SHERLOCK

Security Review For Evro Finance

Collaborative Audit Prepared For: **Evro Finance**
Lead Security Expert(s): **0x37**
0x73696d616f
Date Audited: **December 26 - December 30, 2025**

Introduction

EVRO is an open-source protocol deployed on Gnosis that enables you to CREATE, MANAGE AND BALANCE COLLATERALIZED VAULTS without issuers, custodians, or intermediaries. Built on Liquity V2 principles, EVRO operates through fully decentralized smart contracts with no administrative control and no upgrade authority.

Scope

Repository: evro-finance/evro

Audited Commit: edb1697daa450769bc92223bc25bc1e0b2fba70e

Final Commit: b74314835fdee796dcf5d8fda9bd5dc71fcb62ad

Files:

- contracts/src/CoGNO.sol
- contracts/src/Dependencies/Constants.sol
- contracts/src/Dependencies/WBTCWrapper.sol
- contracts/src/Interfaces/IWBTCWrapper.sol
- contracts/src/PriceFeeds/GNOPriceFeed.sol
- contracts/src/PriceFeeds/MainnetPriceFeedBase.sol
- contracts/src/PriceFeeds/OSGNOPriceFeed.sol
- contracts/src/PriceFeeds/SDAIPriceFeed.sol
- contracts/src/PriceFeeds/WBTCPriceFeed.sol
- contracts/src/PriceFeeds/WETHPriceFeed.sol
- contracts/src/PriceFeeds/WSTETHPriceFeed.sol
- contracts/src/TroveNFT.sol
- contracts/src/Zappers/Interfaces/IWBTCZapper.sol
- contracts/src/Zappers/Modules/Exchanges/Curve/ICurveStableswapNGPool.sol
- contracts/src/Zappers/Modules/Exchanges/CurveNGExchange.sol
- contracts/src/Zappers/WBTCZapper.sol

Final Commit Hash

b74314835fdee796dcf5d8fda9bd5dc71fcb62ad

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
3	5	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: SDAIPriceFeed is vulnerable to donation attack via manipulatable vault rate [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/18

Summary

The `SDAIPriceFeed` uses the instantaneous vault rate from `convertToAssets()` which can be manipulated through donation attacks, allowing attackers to drain the protocol.

Vulnerability Detail

The `SDAIPriceFeed` calculates the sDAI price using the ERC4626 vault's `convertToAssets()` function, which returns the current exchange rate between sDAI shares and DAI assets. This rate can be manipulated by donating assets directly to the vault.

Attack scenario:

1. Attacker deposits 500 sDAI when vault has 500 sDAI shares and 500 DAI (1:1 rate)
2. Attacker donates 9000 DAI directly to the vault
3. Now 1000 sDAI shares = 10000 DAI (1:10 rate)
4. Attacker opens a trove with 1000 sDAI and borrows 10000 EVRO at the inflated rate
5. Attacker redeems the 10000 EVRO against other collaterals, draining the protocol

Impact

An attacker can manipulate the sDAI price to borrow significantly more EVRO than the collateral is worth, then redeem against other collaterals to drain the protocol.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/PriceFeeds/SDAIPriceFeed.sol#L47

```
function _fetchPricePrimary() internal returns (uint256, bool) {
    assert(priceSource == PriceSource.primary);

    (uint256 daiUSDPrice, bool daiEurOracleDown) = _getOracleAnswer(ethUsdOracle);
    (uint256 eurUsdPrice, bool eurUsdOracleDown) = _getOracleAnswer(eurUsdOracle);

    // Get the DAI rate of the SDAI vault
    uint256 sdaiDaiRate = sdai.convertToAssets(1e18);  // @audit - manipulatable!
```

```
    // ...

    uint256 sdaiUSDPrice = FixedPointMathLib.mulWadUp(daiUSDPrice, sdaiDaiRate);
    lastGoodPrice = FixedPointMathLib.divWad(sdaiUSDPrice, eurUsdPrice);

    return (lastGoodPrice, false);
}
```

