamond (not quote-controlled). The token and amount are weak. Residual allowance can remain after a bridge even when the executed call used a different destination than the diamond.:

Impact

If the LI.FI API response is tampered with (via DNS poisoning, TLS MITM on a compromised network, or a LI.FI backend breach):

- An attacker could substitute a malicious `to` address that calls `transferFrom` on the approved USDC (since approval is to the Diamond, not to `to`, this particular vector requires `to` to be the Diamond with malicious `data`).
- A malicious `data` payload sent TO the Diamond could invoke an unintended route/swap that extracts value (the Diamond is a proxy with many faceted functions).
- The `value` field is also unvalidated, a manipulated quote could request an excessive native-token value.

Precondition: the attacker must be able to modify the HTTPS response from `li.quest`. This is a trust-boundary issue - the app treats the API as fully trusted for a fund-moving call.

Code Snippet

```
export async function executeBridge(
  quote: BridgeQuote,
  client: SmartWalletClient,
): Promise<BridgeExecutionResult | null> {
  if (!client?.account?.address) {
    console.error('[Bridge] Smart wallet not available');
    return null;
  }

  const { transactionRequest } = quote;
  const sourceChainId = transactionRequest.chainId;
  const walletAddress = client.account.address;
  const usdcAddress = quote.action.fromToken.address;
  const fromAmount = BigInt(quote.action.fromAmount);

  debugLog(`[Bridge] Executing bridge: ${quote.tool}, chain ${sourceChainId}`);
  debugLog(`[Bridge] Wallet: ${walletAddress}`);

  try {
    // Step 1: Check allowance and approve if needed
    const chain = getBridgeChainById(sourceChainId);
    if (chain) {
      const allowance = await checkAllowance(
        usdcAddress,
        walletAddress,
        LIFI_DIAMOND_ADDRESS,
        chain.rpcUrl,
      );

      if (allowance < fromAmount) {
        debugLog(`[Bridge] Allowance ${allowance} < ${fromAmount}, approving USDC`);
        try {
          const approveData = buildApproveData();
          await client.sendTransaction({
            to: usdcAddress as `0x${string}`,
            data: approveData,
          });
          debugLog('[Bridge] USDC approved');
        } catch (error) {
          const msg = error instanceof Error ? error.message : String(error);
          console.error('[Bridge] Approve failed:', msg);
          return { txHash: null, error: 'approve_failed', errorMessage: msg };
        }
      } else {
        debugLog(`[Bridge] Allowance sufficient (${allowance}), skipping approve`);
      }
    }
```

```
    // Step 2: Submit bridge transaction
    const hash = await client.sendTransaction({
      to: transactionRequest.to as `0x${string}`,
      data: transactionRequest.data as `0x${string}`,
      value: transactionRequest.value ? BigInt(transactionRequest.value) :
      ↪  BigInt(0),
    });

    if (!hash || typeof hash !== 'string' || !hash.startsWith('0x')) {
      console.error('[Bridge] Invalid transaction hash:', hash);
      return { txHash: null, error: 'execution_failed', errorMessage: 'Invalid
      ↪  transaction hash' };
    }

    debugLog(`[Bridge] Transaction submitted: ${hash}`);
    return { txHash: hash as `0x${string}` };
  } catch (error) {
    const classified = classifyError(error);
    console.error(`[Bridge] Transaction failed (${classified.type}):`,
    ↪  classified.message);
    return { txHash: null, error: classified.type, errorMessage: classified.message
    ↪  };
  }
}
```

Tool Used

Manual Review

Recommendation

Before sending the bridge tx, validate:

```
if (getAddress(transactionRequest.to) !== LIFI_DIAMOND_ADDRESS) {
  throw new Error('Bridge tx target is not the LI.FI Diamond');
}
```

1. Approve exact fromAmount (or amount plus a small buffer), not maxUint256.
2. Require getAddress(fromToken) to equal the configured chain USDC before approve or send.
3. Optionally revoke allowance after a successful bridge.

Also consider bounding `transactionRequest.value` to a reasonable maximum (e.g., 0 for a USDC bridge where no native token should be needed).

Discussion

alexanderpi007

Fixed in 1a91b469. Root cause: executeBridge sent to the LI.FI quote's transactionRequest.to unvalidated, approved maxUint256 to the Diamond, and bounded neither token nor value – a tampered LI.FI HTTPS quote could move funds or drain native. Fix: assertBridgeQuoteSafe(quote, chain.usdcAddress) runs before any approve/send, throwing (→ execution_failed, no tx) if transactionRequest.to !== LIFI_DIAMOND_ADDRESS, fromToken.address !== the configured chain USDC, or transactionRequest.value !== 0. An unknown source chain is now a hard reject (was: skip approve but still send). Approve grants the exact fromAmount, not maxUint256 (removes the standing-allowance risk without an extra revoke tx). Tests: non-Diamond to / non-USDC fromToken / non-zero value each rejected with no send; approve grants exactly fromAmount; happy path still executes. Unit-tested (a tampered LI.FI quote can't be forced by hand).

Issue M-2: `delete-account` scrub writes ignore supabase-js query errors, so a failed erasure still returns success [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/4

Summary

`handleDeleteAccount` runs five wallet-keyed scrub writes inside a `try/catch`, but supabase-js query builders (`.delete()` / `.update()`) resolve with a `{ data, error }` object and do NOT throw on a Postgres/PostgREST error. The code awaits each write without inspecting `.error` and without `.throwOnError()`, so a database-layer failure on any scrub write is silently ignored. Control then proceeds to delete the Privy identity and the function returns `{ ok: true }` with HTTP 200, telling the client the account was deleted while the user's PII rows persist. This breaks the GDPR / App Store 5.1.1(v) erasure guarantee the endpoint exists to provide.

Vulnerability Detail

In `supabase/functions/delete-account/handler.ts:116-151`:

```
try {
  await deps.admin.from('push_tokens').delete().ilike('wallet_address', walletLc);
  await deps.admin.from('referral_codes').delete().ilike('wallet_address',
  ↪  walletLc);
  await deps.admin
    .from('referrals')
    .delete()
    .or(`referrer_address.ilike.${walletLc},referred_address.ilike.${walletLc}`);
  await deps.admin
    .from('moonpay_transactions')
    .update({ wallet_address: null, external_customer_id: null, ip_address: null,
    ↪  country: null })
    .or(`wallet_address.ilike.${walletLc},external_customer_id.eq.${did}`);
  await deps.admin
    .from('terms_acceptances')
    .update({ user_id: 'deleted', email: null, device_info: null })
    .eq('user_id', did);
} catch (err) {
  const { code, message } = redactPgError(err, 'db_write_failed', { log:
  ↪  console.error });
  return json({ code, message }, 500);
}
```

supabase-js (`@supabase/supabase-js@2.45.4`, imported in this handler) returns a thenable

that RESOLVES with `{ data, error, status, statusText }`. It does not reject or throw on a PostgREST/Postgres error (RLS denial, permission error, constraint violation, malformed filter, PostgREST 4xx/5xx); in 2.x it even wraps network/fetch failures into `{ error }` rather than rejecting. The only way to make these calls throw is `.throwOnError()`, which is not used. This return-value-not-exception behavior is documented in Supabase's Handling errors in supabase-js guide and the delete and update API references. Because none of the five writes destructure `const { error } = await ...` or call `.throwOnError()`, the `try/catch` at lines 116/148 catches only genuine thrown exceptions, which these calls essentially never produce. `redactPgError` (imported line 25, called only at line 149) is therefore dead code on the common error path.

The handler's own comment confirms the incorrect assumption behind the bug. Lines 153-154 state: "Delete the Privy identity LAST ... If anything above failed we never reach here, so a retry is safe." That is only true if a failed write throws. Since supabase-js resolves-with-error instead, a failed scrub write does NOT stop execution, and control does reach line 155 and the `return json({ ok: true }, 200)` at line 161.

Impact

On any database-layer failure of a scrub write, the endpoint reports success while retaining user PII:

1. A write (for example the `push_tokens` delete at line 118) fails at the DB layer and resolves `{ error }`.
2. The `await` completes normally, no exception is thrown, and the remaining writes run (and may also silently fail).
3. `deletePrivyUser` runs (line 155) and, on success, the function returns `{ ok: true }` 200 (line 161).

The client is told the account was deleted and the Privy auth identity is removed, but the user's wallet-keyed rows (`push_tokens` / `referral_codes` / `referrals`) and/or the `moonpay_transactions` and `terms_acceptances` anonymization are left in place. For an endpoint whose sole purpose is GDPR / App Store Guideline 5.1.1(v) erasure, silently returning success while the data survives is a correctness and compliance defect. It is also hard to detect, since the caller has no signal that anything failed.

Severity is Medium: the outcome is serious for a compliance-critical path, but it is conditional on a DB error actually occurring (not attacker-driven).

Precondition: any Postgres/PostgREST error on one of the five writes (a transient PostgREST 5xx, an RLS/permission change, a constraint issue, a connectivity fault). No special privileges or attacker input are required.

Code Snippet

```
// 4. Scrub / delete app data (service-role; all idempotent on re-run).
try {
```

```
  // Wallet-keyed tables -> hard delete.
  await deps.admin.from('push_tokens').delete().ilike('wallet_address', walletLc);
  await deps.admin.from('referral_codes').delete().ilike('wallet_address',
  ↪  walletLc);
  await deps.admin
    .from('referrals')
    .delete()
    .or(`referrer_address.ilike.${walletLc},referred_address.ilike.${walletLc}`);
  // ... moonpay_transactions + terms_acceptances updates (also unchecked) ...
} catch (err) {
  const { code, message } = redactPgError(err, 'db_write_failed', { log:
  ↪  console.error });
  return json({ code, message }, 500);
}

// 5. Delete the Privy identity LAST ... If anything above failed we never reach
↪  here, so a retry is safe.
const privy = await deps.deletePrivyUser(did);
if (!privy.ok) {
  console.error(`[delete-account] Privy deletion failed: status=${privy.status}`);
  return json({ code: 'privy_delete_failed' }, 502);
}

return json({ ok: true }, 200);
```

Tool Used

Manual Review

Recommendation

Inspect the resolved `error` on every write (or opt every write into throwing) so failures reach the existing `redactPgError` / 500 path before the function reports success:

```
for (const op of [
  deps.admin.from('push_tokens').delete().ilike('wallet_address', walletLc),
  deps.admin.from('referral_codes').delete().ilike('wallet_address', walletLc),
  // ...the referrals / moonpay_transactions / terms_acceptances writes...
]) {
  const { error } = await op;
  if (error) {
    const { code, message } = redactPgError(error, 'db_write_failed', { log:
    ↪  console.error });
    return json({ code, message }, 500);
  }
}
```

Alternatively, add `.throwOnError()` to each write so the existing `try/catch` fires. Also consider wrapping the five writes in a single Postgres `SECURITY DEFINER` RPC so the scrub is atomic and one round trip, and correct the misleading comment at lines 153-154.

Discussion

alexanderpi007

Fixed in eb22b8d08eceead344e1ce759d8a27d677d58824. Root cause: delete-account's five scrub writes awaited inside one try/catch, but supabase-js resolves { data, error } instead of throwing – a DB-layer failure (RLS/permission/constraint/timeout) never hit the catch, control fell through to deletePrivyUser, and the function returned { ok:true } 200 with PII surviving, breaking the GDPR / App Store 5.1.1(v) erasure guarantee. Fix: the five writes now loop with .error inspected on each await; the first error returns 500 db_write_failed (redacted + server-logged) BEFORE deletePrivyUser, short-circuiting the rest. Outer try/catch kept for genuine throws; the misleading step-5 comment corrected. Tests: forcing a scrub write to resolve { error } → 500, no ok:true, deletePrivyUser never runs (identity preserved for safe retry); happy path still ok:true. 49 deno tests green.

Issue M-3: Empty indexer history is treated as confirmed-zero cost basis [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/12

Summary

When the USDC transfer fetch returns success with an empty list (including Blockscout `status === '0'` and message `No transactions found`), `calculateDepositedFromChain` reports `basis: 'confirmed'` and `deposited = 0`. There is no check against live vault share balances. Fetch failures correctly become `basis: 'unknown'` and block withdraw. A successful empty response does not. A profitable position can therefore withdraw at zero fee if the indexer returns empty while shares still exist.

Vulnerability Detail

In `src/services/depositTracker.ts:294-297` (also `src/services/depositTracker.ts:487-497`, `src/services/strategyExecution.ts:456-458`).

Fetch outcome	Result
Error and no cache	`basis: 'unknown'`; screen blocks withdraw
Success with empty list	`basis: 'confirmed'`, deposited 0; fee charged as zero
Success with truncated list (no pagination)	Understated deposits and under-collected fee

Blockscout maps ”No transactions found” to a successful empty result. `buildWithdrawBatch` then treats confirmed zero basis as no profit and no fee, while still redeeming live equity.

