hexens x Elfomo Labs

Security Review Report for ElfomoFi

April 2026

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit focuses on the vault accounting layer of the elfomoFi protocol. The design keeps the manager generic and lightweight, while delegating asset accounting and rate logic to individual vault implementations. Currently, only EpochBasedVault is implemented, but additional vault types are expected in the future.

The review was conducted over one week and included a comprehensive analysis of all smart contracts in the repository.

No high- or medium-severity issues were identified. We reported three low-severity issues and one informational finding.

All findings were either fixed or acknowledged by the development team and subsequently verified by our auditors.

Following remediation, we conclude that the protocol's overall security posture and code quality have improved.

3. Security Review Details

▪ Review Led by

Trung Dinh, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/ElfomoFi/elfomofi-vaults/tree/0d1c66ed40a87d6bf7cb0c18b3a2e9cf1665f3d8

The issues described in this report were fixed in the following commit:

🔗 https://github.com/ElfomoFi/elfomofi-vaults/tree/36b49cce7e7bddd322a80b7bd020a9451836e44e

▪ Changelog

6 April 2026	Audit start
13 April 2026	Initial report
14 April 2026	Revision received
15 April 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

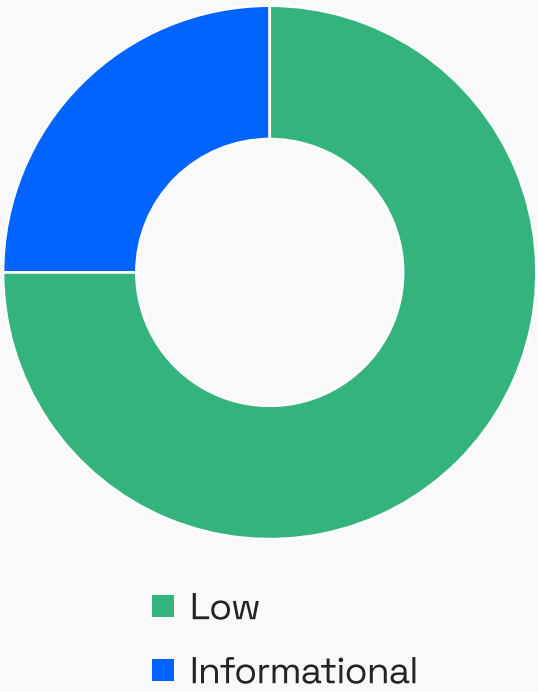
- **Issue Symbolic Codes**

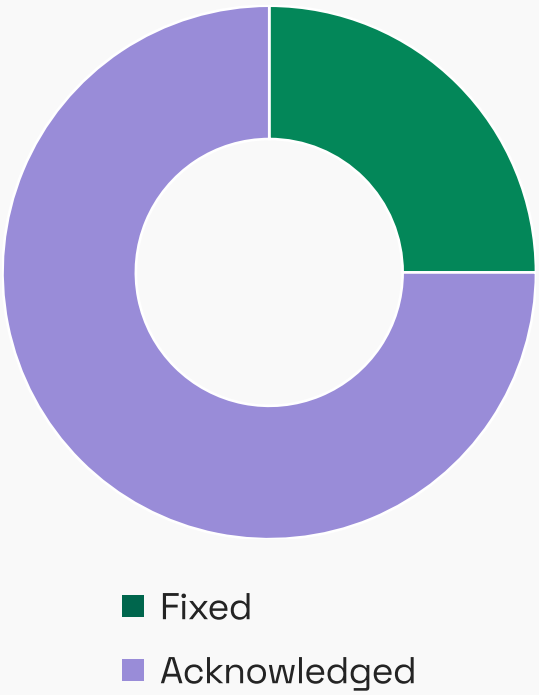
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	0
Low	3
Informational	1
Total:	**4**





6. Weaknesses

This section contains the list of discovered weaknesses.