Tool Used

Manual Review

Recommendation

Use the max price between an oracle rate and the instantaneous rate for redemptions, and the min price for calculating ICR when opening/adjusting troves:

```
function _fetchPricePrimary(bool _isRedemption) internal returns (uint256, bool) {
    uint256 sdaiDaiRate = sdai.convertToAssets(1e18);
    (uint256 oracleRate, bool oracleDown) = _getOracleAnswer(sdaiOracle);

    uint256 rateToUse;
    if (_isRedemption) {
        rateToUse = LiquityMath._max(sdaiDaiRate, oracleRate);
    } else {
        rateToUse = LiquityMath._min(sdaiDaiRate, oracleRate);
    }
    // ... rest of calculation
}
```

Issue H-2: WSTETHPriceFeed returns USD price Instead of EUR on oracle failure [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/19

Summary

When the stETH-USD oracle fails, `WSTETHPriceFeed` switches to `ETHUSDxCanonical` mode which returns prices in USD instead of EUR, causing incorrect price calculations.

Vulnerability Detail

The `WSTETHPriceFeed` has a fallback mechanism when the stETH-USD oracle fails. It calls `_shutDownAndSwitchToETHUSDxCanonical()` which calculates the price using ETH-USD and the canonical wstETH rate. However, the `_fetchPriceETHUSDxCanonical()` function in `CompositePriceFeed` never converts the USD price to EUR before returning.

This means after the stETH-USD oracle failure:

1. All prices returned are in USD (~1.05x higher than EUR)
2. The `lastGoodPrice` gets corrupted with a USD value
3. All subsequent operations use the wrong price denomination

Impact

After stETH-USD oracle failure, all wstETH collateral operations will use incorrect prices. Users could be incorrectly liquidated or be able to borrow too much. The corrupted `lastGoodPrice` persists even after further fallbacks.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/PriceFeeds/WSTETHPriceFeed.sol#L70-L72

```
// If the STETH-USD feed is down, shut down and try to substitute it with the
↪  ETH-USD price
if (stEthUsdOracleDown) {
    return
    ↪  (_shutDownAndSwitchToETHUSDxCanonical(address(stEthUsdOracle.aggregator),
    ↪  ethUsdPrice), true);
}
```

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/PriceFeeds/CompositePriceFeed.sol#L76-L95

```
function _fetchPriceETHUSDxCanonical(uint256 _ethUsdPrice) internal returns
↪  (uint256) {
    assert(priceSource == PriceSource.ETHUSDxCanonical);
    (uint256 lstRate, bool exchangeRateIsDown) = _getCanonicalRate();

    if (exchangeRateIsDown) {
        priceSource = PriceSource.lastGoodPrice;
        return lastGoodPrice;
    }

    // Calculate the canonical LST-USD price: USD_per_LST = USD_per_ETH *
    ↪  underlying_per_LST
    uint256 lstUsdCanonicalPrice = _ethUsdPrice * lstRate / 1e18;  // @audit - USD,
    ↪  not EUR!

    uint256 bestPrice = LiquityMath._min(lstUsdCanonicalPrice, lastGoodPrice);

    lastGoodPrice = bestPrice;  // @audit - corrupts lastGoodPrice with USD value

    return bestPrice;  // @audit - returns USD instead of EUR
}
```

Tool Used

Manual Review

Recommendation

Convert the USD price to EUR in the `_fetchPriceETHUSDxCanonical()` function or override it in `WSTETHPriceFeed` to include EUR conversion:

```
function _fetchPriceETHUSDxCanonical(uint256 _ethUsdPrice) internal override
↪  returns (uint256) {
    (uint256 lstRate, bool exchangeRateIsDown) = _getCanonicalRate();
    (uint256 eurUsdPrice, bool eurUsdOracleDown) = _getOracleAnswer(eurUsdOracle);

    if (exchangeRateIsDown || eurUsdOracleDown) {
        priceSource = PriceSource.lastGoodPrice;
        return lastGoodPrice;
    }

    uint256 lstUsdCanonicalPrice = _ethUsdPrice * lstRate / 1e18;
    uint256 lstEurCanonicalPrice = FixedPointMathLib.divWad(lstUsdCanonicalPrice,
    ↪  eurUsdPrice);

    uint256 bestPrice = LiquityMath._min(lstEurCanonicalPrice, lastGoodPrice);
```

```
    lastGoodPrice = bestPrice;

    return bestPrice;
}
```