Empty-while-positioned cases include indexer lag, an empty or partial CDP cache, filtered explorer responses, and any untrusted path that returns a well-formed empty success body. The audit treats Blockscout and CDP as untrusted inputs for fee-critical values.

Impact

Platform performance fee can be under-collected when the indexer returns a successful empty history while the user still holds vault equity (indexer lag, partial cache, or a degraded third-party response). This is not direct theft of user principal. Impact is revenue integrity and requires the empty-success precondition, unlike a pure on-device basis wipe that needs no third party.

Code Snippet

Tool Used

Manual Review

Recommendation

1. Before accepting confirmed zero, check live vault equity. If equity is above dust, return `basis: 'unknown'` or recompute from a stronger source.
2. Prefer a server-side basis for fee calculation so the client is not the sole authority.
3. Paginate and integrity-check transfer history so incomplete explorer results are not treated as complete.

Discussion

alexanderpi007

Fixed in 5db5f3d3e6e5de157b8b48ff4b77cca250a8a2fb (same branch/file family as #11, #8 and #14).

Root cause: a SUCCESSFUL but empty indexer response – Blockscout status:'0' "No transactions found", or a genuinely-empty CDP cache – returned basis:'confirmed' deposited:0 with no live-equity check, in an early return BEFORE the #11 replay/live-equity guard. During indexer lag or a reindex, a profitable position with live vault shares could therefore withdraw at ZERO fee (strategyExecution treats totalDeposited===0 + positive value as no-fee). This was closed by neither #11/#8 (which guard a WIPED basis – that needs prior deposits in the replay) nor #14 (which guards a TRUNCATED fetch – this fetch is a clean, empty success), so it remained a distinct gap.

Fix:

- The empty-transfers branch now calls readLiveVaultEquityUsd(address) before returning. Live equity > FULL_WITHDRAWAL_THRESHOLD ($1) → basis:'unknown' (blocked, uncached, retry prompt); zero equity → confirmed-zero as before.
- Only runs on a SUCCESSFUL empty fetch; the status:'error' empty case still returns earlier, unchanged. Reuses the existing readLiveVaultEquityUsd + 'unknown' plumbing – no consumer signatures change.

Null-equity policy: a failed equity read → confirmed-zero (NOT block), deliberately diverging from #11's null→block. This is safe because the empty branch is an UNCACHED early return (the only saveBlockchainCache is past the replay loop, unreachable on an empty history), so a null decision leaves no cached basis behind – every later call re-runs the equity check, and a withdraw at equity-readable time independently re-evaluates to 'unknown' and blocks. Documented in a code comment and pinned by a poisoning test.

Preserves new-user UX: a brand-new $0 wallet, and an empty wallet during an RPC blip, both stay unblocked.

Tests: empty + equity $100 → 'unknown'/blocked/not-cached (exploit closed); empty + $0 equity → confirmed deposited:0 (new user works); empty + null equity → confirmed (no false block); poisoning test → a null-equity confirmed-zero leaves the cache empty and a later equity-readable call returns 'unknown'. The #11/#8/#14 suites pass unchanged. Full suite 97 suites / 1785 tests green, tsc + ESLint clean. Verified on-device (existing covered wallet unchanged at deposited $105.15 [blockchain/confirmed]; zero-equity wallet resolves to $0.00 [blockchain/confirmed] with no false block).

Issue M-4: Production withdraw never applies referral yield boost [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/13

Summary

`getEffectiveFeePercent` correctly clamps referral boost and reduces the 15% fee. Covered withdraw inherits `batch.feePercent`. Production `StrategiesScreen` always passes `undefined` as `yieldBoost` into `buildWithdrawBatch`. Users with active referrals therefore pay the full fee even when the product promises a waiver.

Vulnerability Detail

In `src/screens/StrategiesScreen.tsx:1132-1138` (also `src/services/strategyExecution.ts:390-394`).

```
const batch = buildWithdrawBatch(
  positions,
  displayAddress as `0x${string}`,
  depositedRaw,
  undefined, // yieldBoost never loaded
  true,
);
```

`getActiveYieldBoost` exists in `referralService.ts` and feeds display paths. It is not connected to the fee-bearing batch builder. Covered legs inherit the same unboosted `batch.feePercent` through the orchestrator.

Impact

Users with active referrals are overcharged relative to advertised economics (up to a full fee waiver at max boost).

Code Snippet

Tool Used

Manual Review

Recommendation

Load `getActiveYieldBoost(displayAddress)` before building the withdraw batch. Pass it into `buildWithdrawBatch` and into covered `feeParams.feePercent`. Add a test that an active referral reduces fee percent on the production path.

Discussion

alexanderpi007

Fixed in 066b184793b3f8a165cbeb7702e3cace11fe4dce (on branch fix/sherlock-13-referral-boost, stacked on the cost-basis cluster).

Root cause: StrategiesScreen.tsx handleWithdraw passed `undefined` as the yieldBoost argument into buildWithdrawBatch. getEffectiveFeePercent (strategyExecution.ts) and the covered leg were both already boost-aware, but getActiveYieldBoost was only ever loaded on display paths – never on the withdraw path. So batch.feePercent stayed at the nominal 15%, and users with an active referral paid the full fee: the advertised waiver never applied in production.

Fix:

- Load getActiveYieldBoost(displayAddress) in handleWithdraw and pass it into buildWithdrawBatch. This sets batch.feePercent = getEffectiveFeePercent(boost).
- BOTH legs derive from batch.feePercent – the raw leg internally, the covered leg via feeParams.feePercent = batch.feePercent in executeUnifiedWithdraw – so this single wiring point waives the fee on raw AND covered positions. There is no separate covered arg to set at the screen.
- getActiveYieldBoost fails safe to zero-boost on any error, so a lookup blip charges the standard 15% and never over-waives.

Tests: raw leg – existing strategyExecution.test.ts (buildWithdrawBatch boost 0.25 → 10%, 0.75 → 0%, null → 15%); covered leg – new withdrawOrchestration.test.ts asserting a boosted batch hands the covered executor the reduced feeParams.feePercent (10, not 15) and a no-boost batch → 15%. Full suite 97 suites / 1787 tests green, tsc + ESLint clean.

Verified on-device via a temporary boost stub (no live referred wallet was available) – the withdraw fee preview halved to 7.5% of yield ($0.009584 vs $0.019) on a Withdraw All across both raw and covered legs; propagation is also unit-tested end-to-end. Stub reverted; the committed fix is clean.

Issue M-5: Blockscout `tokentx` fetch is single-page and can truncate cost basis [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/14

Summary

`fetchBlockscoutTokenTransfers` issues one `account/tokentx` request (`startblock=0`, `endblock=99999999`, `sort=asc`) with no pagination. Explorers cap result size. Long USDC histories can return a truncated list that is still treated as complete (`status: 'ok'`). Cost basis then understates deposits or misses cycle resets.

Vulnerability Detail

In `src/services/depositTracker.ts:249-257`.

Other CDP transfer paths paginate. Deposit tracking falls back to Blockscout without page loops. Incomplete data is worse when treated as complete truth for fee calculation.

Impact

Power users and long-lived wallets can see systematic fee miscalculation. Missing deposits lower basis and under-collect fee. Missing withdraws break cycle reset logic.

Code Snippet

Tool Used

Manual Review

Recommendation

Paginate until an empty page or a clear completion marker. If a max-page limit is hit without completion, return `status: 'error'` rather than `ok`. Prefer full CDP pagination as the primary path, and fail closed when the page set is incomplete.

Discussion

alexanderpi007

Fixed in 7cacda15efd287ee9b2310e0f6b58c0685ae9918 (same branch/file family as #11 and #8).

Root cause: fetchBlockscoutTokenTransfers issued a single account/tokentx request (startblock=0, endblock=99999999, sort=asc) with no pagination. Explorers cap page size, so a long history returned a TRUNCATED list still treated as complete (status:'ok'). This understated deposits (under-collected fee) and, after the #11/#8 fail-closed logic landed, tripped shareDataIncomplete → basis 'unknown' → the withdraw HARD-BLOCKED permanently for long-history wallets.

Fix:

- Blockscout tokentx now paginates (page/offset, 1000 per page, 25-page cap), aggregating in sort=asc order until a SHORT page marks completion.
- Fail-closed: any page failing after retry/backoff, or the page cap hit while pages are still full, returns { transfers: [], status: 'error' } – partial data never reaches the basis replay (never partial-as-'ok'). The exported signature is unchanged, so all callers (USDC basis fetch, covered share-ledger fetch, covered-tx scan) and the #11/#8 fail-closed plumbing are unaffected.
- Also hardened a CDP-side bug found during this work: fetchFromCdp cached PARTIAL results as complete – MAX_PAGES exhaustion with a pending pageToken, and a mid-crawl page failure (a page-1 failure even cached an empty list as a confirmed-empty history). It now throws on an incomplete crawl instead of caching, so a truncated cdp_transfers_ cache can no longer be consumed as a confirmed basis (the caller falls back to the now-paginated Blockscout path).

Tests: full page + short page → 1005 transfers aggregated 'ok' in 2 calls; 25-page cap without completion → 'error'/[] (no partial-as-ok); a failing page mid-pagination → 'error' with no partial data; fresh CDP cache → zero Blockscout calls for USDC. The #11/#8 suite passes unchanged (fail-closed now fires only when pagination genuinely can't complete). Full suite 97 suites / 1781 tests green, tsc + ESLint clean. Verified on-device (677-transfer wallet still reconciles to deposited $105.15 / covered $5 [blockchain/confirmed] via the new paginated path).

Issue M-6: Covered Phase B redeems all `bbqUSDC` while fee uses covered-only basis [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/15

Summary

After a covered redeem settles, `convertCoveredToUsdc` redeems the wallet's full Morpho `bbqUSDC` balance. The fee is computed from covered-only pro-rated cost basis. Residual intentional raw `bbqUSDC` (from dust edges, a skipped raw leg, concurrent deposits, or gifted shares) is force-redeemed. If total USDC out exceeds covered basis, raw principal can be treated as yield and overcharged.

Vulnerability Detail

In `src/services/coveredVaultExecution.ts:1185-1196` (also `src/services/coveredVaultExecution.ts:1255-1264`).

Comments state that Phase B sweeps the full `bbqUSDC` balance (settled redemption plus deposit-haircut dust). Unified withdraw tries to run the raw leg first, but any raw bbq still present when Phase B runs is included. Fee uses `previewRedeem(allBbq) - coveredBasis`, so raw principal can look like covered yield.

Impact

Users can overpay the performance fee on raw principal misclassified as covered yield. Intentional raw Steakhouse positions can also be converted to USDC during a covered withdraw without that being the user's intent for that slice.

Code Snippet

Tool Used

Manual Review

Recommendation

Snapshot `bbq` balance before the redeem request. In Phase B redeem only the increase above that baseline (the settled amount). Fee only that delta against covered basis. Leave intentional raw `bbq` for the raw withdraw path.

Discussion

alexanderpi007

Fixed in d170aa86. Root cause: convertCoveredToUsdc redeemed the wallet's FULL bbqUSDC balance while the fee used the covered-only basis, so intentional raw bbqUSDC was force-redeemed and its principal taxed as covered yield; executeCoveredForceClaim had the same defect. Fix: threaded a pre-request baseline into convertCoveredToUsdc – it now redeems only currentBbq − baseline (the settled covered delta), clamped to 0 (fail-safe no-op, never a raw sweep), and previews/fees on that delta. Raw bbq is left for the raw path. Tests: 100e18 raw floor + 40e18 settled covered → redeem targets only the 40e18 delta, fee = 15% of the $3 covered yield ($0.45), not raw-inflated. Unit-tested.

Issue M-7: Coinbase onramp opens an arbitrary backend-returned URL [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/16

Summary

`getOnrampSessionUrl` posts the user's wallet to `api.unflat.app` and returns any `data.url` string. `openOnrampUrl` and `openCoinbaseOnramp` open that URL in an in-app browser with no host allowlist and no client check that the session destination equals the user's wallet. A compromised or malicious API response becomes a phishing or wrong-destination funding surface.

Vulnerability Detail

In `src/services/coinbaseOnramp.ts:60-85` (also `src/services/coinbaseOnramp.ts:104-110`).

The client has no defense-in-depth on the returned URL. Trust rests entirely in `API_BASE_URL`. From the mobile app's perspective, that backend response should still be validated before it is opened as a funding destination.

Impact

After "Buy USDC", the in-app browser can open a phishing page if the backend response is malicious or compromised. Wrong-destination funding requires a bad session binding server-side. It is the same class as other trust-boundary issues (attacker must control or forge the `api.unflat.app` response the client trusts).

Code Snippet

Tool Used

Manual Review

Recommendation

1. Allowlist Coinbase onramp hosts and path prefixes.
2. Prefer structured backend fields (`destinationAddress`, `sessionId`) and assert destination equals the user wallet before opening.
3. Reject non-HTTPS URLs and unexpected query shapes.