ELFI-2 | Incorrect grossVirtualSupplyX18 Reset When Withdrawal Equals currentTotalAssetsInRef

Fixed

Severity:	Low	Probability:	Rare	Impact:	Medium

Path:

src/vaults/EpochBasedVault.sol#L221-L226

Description:

In **_applyWithdrawalAccounting()**, when the (capped) withdrawal amount equals **currentTotalAssetsInRef**, both **currentTotalAssetsInRef** and **grossVirtualSupplyX18** are reset to zero:

```
function _applyWithdrawalAccounting(uint256 refAmount) internal {
    if (refAmount == 0 || currentTotalAssetsInRef == 0) {
        return;
    }

    uint256 assetsOut = refAmount > currentTotalAssetsInRef
                ? currentTotalAssetsInRef
                : refAmount;

    if (assetsOut == currentTotalAssetsInRef) {
        currentTotalAssetsInRef = 0;
        grossVirtualSupplyX18 = 0;   // @audit wiped even if LP tokens remain
        return;
    }
    // ...
}
```

This logic basically assumes that if all tracked assets are drained, then all LP shares must have been redeemed too. But that's not always true. In practice, **currentTotalAssetsInRef** is just the last recorded value before the epoch update, and it can be different from the **totalAssetsInRef**

passed into **settleEpoch()**.

As a result, **grossVirtualSupplyX18** can be reset to zero while LP tokens are still in circulation, leading to incorrect gross rate calculations.

For Example with zero fees, the gross rate should always equal the net rate.

1. Initial state:
   - **totalSupply = 1000**
   - **currentTotalAssetsInRef = 1000**
   - **currentNetRate = 1e18**
   - **grossVirtualSupplyX18 = 1000e18**
2. User requests withdrawal of 900 LP tokens at rate 1.0
   - **refAmount = 900**
3. A loss epoch is settled (no withdrawals processed)
   - **currentTotalAssetsInRef = 900**
4. Strategy recovers to 1000 assets. Next epoch settles the withdrawal with **fillAmount = 900**, passing **totalAssetsInRef = 100** (i.e., 1000 - 900):
   - **_applyWithdrawalAccounting(900)**
     → **assetsOut = min(900, 900) = 900**
     → equals **currentTotalAssetsInRef** → triggers full reset
     → **grossVirtualSupplyX18 = 0**
   - **fillWithdrawRequest** burns 900 LP tokens
     → **totalSupply = 100**

   At this point, 100 LP tokens still exist, but **grossVirtualSupplyX18 = 0**.
5. A new deposit of 100 arrives
   - **_applyDepositAccounting()** detects **grossVirtualSupplyX18 = 0**, falls back to the net rate (1.0), and only accounts for the new deposit:
     - **totalSupply = 200**
     - **currentTotalAssetsInRef = 200**
     - **grossVirtualSupplyX18 = 100e18**

As the result:

- **netRate = 200 * 1e18 / 200 = 1.0e18**
- **grossRate = 200 * 1e36 / 100e18 = 2.0e18**

This creates a 2x divergence in a zero-fee system, where gross and net rates should be identical.

Remediation:

Consider reverting the withdrawal transaction when **refAmount > currentTotalAssetsInRef** as the withdrawn amount shouldn't exceed the total asset.

Proof of Concept:

Place the following test in **test/hardhat/vaults/VaultManager.epoch.test.ts** file.

```
it("HEXENS_02", async function () {
    const ctx = await deployVaultFixture();

    // Step 1: 1000 LP tokens, totalAssets = 1000, rate = 1.0
    await ctx.makeDeposit(parseUnits("1000", 6));

    await ctx.refToken.mint(ctx.ownerAddress, parseUnits("10000000", 6));
    await ctx.refToken.connect(ctx.owner).approve(ctx.epochVault.address, ethers.constants.MaxUint256);

    // Step 2: User requests withdrawal of 900 LP tokens at rate 1.0 → refAmount = 900
    const requestId = await ctx.makeWithdrawRequest(
      parseUnits("900", 6),   // lpAmount
      parseUnits("900", 6),   // targetRefAmount (rate 1.0)
      parseUnits("1", 6)      // minRefAmount (low to allow fill at smaller amount)
    );

    // Step 3: Settle a small loss epoch — totalAssets drops to 900, NO withdrawal processing
    // (settlement was prepared before the withdraw request)
    // Rate change: 1.0 → 0.9 (10% drop, within 100% max)
    await ctx.epochVault.connect(ctx.owner).settleEpoch(parseUnits("900", 6), []);

    // State: currentTotalAssetsInRef = 900, totalSupply = 1000 (900 escrowed + 100 held)
    expect(await ctx.epochVault.currentTotalAssetsInRef()).to.equal(parseUnits("900", 6));
    expect(await ctx.shares.totalSupply()).to.equal(parseUnits("1000", 6));

    // Step 4: Profit brings actual assets to 1100
    // Step 5: Settle epoch, processing the withdrawal with refAmount = 900
    //   (equals currentTotalAssetsInRef exactly)
    //
    // In _applyWithdrawalAccounting(900):
    //   assetsOut = min(900, 900) = 900 == currentTotalAssetsInRef → FULL RESET!
    //   currentTotalAssetsInRef = 0, grossVirtualSupplyX18 = 0
    //
    // Then fillWithdrawRequest burns 900 LP tokens → totalSupply = 100
    // But grossVirtualSupplyX18 = 0 while 100 LP tokens still exist!
    //
    // Profit brings assets back to 1000.
    // totalAssetsInRef passed = 1000 - 900 = 100 (remaining assets after withdrawal)
    // Rate change: 0.9 → 1.0 (~11%, within 100% max) ✓
    const fillAmount = parseUnits("900", 6);
    const remainingAssets = parseUnits("100", 6);
```

```
await ctx.epochVault.connect(ctx.owner).settleEpoch(remainingAssets, [
  { requestId, refAmount: fillAmount },
]);

// State after settlement:
const totalSupply = await ctx.shares.totalSupply();
const grossSupply = await ctx.epochVault.grossVirtualSupplyPrecise();
const totalAssets = await ctx.epochVault.currentTotalAssetsInRef();

expect(totalSupply).to.equal(parseUnits("100", 6));       // 100 LP tokens remain
expect(totalAssets).to.equal(remainingAssets);             // 100 assets remain

// BUG: grossVirtualSupplyX18 was reset to 0 despite 100 LP tokens still existing.
// With no fees, grossVirtualSupplyX18 should equal totalSupply * 1e18.
expect(grossSupply).to.equal(0); // <-- This is the bug! Should NOT be 0

// The gross rate currently uses the fallback (= netRate) so they appear equal...
// But a new deposit will break the invariant:

// Step 6: New deposit of 100 to demonstrate gross/net rate divergence
await ctx.makeDeposit(parseUnits("100", 6));

const finalTotalSupply = await ctx.shares.totalSupply();
const finalGrossSupply = await ctx.epochVault.grossVirtualSupplyPrecise();
const finalTotalAssets = await ctx.epochVault.currentTotalAssetsInRef();

// totalSupply = 200 (100 old + 100 new at rate 1.0), totalAssets = 200
expect(finalTotalSupply).to.equal(parseUnits("200", 6));
expect(finalTotalAssets).to.equal(parseUnits("200", 6));

// _applyDepositAccounting(100):
//   grossRate = _getCurrentGrossRate() → grossVirtualSupplyX18=0, fallback=currentNetRate=1e18
//   grossVirtualSupplyX18 += 100 * 1e36 / 1e18 = 100e18
//   grossVirtualSupplyX18 = 100e18 (only accounts for the new deposit, missing the old 100 LP!)

// netRate  = 200 * 1e18 / 200 = 1.0e18
// grossRate = 200 * 1e36 / 100e18 = 2.0e18
//
// With no fees: grossRate MUST equal netRate, but 2.0 != 1.0
const netRate = finalTotalAssets.mul(RATE_PRECISION).div(finalTotalSupply);
const grossRate = finalTotalAssets
  .mul(RATE_PRECISION)
  .mul(ethers.constants.WeiPerEther)
  .div(finalGrossSupply);

console.log("netRate:  ", netRate.toString());
```

```
  console.log("grossRate:", grossRate.toString());

  // BUG DEMONSTRATED: grossRate != netRate with zero fees
  // grossRate is 2x the netRate because grossVirtualSupplyX18 lost track of 100 LP tokens
  expect(grossRate).to.not.equal(netRate);

  // Step 7: Another deposit of 100 — grossVirtualSupplyX18 grows differently than step 6
  // because the corrupted grossRate (2.0) is now used to price the new gross shares.
  const grossSupplyBeforeStep7 = finalGrossSupply; // 100e18
  await ctx.makeDeposit(parseUnits("100", 6));

  const step7TotalSupply = await ctx.shares.totalSupply();
  const step7GrossSupply = await ctx.epochVault.grossVirtualSupplyPrecise();
  const step7TotalAssets = await ctx.epochVault.currentTotalAssetsInRef();

  // totalSupply = 300, totalAssets = 300
  expect(step7TotalSupply).to.equal(parseUnits("300", 6));
  expect(step7TotalAssets).to.equal(parseUnits("300", 6));

  // _applyDepositAccounting(100):
  //   grossRate = _getCurrentGrossRate() = 200 * 1e36 / 100e18 = 2.0e18 (corrupted!)
  //   grossVirtualSupplyX18 += 100 * 1e36 / 2.0e18 = 50e18
  //   grossVirtualSupplyX18 = 100e18 + 50e18 = 150e18
  const grossSharesAddedStep7 = step7GrossSupply.sub(grossSupplyBeforeStep7);
  const grossSharesAddedStep6 = grossSupplyBeforeStep7; // 100e18

  // Step 6 added 100e18 gross shares for 100 deposit (at fallback rate 1.0)
  // Step 7 added  50e18 gross shares for 100 deposit (at corrupted rate 2.0)
  // Same deposit amount, different gross shares — the accounting is inconsistent
  console.log("grossShares added step 6:", grossSharesAddedStep6.toString());
  console.log("grossShares added step 7:", grossSharesAddedStep7.toString());
  expect(grossSharesAddedStep6).to.not.equal(grossSharesAddedStep7);

  // With no fees and identical deposit amounts at the same netRate (1.0),
  // both deposits should have added the same gross shares.
});
```