Discussion

Ali-Kalout

This requires the backend (`api.unflat.app`) to be compromised or to forge its response, which is a trusted first-party source with no untrusted-input path reaching it, so I believe Low/Info is a better fit for this. wdyt?

Raiders0786

Cool, can be done. As it's a full whitehat security review and this impact the users funds directly I made it ”medium” but ideally in a real world scenario this is a low. So I'll update it if that's helpful.

Ali-Kalout

> Cool, can be done. As it's a full whitehat security review and this impact the users funds directly I made it ”medium” but ideally in a real world scenario this is a low. So I'll update it if that's helpful.

Makes sense, let's keep as med for now then

alexanderpi007

Fixed in e6a452a3. Root cause: getOnrampSessionUrl returned any data.url from the backend and openOnrampUrl opened it in an in-app browser with no host allowlist and no destination check. Fix: isAllowedOnrampUrl(url, wallet?) enforces HTTPS-only, host === coinbase.com or *.coinbase.com (subdomain-anchored, lookalikes rejected), path prefix ∈ {/buy, /onramp}, and – since the backend returns no structured fields – if the URL exposes a destination address it must equal the user's wallet. Wired into getOnrampSessionUrl (invalid → null → safe fallback) and openOnrampUrl (host/scheme gate at the actual open). Tests: non-Coinbase host / lookalike host / http:// / mismatched destination all rejected; a valid Coinbase URL accepted; openOnrampUrl opens no browser for a bad URL. Unit-tested.

Issue M-8: Covered deposit sizes `UserOp-2` from balance delta when receipt events are missing [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/17

Summary

On the happy path, cover request size comes from owner-filtered Morpho `Deposit` logs on the receipt. If `mintedShares` is still null (missing event or transport-drop recovery), code falls back to `balanceNow - baselineBbq`. Concurrent mints or credits between baseline and measure inflate `requestShares` (`mintedShares + legacyDustShares`). That can requestDeposit extra bbq (pull intentional raw into cover) or produce reverts and stuck cover-pending states.

Vulnerability Detail

In `src/services/coveredVaultExecution.ts:770-793`.

```
mintedShares = balanceNow > baselineBbq ? balanceNow - baselineBbq : BigInt(0);
// ...
const requestShares = mintedShares + legacyDustShares;
```

The receipt path is sound. The balance-delta fallback is not exclusive under concurrent activity.

Impact

Cover sizing can be wrong. Intentional raw bbq can be pulled into an insured vault request. Concurrent activity can desynchronize the covered deposit state machine.

Code Snippet

Tool Used

Manual Review

Recommendation

If receipt events cannot resolve minted shares, return `covered: false` with `shares_unresolved` and do not size UserOp-2 from balance delta. Retry only after a successful receipt re-fetch.

Discussion

alexanderpi007

Fixed in 0845bfa2. Root cause: on a missing Morpho Deposit receipt event, executeCoveredDeposit sized the cover request from balanceNow – baselineBbq; a concurrent mint/credit between baseline and measure over-sizes it, pulling intentional raw bbqUSDC into the insured position. Fix: removed the balance-delta fallback. When the receipt event can't resolve minted shares → covered:false, pendingReason:'shares_unresolved', intent retained → a retry re-fetches the receipt and resolves the exact event. Never sizes from a guess (fail-closed, consistent with #8/#11/#12/#15/#19). Tests: no Deposit event + a 100e18→500e18 balance jump → covered:false/shares_unresolved, only UserOp-1 sent, intent retained; present-event happy path sizes to the exact minted shares. Unit-tested.

Issue M-9: Auto-bridge treats a LI.FI status of DONE as success without checking the substatus, so a refunded or partial bridge is reported as complete [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/18

Summary

The auto-bridge flow reports success whenever LI.FI returns `status === 'DONE'`, but LI.FI marks a bridge `DONE` for three different outcomes distinguished only by `substatus`: completed, partial (a different token was delivered), and refunded. `pollBridgeStatus` returns on the first `DONE` without inspecting `substatus`, and `executeAutoBridge` then calls `onStatusChange('done')` and persists `DONE`. A bridge that was refunded to the source chain (or delivered as a different token) is shown to the user as a completed bridge to Base, and the 30-minute success lock is left in place instead of the 5-minute failure cooldown, so the automatic retry is delayed.

Vulnerability Detail

LI.FI's status object carries both `status` and `substatus`, and the code already models `substatus` (`src/services/bridgeService.ts:69`) and logs it on failure (`src/services/bridgeExecution.ts:246`). But the completion check ignores it (`src/services/bridgeExecution.ts:240-243`):

```
if (status.status === 'DONE') {
  debugLog(`[Bridge] Bridge completed! Receiving tx: ${status.receiving?.txHash}`);
  return status;                      // returns for DONE + ANY substatus
}
```

Per the LI.FI status contract, when `status` is `DONE` the `substatus` is one of `COMPLETED` (the transfer succeeded), `PARTIAL` (the transfer completed but the user received a different token than requested), or `REFUNDED` (the transfer failed and the tokens were returned to the sender on the source chain), as also described in the API reference. `pollBridgeStatus` returns the first `DONE` regardless of which substatus it is.

The caller treats the returned `DONE` as unconditional success (`src/services/autoBridgeService.ts:178-187`):

```
if (status?.status === 'DONE') {
  onStatusChange('done');                                  // user sees "bridge
  ↪  complete"
  await updateBridgeStatus(walletAddress, bridgeId, 'DONE');
} else if (status?.status === 'FAILED') {
```

```
  onStatusChange('failed', 'Bridge transaction failed. Your funds are safe.');
  await updateBridgeStatus(walletAddress, bridgeId, 'FAILED');
} else {
  onStatusChange('bridging');
}
```

There is no branch for `DONE` with `substatus` `REFUNDED` or `PARTIAL`. Both are folded into the success branch. This also affects the dedup lock lifecycle. The lock is set to `'bridging'` (a 30-minute TTL, `DEDUP_TTL_MS`) before the bridge starts, and only the failure path downgrades it to the 5-minute `'failed'` cooldown (`FAILURE_TTL_MS`). Because a refund is routed through the success branch, the `'done'` handler in `useAutoBridge` only clears `bridgingRef` and leaves the lock at `'bridging'` (`src/hooks/useAutoBridge.ts:61-72`):

```
if (newStatus === 'done') {
  bridgingRef.current = false;            // lock stays 'bridging' => 30-min TTL
}
if (newStatus === 'failed') {
  bridgingRef.current = false;
  setAutoBridgeLock(detected.chainId, 'failed');  // downgrade to 5-min cooldown
}
```

So a refund that should have retried after ~5 minutes is instead held off for the full 30-minute success TTL.

Impact

On a `REFUNDED` bridge no funds are lost: the tokens are returned to the user's own wallet on the source chain. The two effects are (1) a false completion message, the user is told the bridge to Base completed while the money is back where it started, and (2) a delayed automatic retry, the lock stays at the 30-minute `'bridging'` TTL instead of the 5-minute `'failed'` cooldown, so the stray funds are not re-bridged for about 30 minutes rather than about 5. This does self-correct: once the 30-minute lock expires, the balance poll re-detects the source-chain funds and starts a new bridge. The impact is a false "Complete" state on a money-movement path plus a longer-than-necessary retry gap, not a loss of funds.

Proof of Concept

Preconditions: an auto-bridge is initiated (stray funds detected on a supported source chain) and the route refunds or partially fills, which LI.FI surfaces as `status: 'DONE'` with `substatus: 'REFUNDED'` or `'PARTIAL'`.

1. `executeAutoBridge` submits the bridge and calls `pollBridgeStatus`.
2. LI.FI returns `{ status: 'DONE', substatus: 'REFUNDED' }`.

3. `pollBridgeStatus` returns that object at the `status.status === 'DONE'` check without reading `substatus`.
4. `executeAutoBridge` runs the `DONE` branch: `onStatusChange('done')` and `updateBridgeStatus(..., 'DONE')`. The `'done'` handler leaves the lock at `'bridging'` (30-minute TTL).
5. The user sees a completed bridge while the funds are on the source chain, and the next auto-bridge attempt for that chain is suppressed for ~30 minutes rather than the ~5 minutes a correctly-classified failure would use.

Code Snippet

```
// src/services/bridgeExecution.ts:240
if (status.status === 'DONE') {
  return status;                // does not distinguish COMPLETED vs PARTIAL vs
  ↪  REFUNDED
}

// src/services/autoBridgeService.ts:178
if (status?.status === 'DONE') {
  onStatusChange('done');       // refunded/partial reported as complete
  await updateBridgeStatus(walletAddress, bridgeId, 'DONE');
}
```

Tool Used

Manual Review

Recommendation

Treat `DONE` as success only when the substatus confirms full delivery, and route refunded/partial outcomes to a distinct branch:

```
// pollBridgeStatus: keep returning on DONE/FAILED, but let the caller read
↪  substatus.
if (status?.status === 'DONE') {
  if (status.substatus === 'COMPLETED') {
    onStatusChange('done');
    await updateBridgeStatus(walletAddress, bridgeId, 'DONE');
  } else {
    // REFUNDED / PARTIAL: funds are (partly) back on the source chain.
    onStatusChange('refunded', 'The transfer did not complete and your funds were
    ↪  returned.');
    await updateBridgeStatus(walletAddress, bridgeId, 'REFUNDED');
    // do not set the dedup lock, so the funds can be re-bridged.
```

```
  }
} else if (status?.status === 'FAILED') {
  // ...unchanged...
}
```

Only `COMPLETED` should show success and persist `DONE`; `REFUNDED` and `PARTIAL` should show an accurate message and leave the flow free to retry (downgrade the lock to the `'failed'` cooldown so the funds are re-bridged promptly).

Discussion

alexanderpi007

Fixed in 6c83a533. Root cause: LI.FI marks status:'DONE' for COMPLETED, PARTIAL, and REFUNDED (distinguished by substatus); executeAutoBridge folded all three into the success branch – reporting a refunded/partial bridge as complete and leaving the dedup lock at the 30-min 'bridging' TTL instead of the 5-min 'failed' cooldown. Fix: DONE is success (persist DONE) only when substatus === 'COMPLETED'; REFUNDED/PARTIAL (and any ambiguous DONE) route through onStatusChange('failed', 'The transfer did not complete and your funds were returned.') and persist REFUNDED/PARTIAL. Routing through 'failed' reuses the hook's existing lock downgrade, so stray funds re-bridge in ~5 min not ~30. substatus already flowed to callers via the type. Tests: DONE+COMPLETED → success/persist DONE; DONE+REFUNDED → failed + ”returned” msg + persist REFUNDED; DONE+PARTIAL → failed + persist PARTIAL. Unit-tested (the refund path can't be forced by hand).

Issue M-10: Concurrent cross-session cover recovery can clear a fresh covered deposit's intent, leaving that deposit uninsured with no automatic recovery [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/19

Summary

The covered-deposit intent is a single per-wallet slot, and `retryCoveredDepositRequest` ends by calling `intentStore.clear()` unconditionally (a plain `removeItem`, not a compare-and-clear). The app runs this recovery in two places that are not mutually exclusive: a fire-and-forget cross-session recovery on screen mount, and a "finish lingering cover first" step inside a new covered deposit. The mount recovery is not gated before the deposit UI is enabled, so its `clear()` can land after a brand-new covered deposit has already written its own intent to the same slot. The fresh deposit's intent is erased mid-flight, and if that deposit's cover request then fails, both recovery mechanisms (in-session retry and cross-session app-open retry) have nothing to act on, so the deposit stays uninsured.

Vulnerability Detail

The intent store is one slot per wallet and `clear()` is unconditional (`src/services/coveredDepositIntent.ts:33-72`):

```
const KEY_PREFIX = 'covered_deposit_intent_v1_';
const keyFor = (wallet: Address) => KEY_PREFIX + getAddress(wallet);
// ...
async clear(): Promise<void> {
  await AsyncStorage.removeItem(key);   // erases whatever intent currently
  ↪  occupies the slot
},
```

`retryCoveredDepositRequest` clears the slot at every terminal branch, including after a successful cover request for the shares it read at entry (`src/services/coveredVaultExecution.ts:919, 930, 939`):

```
const intent = await intentStore.get();   // :883  snapshots the OLD intent
// ...guards...
await confirmTx(requestTxHash);
await intentStore.clear();                // :939  clears the slot, regardless of
↪  what is in it now
```

The screen invokes this recovery in two paths that can overlap:

1. Mount / app-open cross-session recovery, fire-and-forget and not awaited before deposits are allowed (`src/screens/StrategiesScreen.tsx:487-498`):

```
const coverRecoveryAttempted = useRef(false);
useEffect(() => {
  if (coverRecoveryAttempted.current || !smartWalletClient || !displayAddress)
  ↪  return;
  coverRecoveryAttempted.current = true;
  void (async () => {
    const result = await retryCoveredDepositRequest({ client: smartWalletClient,
    ↪  config: STEAKHOUSE_COVERED, intentStore: makeCoveredIntentStore(wallet) });
    // ...
  })();
}, [/* ... */]);
```

1. Inside a new covered deposit, which then writes a fresh intent via `executeCoveredDeposit` (`src/screens/StrategiesScreen.tsx:867-896` and `coveredVaultExecution.ts:736`):

```
// Finish any lingering pending cover from a previous deposit FIRST ...
await retryCoveredDepositRequest({ client, config: STEAKHOUSE_COVERED, intentStore
↪  });
// ...
const result = await executeCoveredDeposit({ /* ... */ intentStore }); //
↪  executeCoveredDeposit -> intentStore.save({...})
```

The in-deposit comment at line 867 shows the authors know the slot must not be overwritten while a cover is live, and their mitigation (run the retry before the new deposit) handles the sequential case. It does not handle the concurrent case: path 1 was started earlier, on mount, and is still doing its network round-trips (`snapshotCoveredPending`, `getMorphoShareBalance`, `sendCalls`, `confirmTx`) when path 2 runs `save()` for the new deposit. When path 1 finally reaches `clear()`, it deletes the new deposit's intent, because `clear()` does not check that the slot still holds the intent path 1 was working on.

Once the fresh intent is cleared, the new deposit cannot re-persist it either. `executeCoveredDeposit` continues and calls `intentStore.update({ depositTxHash })` and `update({ mintedShares })`, but `update` is a read-modify-write that bails when the slot is empty (`src/services/coveredDepositIntent.ts:47-54`):

```
async update(patch: Partial<CoveredDepositIntentRecord>): Promise<void> {
  const current = await AsyncStorage.getItem(key);
  if (!current) return;   // slot was cleared -> update is a silent no-op
  await AsyncStorage.setItem(key, JSON.stringify({ ...JSON.parse(current), ...patch
  ↪  }));
},
```

So after the clobber, every subsequent update for that deposit silently no-ops and the intent stays gone for the rest of the deposit's life.

Impact

No funds are ever at risk. The new deposit's bbqUSDC shares are minted, earning, and fully withdrawable (swept and redeemed on the next covered withdraw, per `coveredVaultExecution.ts:906-907`). What is lost is the insurance: a deposit the user chose as "Protected" can silently end up as uninsured raw bbqUSDC, with both intent-based recovery paths (the in-session retry button and the next app-open recovery) defeated because they read a now-empty slot. It does not self-heal, but it is not strictly permanent either: a later covered deposit's legacy-dust folding can re-insure the orphaned shares.

Proof of Concept

Preconditions: a lingering covered-deposit intent exists at app open (a previous covered deposit died between its two UserOps).

1. Open the Strategies screen. The mount effect fires `retryCoveredDepositRequest` (path 1) for the lingering intent; it begins its on-chain reads and does not block the UI.
2. While path 1 is in flight, start a new covered deposit. Path 2 runs its own `retryCoveredDepositRequest`, then `executeCoveredDeposit`, which calls `intentStore.save(newIntent)` before UserOp 1.
3. Path 1 completes its work for the OLD shares and calls `intentStore.clear()`, deleting `newIntent`. The new deposit's later `update()` calls then no-op (empty slot), so the intent stays gone.
4. The new deposit's UserOp 2 (cover request) fails or is interrupted. There is no intent to drive in-session or cross-session recovery, so the deposit is left as uninsured bbqUSDC. The funds remain visible and withdrawable, but the position is not covered until a later covered deposit happens to fold it back in.

Code Snippet

```
// src/services/coveredDepositIntent.ts:65  (unconditional clear - root cause)
async clear() { await AsyncStorage.removeItem(key); }

// src/services/coveredVaultExecution.ts:883 + :939
const intent = await intentStore.get();   // reads OLD intent
// ...
await intentStore.clear();                // clears whatever is in the slot NOW
↪  (possibly a newer intent)
```

Tool Used

Manual Review

Recommendation

Make the clear a compare-and-clear so a recovery only removes the intent it was actually working on:

```
// CoveredIntentStore
async clearIf(predicate: (i: CoveredDepositIntentRecord) => boolean): Promise<void>
 ↪  {
  const current = await this.get();
  if (current && predicate(current)) await AsyncStorage.removeItem(key);
}
```

Give each intent a stable id (for example the UserOp-1 tx hash or a created-at token) and have `retryCoveredDepositRequest` capture that id at entry and call `clearIf(i => i.id === capturedId)` instead of `clear()`. As a lighter alternative, gate the deposit action until the mount recovery has settled (await the mount `retryCoveredDepositRequest` before enabling covered deposits) so the two paths cannot overlap.

Discussion

alexanderpi007

Fixed in a2e6a20c. Root cause: the per-wallet covered-deposit intent slot used an unconditional clear(); the mount/app-open recovery and the in-deposit finish-lingering step overlap, so a recovery holding an OLD intent snapshot could clear() a fresh intent a new deposit had just saved – leaving that deposit uninsured, with later update() calls no-op'ing on the empty slot. Fix: compare-and-clear – each intent carries a stable id; retryCoveredDepositRequest captures it at entry and calls clearIf(i => i.id === capturedId) for all terminal clears (incl. executeCoveredDeposit's own). Backstop: the new deposit awaits the mount recovery before saving, bounded by Promise.race(8s) so a stuck recovery can't wedge deposits. Tests: a stale recovery clearIf() no-ops after a new deposit saved a fresh intent; the fresh intent survives and update() persists. Unit-tested (the cross-session race can't be forced by hand).

Issue M-11: Auto-bridge auto-executes with no minimum-received floor and a flat $1 trigger on every chain [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/21

Summary

The auto-bridge flow moves any stray USDC balance of $1 or more from a supported chain to Base automatically, with no user confirmation. It fetches a single LI.FI quote and executes it, and the only gate is whether a quote came back. It never checks that the amount that will land on Base is economically reasonable relative to the amount leaving the source chain: there is no minimum-received floor, no fee ceiling, and no slippage bound passed to LI.FI. The trigger threshold is a flat $1 for every supported chain, including Ethereum mainnet, where bridge and gas costs routinely dwarf a $1 principal.

Vulnerability Detail

The trigger threshold is a flat $1 in each token's native decimals, applied to every chain (`src/services/autoBridgeService.ts:7-8, :80-81`):

```
export function getMinBridgeAmount(usdcDecimals: number): bigint {
  return 10n ** BigInt(usdcDecimals); // $1, same for every chain
}
// detectBridgeableBalance:
if (balance < getMinBridgeAmount(chain.usdcDecimals)) continue;
```

Ethereum mainnet is a first-class supported chain with the same $1 gate as the cheap L2s (`src/constants/bridge.ts:47-52`):

```
{ name: 'Ethereum', chainId: 1, usdcDecimals: 6, /* ... */ }
```

`executeAutoBridge` fetches one quote and executes it, with `if (!quote)` as the only guard. It never inspects `quote.estimate.toAmount`, `quote.estimate.toAmountMin`, or `quote.estimate.feeCosts`, and enforces no minimum-received (`src/services/autoBridgeService.ts:119-134`):

```
const amountRaw = Number(detected.balance);
onStatusChange('quoting');
const quote: BridgeQuote | null = await getQuote(detected.chainId, amountRaw,
↪  walletAddress);
if (!quote) {                                    // the only check
  onStatusChange('failed', 'Could not get a bridge quote. Try again.');
  return;
```

```
}
// proceeds straight to getClientForChain + executeBridge, no economic check
```

`getQuote` sends no slippage or minimum-amount parameter to LI.FI, and although the LI.FI response carries the received amount, the code only logs it (`src/services/bridgeService.ts:152-178`):

```
url.searchParams.set('fromAmount', String(fromAmount));
url.searchParams.set('bridges', LIFI_ALLOWED_BRIDGES.join(','));
url.searchParams.set('integrator', LIFI_INTEGRATOR_ID);
// no 'slippage', no minimum-amount param
// ...
debugLog(`[Bridge] Quote: ${data.tool}, receive ${data.estimate.toAmount} USDC
↪  ...`); // logged, not acted on
return data;
```

The whole path runs automatically on a balance poll, with no confirmation (`src/hooks/useAutoBridge.ts:54-61`):

```
const locked = await isAutoBridgeLocked(detected.chainId);
if (locked || bridgingRef.current) return;
bridgingRef.current = true;
await setAutoBridgeLock(detected.chainId, 'bridging');
await executeAutoBridge(detected, walletAddress, getClientForChain, /*
↪  onStatusChange */ ...);
```

Per the LI.FI quote reference, the `estimate` object already returns `toAmount` (estimated received) and `toAmountMin` (guaranteed minimum after slippage), and the `/quote` endpoint accepts a `slippage` parameter. The code uses none of them for a decision.

Impact

A small balance on an expensive chain is bridged at whatever rate LI.FI returns, silently and without consent, so the user can receive materially less than they held for a sweep they never approved. For a product whose users are meant to never see crypto mechanics, an unconfirmed lossy bridge of their own funds is a real intent-versus-outcome gap.

Proof of Concept

Preconditions: auto-bridge is enabled and the user's smart wallet holds a small USDC balance (for example $3 to $8) on a high-cost chain such as Ethereum, above the flat $1 trigger.

1. The 20 second balance poll in `useAutoBridge` detects the non-Base balance and calls `executeAutoBridge` with no confirmation.

2. `getQuote` requests a LI.FI quote with no slippage or minimum-amount constraint.
3. LI.FI returns a valid route whose `estimate.toAmount` is well below `fromAmount` because the relayer and gas-reimbursement fee consume a large share of the small principal.
4. `executeAutoBridge` runs the bridge because its only gate is `if (!quote)`; the user receives the low `toAmount` on Base with no floor and no consent.

Code Snippet

```
// src/services/autoBridgeService.ts:129-134  (no economic-viability check)
const quote = await getQuote(detected.chainId, Number(detected.balance),
↪  walletAddress);
if (!quote) { onStatusChange('failed', '...'); return; }
// executes with no comparison of estimate.toAmountMin to fromAmount

// src/services/bridgeService.ts  getQuote  (no slippage param, toAmount ignored)
url.searchParams.set('fromAmount', String(fromAmount));
// ...no url.searchParams.set('slippage', ...)
```

Tool Used

Manual Review, plus the LI.FI API documentation for the quote request and response schema.

Recommendation

Enforce an economic-viability floor before executing, and make the trigger threshold chain-aware.

1. Pass an explicit slippage bound to LI.FI. Per the LI.FI quote docs the field is `slippage`, a decimal between 0 and 1:

```
url.searchParams.set('slippage', '0.005'); // 0.5%
```

2. Reject economically poor routes using the guaranteed-minimum received the quote already returns (`toAmountMin`), not the `toAmount` estimate. Add `toAmountMin: string` to the `BridgeQuoteEstimate` type, then:

```
const fromAmount = Number(detected.balance);
const toMin = Number(quote.estimate.toAmountMin);
const MIN_RECEIVED_RATIO = 0.97; // reject more than ~3% total erosion; tune per
↪  policy
if (toMin < fromAmount * MIN_RECEIVED_RATIO) {
  onStatusChange('failed', 'Bridge cost too high for this amount right now.');
```

```
  return; // no on-chain tx, no gas spent
}
```

3. Replace the flat $1 trigger with a per-chain minimum that covers that chain's typical bridge cost (for example a `minBridgeUsd` on `BridgeChain`, set high for Ethereum and low for cheap L2s), so a $1 sweep never fires on an expensive chain.

Note that `slippage` and `toAmountMin` protect against price movement between quote and execution, while the fee that erodes a small principal is already inside `toAmount`. The minimum-received floor in step 2 is what actually defends against the lossy small-sweep and, together with a chain-aware threshold, removes the repeated-sponsored-bridge griefing vector.

Discussion

alexanderpi007

Fixed in fcd5da08. Root cause: auto-bridge fetched one LI.FI quote and executed with `if (!quote)` the only gate – no slippage, no minimum-received floor, and a flat $1 trigger on every chain incl. Ethereum mainnet where fees dwarf $1. Fix: getQuote now sends slippage=0.005; BridgeQuoteEstimate gains toAmountMin; executeAutoBridge enforces a minimum-received floor before executing – rejects (no tx/gas) when USD-normalized toAmountMin < fromAmount × 0.97 (decimals normalized via fromToken/toToken.decimals so 18-dec vs 6-dec USDC compare correctly). BridgeChain gains minBridgeUsd (Ethereum $30, cheap L2s $1) driving the trigger, so a $1 sweep never fires on mainnet. Tests: toAmountMin $8 on $10 sent → rejected, no execute; a healthy L2 route (0.99) executes; getMinBridgeAmount = $30 ETH / $1 Arbitrum; a $5 Ethereum balance no longer triggers while $5 on Arbitrum does. Unit-tested (the refund/economic paths can't be forced by hand).