ELFI-4 | Pending withdrawal requests can be kept open and later filled at a stale capped amount

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/vaults/VaultsManager.sol#L78

Description:

A withdrawal request stores a fixed **refAmount** at request time, based on the current vault rate. Later, **fillWithdrawRequest()** only checks that the fill stays within the stored min/max bounds and does not reprice the request using the current rate. The request itself has no expiry, and the user cannot cancel it directly.

```
function requestWithdraw(
    uint256 vaultId,
    uint256 lpAmount,
    uint256 targetRefAmount,
    uint256 minRefAmount,
    uint256 expiry,
    bytes calldata signature
) external nonReentrant returns (uint256 requestId) {
    Vault storage vault = _getVault(vaultId);
    require(vault.state != VaultState.Paused, InvalidVaultState());
    require(lpAmount > 0, InvalidAmount());

    bytes32 digest = _consumeWithdrawSignature(
        msg.sender, vaultId, lpAmount, targetRefAmount, minRefAmount, expiry, signature
    );

    uint256 rate = IVault(vault.vaultAddress).getRate();
    require(rate > 0, InvalidRate());
    uint256 refByRate = Math.mulDiv(lpAmount, rate, RATE_PRECISION);
    uint256 refAmount = Math.min(targetRefAmount, refByRate);
    require(refAmount >= minRefAmount && refAmount > 0, InvalidAmount());

    IERC20(vault.lpTokenAddress).safeTransferFrom(msg.sender, address(this), lpAmount);

    requestId = nextRequestId++;
```

```
        requests[requestId] = WithdrawRequest({
            vaultId: vaultId,
            user: msg.sender,
            lpAmount: lpAmount,
            refAmount: refAmount,
            minRefAmount: minRefAmount,
            status: RequestStatus.Pending
        });

        emit WithdrawRequested(
            requestId, vaultId, msg.sender, lpAmount, targetRefAmount, refAmount, minRefAmount, digest
        );
    }
```

If the vault rate rises while the request remains pending, the user can still be filled only at the older request-time cap.

Remediation

Add an on-chain expiry or timeout to withdrawal requests and allow stale unfilled requests to be canceled through the request lifecycle. The fill path should also avoid relying indefinitely on a request-time capped amount when the current vault rate has moved.