Issue L-1: Auto-bridge dedup lock keyed by `chainId` only and not per-wallet [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/2

Summary

The auto-bridge dedup lock (`isAutoBridgeLocked/setAutoBridgeLock`) is keyed by `chainId` only (`autobridge_lock_${chainId}`) and not by `chainId + walletAddress`. On a shared device or simulator with multiple wallet accounts - one account's bridge lock prevents another account from bridging from the same chain for the lock TTL (30 min success / 5 min failure).

Vulnerability Detail

In `src/services/autoBridgeService.ts:32-53`:

```
const DEDUP_KEY_PREFIX = 'autobridge_lock_';

export async function isAutoBridgeLocked(chainId: number): Promise<boolean> {
  const raw = await AsyncStorage.getItem(`${DEDUP_KEY_PREFIX}${chainId}`);
  // ...
}
```

The key is `autobridge_lock_1` for Ethereum, `autobridge_lock_42161` for Arbitrum, etc with no wallet address component.

Impact

Since unflat is a single-user mobile app and shared-device scenarios are unlikely in production. However, on a developer device or TestFlight with multiple test accounts, this causes confusing cross-account lock-outs. No fund loss, it's only a UX/availability issue where a second account's auto-bridge is delayed.

Code Snippet

```
const DEDUP_KEY_PREFIX = 'autobridge_lock_';
/** Lock TTL after successful bridge: 30 minutes (prevents re-bridging same funds)
↪  */
export const DEDUP_TTL_MS = 30 * 60 * 1000;
/** Lock TTL after failure: 5 minutes (prevents hammering, but retries faster) */
export const FAILURE_TTL_MS = 5 * 60 * 1000;
```

```
interface LockData {
  timestamp: number;
  reason: 'bridging' | 'failed';
}

/** Check if a chain is currently locked (bridge in progress or recently failed) */
export async function isAutoBridgeLocked(chainId: number): Promise<boolean> {
  const raw = await AsyncStorage.getItem(`${DEDUP_KEY_PREFIX}${chainId}`);
  if (!raw) return false;

  // Support legacy format (plain timestamp number)
  const parsed = raw.startsWith('{')
    ? (JSON.parse(raw) as LockData)
    : { timestamp: parseInt(raw, 10), reason: 'bridging' as const };
  const ttl = parsed.reason === 'failed' ? FAILURE_TTL_MS : DEDUP_TTL_MS;
  return Date.now() - parsed.timestamp < ttl;
}
```

Tool Used

Manual Review

Recommendation

Key the lock by `${DEDUP_KEY_PREFIX}${chainId}_${walletAddress.toLowerCase()}`.

Discussion

alexanderpi007

Fixed in b5bdbb55. Root cause: the auto-bridge dedup lock key was autobridge_lock_ (chainId only), so on a multi-account device one account's lock blocked another account on the same chain. Fix: key the lock per (chain, wallet) via lockKey(chainId, wallet.toLowerCase()); walletAddress threaded through isAutoBridgeLocked / setAutoBridgeLock / clearAutoBridgeLock and the hook's call sites. Tests: wallet A locked on Arbitrum → wallet B on the same chain is not locked. Unit-tested.

Issue L-2: `verifyWalletOwnership` matches any linked-account `address` without validating the account `type` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/3

Summary

`verify-wallet-ownership.ts` treats a wallet as owned by a Privy user if **any** entry in the Privy `linked_accounts` array has an `address` field equal to the target wallet, without checking that entry's `type`. The function's own documented policy (lines 11-15) is to accept only `smart_wallet` and embedded `wallet` accounts, but the code collects the `address` off **every** account type. Because this function is the sole authorization boundary for `delete-account` (it binds a client-supplied `wallet_address` to the verified Privy DID before deleting wallet-keyed rows), trusting the `address` of arbitrary account types relies on undocumented Privy invariants and should be tightened to an explicit account-type allowlist.

Vulnerability Detail

In `supabase/functions/_shared/auth/verify-wallet-ownership.ts:71-77`:

```
const target = wallet.toLowerCase();
const owned = linkedAccounts.some((acct) => {
  if (typeof acct !== 'object' || acct === null) return false;
  const addr = (acct as { address?: unknown }).address;
  return typeof addr === 'string' && addr.toLowerCase() === target;
});
```

The predicate reads `acct.address` and compares it (lowercased) to the target wallet, but never inspects `acct.type` or `acct.connector_type`. The docstring (lines 11-15) states only two variants are accepted, `{ type: 'smart_wallet', address }` and `{ type: 'wallet', connector_type: 'embedded', address }`, yet line 16 ("We collect EVERY `address` present") and the loop implement a broader match than the stated policy. The code is broader than its own documentation.

The trust model that makes the broad match *currently* safe rests on two undocumented, Privy-side invariants. (1) Global wallet-address uniqueness: a given wallet address can be linked to at most one Privy DID, so a victim's table-key address cannot also appear in an attacker's `linked_accounts`. (2) No non-wallet account type exposes a top-level, attacker-settable `address`: of current Privy shapes only `wallet` and `smart_wallet` carry a top-level `0x…` `address`, while `email`/`phone`/OAuth carry non-`0x` values that cannot collide with a `0x…` target, and `cross_app`/`farcaster` *nest* their addresses (`embedded_wallets[].address`, `owner_address`) which this loop never reaches. The `linked_accounts` array and its

per-type fields are documented in Privy's Get user API reference.

Neither invariant is asserted by the code, and one is already weaker than assumed: the loop accepts **external** `wallet` accounts (any `connector_type`), whereas the docstring intends **embedded** only. Cross-account uniqueness for *external* wallets is Privy-config-dependent and weaker than for embedded/smart wallets.

Impact

`verifyWalletOwnership` confirms a client-supplied `wallet_address` belongs to the caller's verified Privy DID before `handleDeleteAccount` acts on that wallet's rows. Because it accepts the `address` of any account type instead of validating the type, it is not a robust ownership check; it silently depends on undocumented Privy behavior: that only ownership-proving account types (`wallet`, `smart_wallet`) ever carry a top-level `0x` `address`, and that Privy enforces address uniqueness across accounts. Those assumptions hold for today's Privy account types, so there is no live issue (severity **Low / defense-in-depth**), but a future Privy account type with a settable top-level `address`, or a config allowing the same external wallet on multiple accounts, would let the check pass for a wallet the caller does not actually control. Validating the account type makes the check self-contained instead of relying on those external invariants.

Code Snippet

```
export async function verifyWalletOwnership(
  did: string,
  wallet: string,
): Promise<WalletOwnershipResult> {
  // ... fetch Privy user, parse linked_accounts ...

  const linkedAccounts: unknown[] =
    typeof user === 'object' &&
    user !== null &&
    Array.isArray((user as { linked_accounts?: unknown }).linked_accounts)
      ? (user as { linked_accounts: unknown[] }).linked_accounts
      : [];

  const target = wallet.toLowerCase();
  const owned = linkedAccounts.some((acct) => {
    if (typeof acct !== 'object' || acct === null) return false;
    const addr = (acct as { address?: unknown }).address;
    return typeof addr === 'string' && addr.toLowerCase() === target;
  });

  return { owned, ok: true, status: res.status };
}
```

Tool Used

Manual Review

Recommendation

Allowlist the account types whose `address` proves wallet ownership, matching the documented policy so the enforced behavior and the stated intent agree:

```
const target = wallet.toLowerCase();
const owned = linkedAccounts.some((acct) => {
  if (typeof acct !== 'object' || acct === null) return false;
  const a = acct as { type?: unknown; connector_type?: unknown; address?: unknown };
  // Only account types that PROVE wallet ownership (matches the docstring policy).
  const isOwnedWallet =
    a.type === 'smart_wallet' ||
    (a.type === 'wallet' && a.connector_type === 'embedded');
  if (!isOwnedWallet) return false;
  return typeof a.address === 'string' && a.address.toLowerCase() === target;
});
```

If proven external wallets should also be accepted, use `a.type === 'wallet'` without the `connector_type` clause, but that leans on Privy's external-wallet uniqueness, so make it an explicit decision rather than an omission. Since prod tables are keyed by the deterministic per-user `smart_wallet` address, the tightest policy is to restrict the match to `type === 'smart_wallet'` only, which removes any reliance on external/embedded EOA linking behavior.

Discussion

Raiders0786

@Ali-Kalout - i think we should change this to `general-health` issue rather than low since the function checks the caller's own `linked_accounts` (DID from their verified JWT), and only wallet/smart_wallet types carry a top-level address both require cryptographic proof of key ownership to link - so an attacker can't get a victim's address into their own `linked_accounts` array to pass the match. thanks!

alexanderpi007

Fixed in e4f1d8281af96113e038b14aa831d9a8cac08c6e. Root cause: verifyWalletOwnership matched any linked_accounts entry with address === target, ignoring acct.type – the sole authorization boundary for delete-account, relying on undocumented Privy invariants. Fix: ownership now requires a controlled wallet before the address counts – type==='smart_wallet' || (type==='wallet' && connector_type==='embedded'), then the case-insensitive match. Both types kept deliberately (documented): the scrubbed tables are keyed by smartWalletAddress || eoaAddress, so embedded is a real fallback key and smart_wallet-only would strand

embedded-keyed rows. External/cross_app/email accounts carrying an address no longer prove ownership. Tests: +9 tests – cross_app/email/external-wallet with a matching address → not owned; smart_wallet and embedded → owned; delete-account gate still passes. 49 deno tests green.

Issue L-3: `resolvePrivyUserId` accepts an unverified `privy_user_id` parameter in the request body [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/5

Summary

The `resolvePrivyUserId` helper in `_shared/auth/resolve-privy-user-id.ts` still accepts an **unverified** `privy_user_id` from the request body as a valid identity source when no `X-Privy-Token` header is present.

The sole current in-scope consumer (`delete-account/handler.ts:84`) gates on `source === 'jwt'` and rejects body-sourced auth, so `delete-account` itself is protected. However, the shared helper returns `{ ok: true, privyUserId: bodyId, source: 'body' }` so any future Edge Function that calls `resolvePrivyUserId` without independently checking `source === 'jwt'` would accept an attacker-supplied identity.

Vulnerability Detail

In `supabase/functions/_shared/auth/resolve-privy-user-id.ts`, the resolution logic (around line 112-143):

```
const bodyId =
  body && typeof body.privy_user_id === 'string' &&
  ↪  body.privy_user_id.trim().length > 0
    ? body.privy_user_id.trim()
    : null;
```

When no JWT header is present but `bodyId` is non-null (branch ~line 141):

```
return { ok: true, privyUserId: bodyId, source: 'body' };
```

This returns a successful auth result with an attacker-controlled identity. The caller is responsible for checking `source === 'jwt'` for sensitive operations.

`delete-account/handler.ts:84`:

```
if (auth.source !== 'jwt') return json({ code: 'jwt_required' }, 401);
```

This gate is correct. But the defense is defense-in-depth only as the shared helper itself does not enforce JWT requirement for destructive operations.

Impact

- Turrently no live exploit since the `delete-account` correctly rejects body-sourced auth.
- The shared helper is designed to be reused by all Edge Functions. Any handler that omits the `source === 'jwt'` check would allow full account impersonation (an attacker can supply any `privy_user_id` in the request body to act as that user).

Code Snippet

```
const token = extractPrivyToken(req);

if (token !== null) {
  // ... verify JWT, return { ok: true, privyUserId: jwtSub, source: 'jwt' } ...
}

// No X-Privy-Token: accept the body value unverified (Phase B legacy path).
if (bodyPrivyId !== null) {
  return { ok: true, privyUserId: bodyPrivyId, source: 'body' };
}

return { ok: false, code: 'missing_auth', status: 401 };
```

Tool Used

Manual Review

Recommendation

Remove the body fallback from `resolvePrivyUserId`. If body traffic truly reached near-zero then the fallback serves no purpose. If it hasn't, enforce `source === 'jwt'` at the helper level for all callers and log/alert on body-sourced attempts rather than accepting them.

Discussion

alexanderpi007

Fixed in 9a97e62c973bc4ad3aa07eb55093f2cf1e1e6679. Root cause: resolvePrivyUserId returned { ok:true, source:'body' } from an unverified body privy_user_id when no JWT was present; only delete-account's source!=='jwt' guard made it safe, so any future caller omitting it would allow impersonation. Fix: one production caller (delete-account, already JWT-gated) and past the docstring's Phase C removal target → removed the body fallback entirely: no JWT → missing_auth 401. The body is now read only to detect a

client bug (JWT + mismatch → 403), never trusted as identity. AuthSource narrowed to 'jwt' so the invariant is type-enforced. Tests: body-source cases now assert missing_auth; a victim DID in the body with no JWT is rejected, not impersonated; conflict detection still 403; JWT happy-path delete still works. 49 deno tests green.

Issue L-4: Auto-bridge dedup lock legacy parse can produce `NaN` which bypasses the lock [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/6

Summary

The legacy-format fallback in `isAutoBridgeLocked` parses non-JSON AsyncStorage values via `parseInt(raw, 10)`. A corrupted or non-numeric value produces `NaN`, and `Date.now() - NaN < ttl` evaluates to `false` (NaN comparisons are always false) causing the lock to appear expired and enabling a duplicate bridge.

Vulnerability Detail

In `src/services/autoBridgeService.ts:44-54`:

```
export async function isAutoBridgeLocked(chainId: number): Promise<boolean> {
  const raw = await AsyncStorage.getItem(`${DEDUP_KEY_PREFIX}${chainId}`);
  if (!raw) return false;

  const parsed = raw.startsWith('{')
    ? (JSON.parse(raw) as LockData)
    : { timestamp: parseInt(raw, 10), reason: 'bridging' as const };
  const ttl = parsed.reason === 'failed' ? FAILURE_TTL_MS : DEDUP_TTL_MS;
  return Date.now() - parsed.timestamp < ttl;
}
```

If `raw` is a corrupted string like `"abc"` or an empty-ish value that doesn't start with `{`:

- `parseInt("abc", 10)` → `NaN`
- `Date.now() - NaN` → `NaN`
- `NaN < 1800000` → `false`
- Lock reports as unlocked.

Impact

As the `AsyncStorage` corruption is rare, and the legacy path is only reached for pre-JSON values from an old app version the issue is less severe. However, if a malicious app on a rooted device writes a non-numeric string to the lock key, it disables dedup protection, potentially causing a double bridge submission (double-spending the USDC balance, though only one would succeed onchain).

Code Snippet

```
export async function isAutoBridgeLocked(chainId: number): Promise<boolean> {
  const raw = await AsyncStorage.getItem(`${DEDUP_KEY_PREFIX}${chainId}`);
  if (!raw) return false;

  // Support legacy format (plain timestamp number)
  const parsed = raw.startsWith('{')
    ? (JSON.parse(raw) as LockData)
    : { timestamp: parseInt(raw, 10), reason: 'bridging' as const };
  const ttl = parsed.reason === 'failed' ? FAILURE_TTL_MS : DEDUP_TTL_MS;
  return Date.now() - parsed.timestamp < ttl;
}
```

Tool Used

Manual source review.

Recommendation

Add a `isNaN` check after `parseInt`:

```
const timestamp = parseInt(raw, 10);
if (isNaN(timestamp)) return false; // or return true (fail-closed, assume locked)
```

Fail-closed (`return true` on parse failure) is safer where a corrupted lock should block, not enable.

Discussion

alexanderpi007

Fixed in 739a8a4b. Root cause: isAutoBridgeLocked's legacy fallback parseInt(raw,10) yields NaN on a corrupted value, and Date.now() − NaN < ttl is false → the lock reports UNLOCKED, allowing a duplicate bridge. Fix: after parseInt, Number.isNaN(timestamp) → return true (fail closed: a lock we can't read is treated as locked). Tests: a corrupted non-numeric legacy value → isAutoBridgeLocked returns true. Unit-tested.

Issue L-5: `jose@5` `esm.sh` import uses major-version range poses supply-chain risk [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/7

Summary

`privy-jwks-verifier.ts` imports jose via `https://esm.sh/jose@5` is a major-version range. `esm.sh` resolves semver ranges at request time, so any `5.x.y` release (including a compromised or buggy one) would be served transparently on the next Edge Function cold start, with no code change and no review. All other `esm.sh` imports in the codebase are correctly exact-pinned (`@supabase/supabase-js@2.45.4`, `viem@2.21.0`).

Vulnerability Detail

In `supabase/functions/_shared/auth/privy-jwks-verifier.ts:22`:

```
import { createRemoteJWKSet, jwtVerify } from 'https://esm.sh/jose@5';
```

Also in the test file (`privy-jwks-verifier.test.ts:17`):

```
import { exportJWK, generateKeyPair, type KeyLike, SignJWT } from
↪  'https://esm.sh/jose@5';
```

jose is the SOLE JWT verification library and it guards the entire auth boundary. A compromised jose 5.x release could:

- Bypass signature verification (accepting forged JWTs)
- Leak the JWKS data
- Introduce timing attacks

Impact

An attacker who compromises the jose npm package (or esm.sh's build/CDN) could inject a malicious jose 5.x that transparently passes JWT verification for attacker-crafted tokens. This would bypass the entire Privy auth boundary for all Edge Functions, enabling full account impersonation.

Likelihood is low (requires npm/CDN supply-chain compromise) but impact is critical (total auth bypass) and the fix is trivial.

Code Snippet

```
import { createRemoteJWKSet, jwtVerify } from 'https://esm.sh/jose@5';
```

Tool Used

Manual source review + grep of all esm.sh imports.

Recommendation

Pin to exact version:

```
import { createRemoteJWKSet, jwtVerify } from 'https://esm.sh/jose@5.9.6';
```

Check current latest 5.x version and pin it. Add a `deno.lock` entry or a CI check that flags unpinned `esm.sh` imports.

Discussion

Ali-Kalout

I believe `deno.lock` already resolves `jose@5` -> `jose@5.10.0` and stores SHA-256 integrity hashes for every file (line 7 + 85+), so it's effectively pinned + tamper-checked already, a new or swapped 5.x wouldn't be silently picked up, it'd fail the integrity check. Am I missing anything?

Raiders0786

Good catch @Ali-Kalout - you're right that `deno.lock` does redirect `https://esm.sh/jose@5` → `https://esm.sh/jose@5.10.0` and stores SHA-256 integrity hashes for those remote modules (see redirects + remote entries in `deno.lock`). For local Deno runs that honor the lock, a swapped 5.x would fail integrity rather than load silently.

On Supabase Edge Functions specifically, deploy packages the module graph into an eszip. Hosted cold starts load from that bundle; they do not re-resolve a floating major from esm.sh on every cold start the way the original writeup implied. So the "compromised jose 5.x is served on the next cold start with no code change" story is not accurate here.

What remains is weaker and hygiene-level:

1. The source import is still a major-only range (`jose@5`), unlike the exact pins for `viem@2.21.0` and `@supabase/supabase-js@2.45.4`.
2. Safety depends on lock enforcement at resolve/deploy time. If someone deploys without the lock (or regenerates the lock carelessly), the floating major is back in play.

3. Exact pinning in source still makes upgrades intentional and reviewable.

I'll downgrade this to Low/Info.

My recommendation stands as best practice: pin e.g. `https://esm.sh/jose@5.10.0` in source so the import and lock agree without relying on the lock redirect alone.

Happy to leave the issue open as Low/Info unless we agree to close it as mitigated-by-lock.

Ali-Kalout

> Good catch @Ali-Kalout - you're right that `deno.lock` does redirect `https://esm.sh/jose@5` → `https://esm.sh/jose@5.10.0` and stores SHA-256 integrity hashes for those remote modules (see redirects + remote entries in `deno.lock`). For local Deno runs that honor the lock, a swapped 5.x would fail integrity rather than load silently.
> On Supabase Edge Functions specifically, deploy packages the module graph into an eszip. Hosted cold starts load from that bundle; they do not re-resolve a floating major from esm.sh on every cold start the way the original writeup implied. So the "compromised jose 5.x is served on the next cold start with no code change" story is not accurate here.
> What remains is weaker and hygiene-level:
> 1. The source import is still a major-only range (`jose@5`), unlike the exact pins for `viem@2.21.0` and `@supabase/supabase-js@2.45.4`.
> 2. Safety depends on lock enforcement at resolve/deploy time. If someone deploys without the lock (or regenerates the lock carelessly), the floating major is back in play.
> 3. Exact pinning in source still makes upgrades intentional and reviewable.
>
> I'll downgrade this to Low/Info.
> My recommendation stands as best practice: pin e.g. `https://esm.sh/jose@5.10.0` in source so the import and lock agree without relying on the lock redirect alone.
> Happy to leave the issue open as Low/Info unless we agree to close it as mitigated-by-lock.

Makes sense to me; agree that we should leave this as a low/info.

alexanderpi007

Fixed in 16b471ed037ee9dace6f2cbf9c4de14845b0bf35. Root cause: privy-jwks-verifier.ts imported jose via esm.sh/jose@5 (major range) – the sole JWT-verification library. deno.lock already pinned 5.10.0 with integrity hashes, but the source import should be exact so upgrades are intentional and don't rely on lock enforcement alone. Fix: pinned to esm.sh/jose@5.10.0 in both the source and the test import; deno.lock hashes are byte-identical (resolves to the already-locked version). jose was the last unpinned esm.sh import (viem/supabase-js were already exact). Tests: the privy-jwks-verifier tests generate keypairs, sign and verify real JWTs through the pinned lib. 49 deno tests green, type-check clean.

Issue L-6: Offramp shows "Transfer Complete" on submission, without confirming the transfer mined [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/9

Summary

The cash-out flow tells the user "Transfer Complete! ... EUR will arrive in your bank in 1-2 business days" the moment the USDC transfer is submitted, not when it is confirmed on-chain. `handleOfframpTransfer` awaits `sendTransaction` (which resolves on submission) and immediately shows the success alert and fires the completion analytics; the only failure branch is a thrown submission error. A transfer that reverts, is dropped, or never mines still shows as complete. Every other fund-moving flow in the app (deposit, withdraw) confirms the transaction on-chain before declaring success.

Vulnerability Detail

In `src/hooks/useOfframpTransfer.ts:281-302`:

```
await smartWalletClient.sendTransaction({
  to: TOKENS.USDC as `0x${string}`,
  data,
});

// Clear the expected address after successful transfer
await clearExpectedOfframpAddress();

Analytics.trackOfframpCompleted(offrampParams.amount, 'coinbase');

Alert.alert(
  'Transfer Complete!',
  'Your USDC has been sent to Coinbase. EUR will arrive in your bank in 1-2
  ↪  business days.',
  [{ text: 'OK', onPress: () => closeOfframpSheet() }],
);

refetchBalances();
} catch (error) {
  // only reached if sendTransaction() itself throws (i.e. submission failed)
  Alert.alert('Transfer Failed', errorMessage || 'Something went wrong');
}
```

`smartWalletClient.sendTransaction` resolves with the submitted transaction hash (the return value is not even captured here). There is no `waitForTransaction`, no receipt read,

and no `status === 'failed'` check. The `catch` only runs if the submission promise rejects, which is before the transaction is mined, not if the transaction reverts after being accepted.

The deposit and withdraw paths handle the same client differently. In `src/services/strategyExecution.ts`, `executeWithdrawBatch` calls `const confirmation = await waitForTransaction(hash)` and throws on `!confirmation.found` (timeout/drop) and on `confirmation.status === 'failed'` (revert) before reporting success. That the withdraw path adds this step confirms `sendTransaction` returns a submitted-only hash and is not self-confirming.

Impact

On any revert, drop, or non-mine of the transfer, the user is told the cash-out completed and EUR is arriving in 1-2 business days when no USDC left their wallet. No funds are lost: the USDC stays in the user's own non-custodial wallet. The effect is a false confirmation of a money-out event, the user waits for a payout that never comes, may not retry, and the analytics record a completed offramp that did not happen. `refetchBalances()` at line 297 reads chain state, so the wallet balance still shows the USDC and partly contradicts the message.

Code Snippet

```
// src/hooks/useOfframpTransfer.ts  handleOfframpTransfer
try {
  // ...validate params, encode transfer calldata...
  await smartWalletClient.sendTransaction({ to: TOKENS.USDC, data }); // resolves
  ↪  on SUBMIT, hash discarded
  await clearExpectedOfframpAddress();
  Analytics.trackOfframpCompleted(offrampParams.amount, 'coinbase');
  Alert.alert('Transfer Complete!', 'Your USDC has been sent to Coinbase. EUR will
  ↪  arrive in your bank in 1-2 business days.', [/* OK */]);
  refetchBalances();
} catch (error) {
  Alert.alert('Transfer Failed', errorMessage || 'Something went wrong'); // only
  ↪  on submission throw
}
```

Tool Used

Manual Review

Recommendation

Confirm the transfer on-chain before showing success, mirroring the withdraw path:

```
const hash = await smartWalletClient.sendTransaction({ to: TOKENS.USDC, data });
const receipt = await publicClient.waitForTransactionReceipt({ hash });
if (receipt.status === 'reverted') {
  Alert.alert('Transfer Failed', 'The transfer did not complete. Please try
  ↪  again.');
  return;
}
// only now: clearExpectedOfframpAddress(), trackOfframpCompleted(...), and the
↪  'Transfer Complete!' alert
```

Show "Transfer Complete!" and call `trackOfframpCompleted` only after the receipt confirms success; on a revert or timeout, route to the failure branch instead.