ELFI-5 | Incomplete validation of lpTokenAddress when creating a new vault

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

src/vaults/manager/VaultsManagerAdmin.sol#L13-L43

Description:

The **createVault()** function sets the **lpTokenAddress** for a vault. This address is used for minting and burning shares within the **VaultsManager** contract during vault operations.

However, the system does not validate whether the **lpTokenAddress** provided in **createVault()** matches the actual **vault.lpTokenAddress()** variable. If these addresses mismatch, settlements and share pricing will fail because the vault will use an incorrect supply to calculate rates.

Additionally, there is no uniqueness check for **lpTokenAddress** in the **createVault()** function. If two vaults are created with the same **lpTokenAddress**, the share calculations for both vaults will be broken.

```
function createVault(
    ...
) external onlyOwner {
    require(vaultId != 0 && vaultId != type(uint256).max, InvalidAmount());
    require(vaults[vaultId].vaultAddress == address(0), VaultAlreadyExists(vaultId));
    require(
        vaultAddress != address(0)
            && tradingVaultAddress != address(0)
            && curatorAddress != address(0)
            && lpTokenAddress != address(0),
        InvalidAddress()
    );
    require(vaultIdByAddress[vaultAddress] == 0, VaultAddressAlreadyUsed(vaultAddress));
    require(maxLpDepositCap > 0, InvalidAmount());
    require(uint256(protocolFeeMilliBps) + uint256(curatorFeeMilliBps) <= MILLI_BPS_DENOMINATOR,
InvalidFee());
    require(
        IERC20Metadata(lpTokenAddress).decimals() == IERC20Metadata(address(refToken)).decimals(),
        InvalidLpTokenDecimals()
    );
```

```
    uint256 rate = IVault(vaultAddress).getRate();
    require(rate > 0, InvalidRate());

    vaults[vaultId] = Vault({
        vaultAddress: vaultAddress,
        tradingVaultAddress: tradingVaultAddress,
        curatorId: curatorId,
        protocolFeeMilliBps: protocolFeeMilliBps,
        curatorFeeMilliBps: curatorFeeMilliBps,
        curatorAddress: curatorAddress,
        lpTokenAddress: lpTokenAddress,
        state: VaultState.Paused,
        isEnabledForSwaps: isEnabledForSwaps,
        highWaterMarkRate: rate,
        lastUpdatedRate: rate,
        lastUpdatedPreFeeRate: rate,
        lastUpdatedTimestamp: block.timestamp,
        maxLpDepositCap: maxLpDepositCap
    });
    vaultIdByAddress[vaultAddress] = vaultId;

    emit VaultCreated(vaultId, vaultAddress, lpTokenAddress, curatorId);
}
```

Remediation

Add validation checks for LP token address consistency and uniqueness in **createVault()**.

ELFI-3 | Incompatibility with Fee-on-Transfer Tokens

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/vaults/VaultsManager.sol#L43-L53

Description:

The current implementation of **VaultsManager.deposit()** assumes that the full amount specified by the user is received by the contract. This does not hold for fee-on-transfer tokens, which deduct a fee during transfers. Specifically:

- **In deposit()**, LP tokens are minted based on the input **refAmount**, even though the actual amount received may be lower.

This mismatch can break accounting assumptions and lead to inconsistencies in the vault's state, especially if a fee-on-transfer token is used, whether intentionally or by mistake.

```
uint256 lpByRate = Math.mulDiv(refAmount, RATE_PRECISION, rate);
// If the backend caps LP output below the current onchain quote, the excess ref amount stays in the
// vault and benefits existing LPs instead of becoming protocol revenue.
lpAmount = Math.min(targetLpAmount, lpByRate);
require(lpAmount >= minLpAmount && lpAmount > 0, InvalidAmount());

uint256 lpSupply = IVaultShares(lpTokenAddress).totalSupply();
// Fee share minting can push total supply above the cap; the cap only constrains user deposit minting.
require(lpSupply + lpAmount <= vault.maxLpDepositCap, ExceedsLpDepositCap());

refToken.safeTransferFrom(msg.sender, vaultAddress, refAmount);
```

Remediation

To mitigate this issue, consider implementing a balance check before and after the transfer to determine the actual amount transferred.

hexens x Elfomo Labs