Discussion

alexanderpi007

Fixed in 8e069d99. Root cause: handleOfframpTransfer awaited sendTransaction (resolves on submit, hash discarded) then immediately showed "Transfer Complete!" and fired trackOfframpCompleted; only a submission throw was handled, so a reverted/dropped transfer still showed complete. Fix: capture the hash, publicClient.waitForTransactionReceipt, throw on receipt.status === 'reverted' (→ the existing failure branch); run clearExpectedOfframpAddress / trackOfframpCompleted / the success alert only on a confirmed receipt – mirroring the withdraw path's confirmTx. Verified: mirrors the established withdraw confirmTx pattern; the offramp hook body isn't unit-testable in this repo (no React-hook harness), so it's verified by diff against that pattern.

Issue L-7: `getMinimumDepositUsdc` double-converts the covered minimum, so the covered-deposit minimum hint reads $0 [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/10

Summary

`getMinimumDepositUsdc` reads the Insured Vault's `minimumRequestAssets()` and passes it through `morphoVault.convertToAssets()` as if it were an 18-decimal bbqUSDC share amount. On-chain, `minimumRequestAssets()` is already a USDC 6-decimal value ($1). Running a 6-decimal value through an 18-decimal conversion floors it to 0, so the function returns $0 as the covered-deposit minimum. The pre-flight gate and `CoveredMinimumNotMetError` treat the same value correctly as USDC; `getMinimumDepositUsdc` is the inconsistent one, and its docstring mislabels the unit as bbqUSDC.

Vulnerability Detail

In `src/services/coveredVaultExecution.ts:391-406`:

```
export async function getMinimumDepositUsdc(config: CoveredVaultConfig):
↪  Promise<number | null> {
  const minShares = await getMinimumRequestAssets(config);            //
  ↪  minimumRequestAssets() -> USDC 6-dec ($1)
  if (minShares === null || minShares <= BigInt(0)) return 0;
  const usdcRaw = (await publicClient.readContract({
    address: config.morphoVaultAddress,
    abi: metaMorphoAbi,
    functionName: 'convertToAssets',                                   // treats
    ↪  minShares as 18-dec bbqUSDC shares
    args: [minShares],
  })) as bigint;
  return Number(usdcRaw) / 10 ** config.underlyingDecimals;            // 0 / 1e6 =
  ↪  $0
}
```

`minimumRequestAssets()` is USDC 6-decimal, and the codebase already treats it that way in two places:

- The pre-flight gate `buildCoveredDepositPhase1Raw` compares `previewUsdc < minimum`, where `previewUsdc = morphoSharesToUsdc(previewedShares)` is USDC. The comment on `morphoSharesToUsdc` (lines 345-347) states the on-chain gate is `require(asset.convertToAssets(requestAmount) >= minimumRequestAssets)` in USDC.
- `CoveredMinimumNotMetError` formats the value as `Math.ceil(Number(minimumUsdcRaw) / 1e4) / 100`, i.e. USDC 6-decimal.

`getMinimumDepositUsdc` (and the `getMinimumRequestAssets` docstring at line 326, and its own docstring at line 387) instead label it "bbqUSDC" and re-convert it. That extra conversion is the bug.

On-chain verification (Base, chain 8453)

Read directly from the contracts (reproducible on BaseScan):

- Insured Vault `0x98380446865D66E23157378E260CBeC0033f991B` -> `minimumRequestAssets()` = **1000000** (= $1, USDC 6-decimal). If it were 18-decimal bbqUSDC this would be 1e-12 bbqUSDC, which is nonsensical. https://basescan.org/address/0x98380446865D66E23157378E260CBeC0033f991B#readProxyContract
- Morpho Vault (bbqUSDC) `0xbeeff7aE5E00Aae3Db302e4B0d8C883810a58100` -> `convertToAssets(1000000)` = **0**. This is exactly what `getMinimumDepositUsdc` computes, so it returns `0 / 1e6 = $0`. https://basescan.org/address/0xbeeff7aE5E00Aae3Db302e4B0d8C883810a58100#readContract
- Context: `convertToAssets(1000000000000000000)` (1e18) = **1037120**, i.e. 1 bbqUSDC ≈ 1.037 USDC, confirming `convertToAssets` maps 18-decimal bbqUSDC to 6-decimal USDC. So a $1 USDC value (1e6) sits 12 orders of magnitude below one bbqUSDC and floors to 0.

Impact

Display/UX only. `useCoveredVault` feeds `getMinimumDepositUsdc` into `minSliceUsdc`, so `computeCoveredDepositState.meetsMinimum` is always true and the covered-deposit minimum hint shows ~$0. A below-minimum covered deposit is not flagged in the UI; it proceeds to `buildCoveredDepositPhase1Raw` and aborts with `CoveredMinimumNotMetError`, which is thrown before any transaction is sent, so no funds move and no bad on-chain state occurs.

Code Snippet

```
// src/services/coveredVaultExecution.ts:391-406
const minShares = await getMinimumRequestAssets(config);   // 1_000_000 (= $1 USDC,
↪  6-dec)
const usdcRaw = await morphoVault.convertToAssets(minShares); //
↪  convertToAssets(1_000_000) = 0 (on-chain)
return Number(usdcRaw) / 1e6;                                  // 0 -> $0
```

Tool Used

Manual Review + on-chain reads against Base (`minimumRequestAssets()` and `convertToAssets()`, verifiable on BaseScan).

Recommendation

`minimumRequestAssets()` is already USDC 6-decimal, so do not re-convert it. Return it directly:

```
export async function getMinimumDepositUsdc(config: CoveredVaultConfig):
↪  Promise<number | null> {
  const minUsdcRaw = await getMinimumRequestAssets(config);
  if (minUsdcRaw === null || minUsdcRaw <= BigInt(0)) return 0;
  return Number(minUsdcRaw) / 10 ** config.underlyingDecimals;   // 1_000_000 / 1e6
  ↪  = $1
}
```

Also correct the two docstrings (lines 326 and 387) that label `minimumRequestAssets()` as bbqUSDC; it is USDC.

Discussion

alexanderpi007

Fixed in 66662620. Root cause: getMinimumDepositUsdc read minimumRequestAssets() (already USDC 6-dec, $1) and passed it through convertToAssets() as if it were 18-dec bbqUSDC shares → floored to 0 → $0 minimum hint, meetsMinimum always true (display/UX only; the real gate still blocks via CoveredMinimumNotMetError). Fix: return the value directly (Number(minUsdcRaw)/1e6); corrected the two docstrings that mislabeled minimumRequestAssets() as bbqUSDC. Tests: returns ~$1 (not $0), never calls convertToAssets, flags a below-$1 covered slice. Unit-tested.

Issue L-8: Offramp confirm has no synchronous re-entrancy guard, so a fast double-tap sends the USDC twice [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/20

Summary

`handleOfframpTransfer` gates re-entry only with the `isOfframpProcessing` React state flag, which updates on the next render rather than synchronously. Two taps on the cash-out confirm button that land before that re-render both pass the guard and both call `sendTransaction`, so a second identical USDC transfer of the offramp amount is fired for which the backend created no matching sell order. The withdraw path in the same app already protects against this with a synchronous `withdrawInFlightRef`, and even the auto-bridge hook uses a `bridgingRef`; the offramp confirm is the one fund-moving action without it.

Vulnerability Detail

In `src/hooks/useOfframpTransfer.ts:250-306`, the only re-entry gate is a state setter:

```
const handleOfframpTransfer = async () => {
  if (!offrampParams || !smartWalletClient) return;

  setIsOfframpProcessing(true);      // :253  React state - applies on NEXT render,
  ↪  not now
  try {
    // ...validate, encode calldata...
    await smartWalletClient.sendTransaction({   // :281  sends USDC to the Coinbase
    ↪  deposit address
      to: TOKENS.USDC as `0x${string}`,
      data,
    });
    // ...success alert...
  } finally {
    setIsOfframpProcessing(false);   // :304
  }
};
```

`setIsOfframpProcessing(true)` does not mutate a value the next synchronous call can observe; the component re-renders later. Between the first tap and that re-render, a second tap enters `handleOfframpTransfer` again, sees the same stale render scope, and proceeds to a second `sendTransaction`. Whether the button's `disabled={isOfframpProcessing}` prop blocks the second tap depends entirely on render timing, which is exactly the race a synchronous guard exists to remove.

In practice the re-render usually disables the button within a frame, so a normal-paced double-tap hits the disabled button and does nothing. The vulnerable window opens when the JS thread is busy (heavy render, slow device) and the re-render lags, letting both taps through before the button disables. That is precisely why the withdraw path uses a synchronous ref rather than relying on the state-driven `disabled` alone.

The withdraw path does it correctly (`src/screens/StrategiesScreen.tsx:318, 1194-1195`):

```
const withdrawInFlightRef = useRef(false);
// ...
if (withdrawInFlightRef.current) return;   // synchronous: the second tap returns
↪  immediately
withdrawInFlightRef.current = true;
```

The auto-bridge hook uses the same pattern (`src/hooks/useAutoBridge.ts:45`, `const bridgingRef = useRef(false)`). `handleOfframpTransfer` has no `useRef`-based in-flight flag, so it lacks the guarantee both other money-moving flows have.

Impact

When the race is hit, a second identical USDC transfer of the offramp amount is fired. Only one sell order exists (the backend created one when it issued the transfer intent), so the first transfer is the one matched and converted to EUR and the second has no matching order, meaning the user sends the offramp amount a second time without it being converted as expected. The funds go to a legitimate Coinbase-controlled deposit address rather than to an attacker, so nothing is provably lost, but a second money-out the user did not intend has still left the wallet.

Proof of Concept

Preconditions: a valid pending offramp with `offrampParams` populated and a funded wallet.

1. Open the cash-out confirmation sheet so `handleOfframpTransfer` is wired to the confirm button.
2. Double-tap the confirm button faster than one React render (a normal fast tap).
3. Both invocations pass `if (!offrampParams || !smartWalletClient) return` and reach `sendTransaction`, producing two USDC transfers to the same Coinbase deposit address for the same amount.
4. The backend matches one transfer to its single sell order; the second transfer sits unmatched at the deposit address.

Code Snippet

```
// src/hooks/useOfframpTransfer.ts  handleOfframpTransfer
setIsOfframpProcessing(true);            // async state, not a synchronous lock
// ...no `if (ref.current) return; ref.current = true;` ...
await smartWalletClient.sendTransaction({ to: TOKENS.USDC, data }); // fires once
↪  per tap that got through
```

```
// src/screens/StrategiesScreen.tsx  (withdraw path - the correct pattern)
if (withdrawInFlightRef.current) return;
withdrawInFlightRef.current = true;
```

Tool Used

Manual Review

Recommendation

Add a synchronous in-flight ref to the offramp confirm, mirroring the withdraw path:

```
const offrampInFlightRef = useRef(false);

const handleOfframpTransfer = async () => {
  if (!offrampParams || !smartWalletClient) return;
  if (offrampInFlightRef.current) return;   // synchronous double-tap guard
  offrampInFlightRef.current = true;
  setIsOfframpProcessing(true);
  try {
    // ...existing body...
  } finally {
    offrampInFlightRef.current = false;
    setIsOfframpProcessing(false);
  }
};
```

The ref check must be the first thing after the param guard so a second tap returns before any `sendTransaction`. Keep the existing `isOfframpProcessing` state for the button's disabled/spinner UI; the ref is what actually enforces single-flight.

Discussion

alexanderpi007

Fixed in 7823e313. Root cause: the only re-entry gate was the isOfframpProcessing React state, which applies on the next render, so a fast double-tap could pass twice and send

USDC twice. Fix: added a synchronous offrampInFlightRef checked/set as the first thing after the param guard, reset in finally – mirroring the withdraw path's withdrawInFlightRef. isOfframpProcessing retained for the button spinner. Verified: mirrors the established withdraw single-flight ref; hook body not unit-testable in this repo → verified by diff.

Issue L-9: Offramp submitted transfer has no durable consumed-marker, so a cross-session recovery or deep-link re-surfaces the still-pending transfer and permits a duplicate send [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/22

Summary

After the user confirms a cash-out and the USDC transfer is submitted, nothing durable records that the transfer was already sent. While the backend still reports the transfer as `PENDING_TRANSFER` (the on-chain mine plus Coinbase-detection window), the app's own recovery effect and offramp deep link re-fetch that still-pending transfer and re-open the confirm sheet with the same destination and amount. A user who relaunches the app or re-enters the deep link during that window can confirm and send the same USDC a second time.

Vulnerability Detail

On a successful send, the only cleanup is `clearExpectedOfframpAddress()` (a SecureStore key that `validateOfframpParams` no longer reads) and `closeOfframpSheet()`. No transfer id or submitted-marker is persisted (`src/hooks/useOfframpTransfer.ts:281-295`):

```
await smartWalletClient.sendTransaction({ to: TOKENS.USDC, data }); // USDC leaves
↪  the wallet
await clearExpectedOfframpAddress();                                  // non-durable,
↪  not read by validation
Analytics.trackOfframpCompleted(...);
Alert.alert('Transfer Complete!', ...);                              // OK ->
↪  closeOfframpSheet() -> setOfframpParams(null)
// nothing persists "this pendingTransfer was already submitted"
```

The kill/relaunch recovery uses a `useRef(false)` guard that resets on a new process, so it fires again after a relaunch (`src/hooks/useOfframpTransfer.ts:225-246`):

```
const recoveryAttempted = React.useRef(false);            // resets to false on
↪  relaunch
React.useEffect(() => {
  const attemptRecovery = () => {
    if (!shouldAttemptOfframpRecovery(recoveryAttempted.current, displayAddress))
    ↪  return;
    recoveryAttempted.current = true;
    void (async () => {
      const result = await resolveFetchedOfframp(wallet);  // hits the backend again
```

```
      if (result.ok) applyOfframpResult(result);          // re-opens the confirm
      ↪ sheet
    })();
  };
  attemptRecovery();                                       // on mount
  AppState.addEventListener('change', (n) => { if (n === 'active')
  ↪ attemptRecovery(); });
}, [displayAddress, applyOfframpResult]);
```

The `offramp/complete` deep link (`:203-216`) drives the same `resolveFetchedOfframp` path. `fetchOfframpPendingTransfer` returns ok whenever `status === 'PENDING_TRANSFER'` with a non-expired, checksummed transfer, with no check against any locally-consumed set, and `applyOfframpResult` re-populates the sheet (`:181-186`):

```
if (effect.kind === 'confirm') {
  setOfframpParams(effect.params);   // same to-address + amount as the
  ↪ already-sent transfer
  onShowConfirmRef.current?.();      // sheet re-opens
  return;
}
```

The user then taps confirm again and `handleOfframpTransfer` re-encodes and re-sends the identical `transfer(to, amount)`.

Impact

During the pending window the app re-presents an already-sent transfer and lets the user broadcast the same USDC a second time to the Coinbase deposit address. The duplicate has no matching sell order, so it is not converted to EUR. The window closes once Coinbase detects the on-chain deposit and the backend advances the transfer past `PENDING_TRANSFER`, and the funds land at a legitimate Coinbase-controlled deposit address recoverable through reconciliation.

Proof of Concept

Preconditions: a valid pending offramp exists and the user has just confirmed and successfully submitted the USDC transfer; Coinbase has not yet detected the deposit, so the backend still returns `status: 'PENDING_TRANSFER'`.

1. User confirms; `handleOfframpTransfer` submits the first USDC transfer; the success alert shows and the sheet closes. No submitted-marker is persisted.
2. Within the pending window the user kills and relaunches the app (or re-enters the `offramp/complete` deep link). `recoveryAttempted` resets, the mount effect fires, and `resolveFetchedOfframp` returns the same `PENDING_TRANSFER`.
3. `applyOfframpResult` re-opens the confirm sheet with the same to-address and amount.

4. The user taps confirm again; `handleOfframpTransfer` re-sends the identical transfer, a second USDC send to the Coinbase deposit address for which no matching sell order exists.

Code Snippet

```
// src/hooks/useOfframpTransfer.ts  handleOfframpTransfer - send with no durable
↪  consumed-marker
await smartWalletClient.sendTransaction({ to: TOKENS.USDC, data });
await clearExpectedOfframpAddress();  // cleared immediately, not read by validation
// (no persisted "submitted for pendingTransfer X")

// recovery effect re-surfaces the same transfer after relaunch
if (result.ok) applyOfframpResult(result);  // -> setOfframpParams + onShowConfirm
```

Tool Used

Manual Review

Recommendation

Persist a durable consumed-marker on send and honor it on every path that can re-present a transfer:

```
// on a successful send, mark this pendingTransfer consumed (SecureStore /
↪  AsyncStorage)
await smartWalletClient.sendTransaction({ to: TOKENS.USDC, data });
await markOfframpSubmitted(offrampParams.transferId);   // survives relaunch

// in fetchOfframpPendingTransfer / resolveFetchedOfframp, drop an
↪  already-submitted transfer
if (data.status === 'PENDING_TRANSFER' && pendingTransfer) {
  if (await isOfframpSubmitted(pendingTransfer.id)) {
    return { ok: false, reason: 'already_submitted' };  // do not re-open the
    ↪  confirm sheet
  }
  // ...existing validation...
}
```

Clear the marker once the backend advances past `PENDING_TRANSFER` so a genuinely new future cash-out is unaffected.

Discussion

alexanderpi007

Fixed in e4862c3a. Root cause: on a successful send only clearExpectedOfframpAddress() ran (non-durable); nothing recorded the transfer as sent, so during the PENDING_TRANSFER window the mount/app-open recovery and the offramp/complete deep link re-fetched the same transfer and re-opened the confirm sheet → duplicate send. Fix: persist a durable consumed-marker on a confirmed send (AsyncStorage, survives relaunch) keyed by the transfer's derived id (toAddress:amount:expiresAt – PendingOfframpTransfer has no server id); resolveFetchedOfframp returns already_submitted (sheet not re-opened) when the fetched pending transfer is already marked; the marker is cleared once the backend advances past PENDING_TRANSFER so a future cash-out isn't blocked. Tests: resolveFetchedOfframp – an already-submitted transfer → already_submitted/dismiss; a new transfer isn't blocked by a stale marker. Unit-tested (in-memory AsyncStorage mock).

Issue L-10: Force-claim eligibility treats missing `RedeemRequest` logs as eligible [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/23

Summary

`getForceClaimEligibility` uses about 60,000 blocks of lookback (roughly 33 hours at 2s). If `pendingRedeemRequest > 0` but no matching log is found in range, it returns `eligible: true` with `ageMs = MAX_SAFE_INTEGER`. RPC log gaps or indexing issues can show Force Claim early in the UI. Onchain `settleMaturedRedemption` still enforces the real 24h delay. This is mainly UX and wasted gas, not free early settlement of funds.

Vulnerability Detail

In `src/services/coveredVaultExecution.ts:1363-1366`.

Impact

The UI can offer Force Claim before maturity. Users who tap early get failed transactions. Onchain checks remain the real backstop.

Code Snippet

Tool Used

Manual Review

Recommendation

If logs are empty, return `null` (hide the button) or `eligible: false` unless an independent onchain maturity check confirms age. Treat lookback only as a secondary signal.

Discussion

alexanderpi007

Fixed in 888b6299. Root cause: getForceClaimEligibility returned eligible:true, ageMs:MAX when a pending redeem had no matching log in the ~60k-block lookback → the UI could offer Force Claim before 24h maturity, and settleMaturedRedemption reverts (RequestNotMatured), wasting gas. Fix: the block-lookback is now a positive signal only – a found log yields the age; its absence (and a null blockNumber) fails closed

→ null (button hidden). A genuinely matured request in its 24h–33h actionable window is always within the lookback and still resolves. (The ABI exposes no view to confirm request age independently.) Tests: empty logs → null; null block number → null; existing ≥24h and <24h cases still pass. Unit-tested.

Issue L-11: Partner API credentials ship in the mobile `expo.extra` bundle [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/24

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`app.config.ts` injects `lifiApiKey` and `cdpRpcUrl` (often containing API keys) into `expo.extra`, which ships in the client binary. Any user can extract them and abuse partner quotas.

Vulnerability Detail

In `app.config.ts:21-25`.

Impact

LI.FI and CDP project quotas can be stolen or exhausted. That can also degrade the app's own rate limits and raise partner cost.

Code Snippet

Tool Used

Manual Review

Recommendation

Proxy LI.FI and CDP through a backend that holds the keys. Rotate any keys that have shipped in clients. Treat client-embedded keys as public.

Discussion

alexanderpi007

Acknowledged. The two values in expo.extra are partner-integration credentials – LIFI_API_KEY (LI.FI integrator/quota key) and CDP_RPC_URL (Coinbase Developer Platform RPC endpoint). They are quota / rate-limit scoped, not fund-custody or user-auth secrets: abuse means partner-quota exhaustion or added cost, not access to user funds or accounts – which is why we're treating this as General Health rather than a blocking fix. Planned remediation, tracked as a follow-up outside this review's fix set: (1) rotate both credentials, treating the shipped values as public; (2) proxy LI.FI

quote/execute and CDP RPC calls through the unflat backend (api.unflat.app) so the keys live server-side only, and remove them from app.config.ts / expo.extra; (3) add per-user rate limiting at the proxy. Until the proxy ships we accept the residual quota-abuse risk. We'll update this issue when the proxy lands.

Issue L-12: `refreshRedeemCalls` fee recompute drops `UNFLAT_MIN_FEE_USDC` guard [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/25

Summary

`buildWithdrawBatch` and `buildPartialWithdrawBatch` skip the fee transfer when fee is below `UNFLAT_MIN_FEE_USDC`. The pre-execution refresh path in `strategyExecution` recomputes fee and appends a transfer when `feeAmountRaw > 0` only. It does not re-apply the minimum threshold. Covered `computeCoveredFeeRaw` still enforces the min. The paths are inconsistent and dust fee legs are possible.

Vulnerability Detail

In `src/services/strategyExecution.ts:865-881` (refresh fee recompute). Compare with the build path min-fee guard in the same file around the `dFee.greaterThanOrEqualTo(UNFLAT_MIN_FEE_USDC)` checks used by `buildWithdrawBatch` / `buildPartialWithdrawBatch`.

Impact

Very small unexpected treasury transfers can appear. Covered and raw fee paths disagree on dust handling.

Code Snippet

See `src/services/strategyExecution.ts:865-881` for the refresh path (`dFreshFee` / `feeAmountRaw > 0`) versus the build path `dFee.greaterThanOrEqualTo(UNFLAT_MIN_FEE_USDC)`.

Tool Used

Manual Review

Recommendation

Reuse the same min-fee guard after refresh recompute. Prefer one shared fee helper used by both raw and covered paths.

Discussion

alexanderpi007

Fixed in 91ea6d980a5b18ee6232a69d4bc6832a6b2a6d7c. Root cause: buildWithdrawBatch / buildPartialWithdrawBatch and the covered path suppress the treasury fee transfer below UNFLAT_MIN_FEE_USDC, but refreshRedeemCalls's pre-execution recompute set feeAmountRaw from the fresh yield with no minimum check and appended the transfer whenever feeAmountRaw > 0 – so a sub-minimum ("dust") recomputed fee emitted a treasury transfer the other three paths suppress, and the returned feeAmountRaw disagreed with the build path. Fix: guarded the recompute – feeAmountRaw = dFreshFee >= UNFLAT_MIN_FEE_USDC ? (raw units) : 0. Below the floor → 0 → no transfer, matching the build and covered paths. Kept minimal (no four-way helper extraction) since the build paths also need dFee for display/userReceives. Tests: a $0.000004 fresh yield → sub-floor fee → feeAmountRaw 0, no transfer appended; an above-minimum $3 yield → $0.45 fee with the transfer. The #11/#8/#12 fee suites stay green. Full suite 97 suites / 1787 green, tsc + ESLint clean.

Issue L-13: secureStorage migration can leave AsyncStorage plaintext after SecureStore success [RESOLVED]

Source: https://github.com/sherlock-audit/2026-07-unflat-july-9th-2026/issues/26

Summary

`getSecureItem` migrates legacy AsyncStorage values into SecureStore and then deletes the AsyncStorage copy. If the process dies between a successful `setItemAsync` and `removeItem`, the next read returns the SecureStore value first and never retries cleanup. The plaintext legacy copy can remain indefinitely. `setSecureItem` always cleans AsyncStorage. The read path does not.

Vulnerability Detail

In `src/services/secureStorage.ts:28-39`.

Impact

Sensitive values that use this API (for example offramp expected-address keys) may remain in less-protected storage after migration appears complete.

Code Snippet

Tool Used

Manual Review

Recommendation

On every SecureStore hit, best-effort delete any AsyncStorage key of the same name. On migration, schedule a retry cleanup if remove fails.

Discussion

alexanderpi007

Fixed in 3504d31b. Root cause: getSecureItem returned early on a SecureStore hit and never retried the AsyncStorage cleanup, so a migration that died between setItemAsync and removeItem left the plaintext legacy copy indefinitely; the migration removeItem was also unguarded (a cleanup failure threw the read). Fix: best-effort AsyncStorage.removeItem on every SecureStore hit (idempotent, self-healing), and the

migration cleanup is best-effort too. Tests: a both-stores value is scrubbed from AsyncStorage on read; an on-hit cleanup failure doesn't fail the read; a migration cleanup failure still returns the migrated value. Unit-tested.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.