Issue H-3: Critical Parameter Mismatch in Liquidation Gas Compensation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/27

Summary

The project, a fork of Liquity v2 (Bold), incorrectly retains the constant ETH_GAS_COMPENSATION = 0.0375 ether. While this value was designed for ETH (0.0375 ETH around 90$), in this evro project the gas token is xDAI. Consequently, the compensation is valued at only 0.0375 DAI, which is insufficient to cover transaction gas costs, leading to a total failure of the liquidation incentive mechanism.

Vulnerability Detail

In Evro, the gas token is xDAI. When one unhealthy trove is liquidated, around 0.0375 xDAI will be transferred to the liquidator.

When one unhealthy trove is liquidated, the liquidator can gain two kinds of profits:

1. coll compenstaion.
2. gas compenstaion.

If there is less bold token in SP, the coll compensation will be low, even zero. In this case, the liquidator's profit will be `gas compensation - gas cost`.

It's probable that the gas compensation cannot cover the gas cost. Even if the gas compensation can cover the gas cost, the profit will be quite small, this will not provide enough incentive to liquidators. This will cause that some unhealthy positions cannot be liquidated timely.

```
// Amount of xDAI to be locked in gas pool on opening troves
@> uint256 constant ETH_GAS_COMPENSATION = 0.0375 ether;
```

Impact

Unhealthy positions cannot be liquidated timely.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/e047c2c73d7b208974b561012ac97e6cf90bf7ff/evro/contracts/src/Dependencies/Constants.sol#L13

Tool Used

Manual Review

Recommendation

Adjust the ETH_GAS_COMPENSATION to provide enough incentive to liquidators.

Discussion

MrDeadCe11

I figured this was adequate because of the low gas costs on gnosis. with gas costing around .001 gwei most of the time.

Issue M-1: WBTCZapper decimal mismatch causes revert [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/20

Summary

In `closeTroveFromCollateral()`, the calculation `trove.entireColl - _params.flashLoanAmount` mixes 18-decimal wWBTC with 8-decimal WBTC values, causing either transaction revert or bypassed slippage protection.

Vulnerability Detail

The `closeTroveFromCollateral()` function calculates `collLeft` by subtracting `flashLoanAmount` (8 decimals, WBTC) from `trove.entireColl` (18 decimals, wWBTC). This decimal mismatch causes:

1. If `entireColl` is e.g. `1e18` (1 wWBTC) and `flashLoanAmount` is `1e8` (1 WBTC), then `collLeft = 1e18 - 1e8  1e18` - almost the entire collateral
2. The `minExpectedCollateral` check becomes meaningless as `collLeft` is always near `entireColl`
3. When sending `collLeft` to the user, the transaction will revert because the contract doesn't have that much wWBTC

Impact

Users cannot close their troves using the flash loan mechanism, or if the function somehow proceeds, the slippage protection is bypassed.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/Zappers/WBTCZapper.sol#L340

```
function closeTroveFromCollateral(CloseTroveFromCollateralParams calldata _params)
↪  external {
    // ...

    LatestTroveData memory trove = troveManager.getLatestTroveData(_params.troveId);

    // @audit - trove.entireColl is in 18 decimals (wWBTC)
    // @audit - _params.flashLoanAmount is in 8 decimals (WBTC)
    uint256 collLeft = trove.entireColl - _params.flashLoanAmount;  // WRONG!
```

```
    require(collLeft >= _params.minExpectedCollateral, "Insufficient collateral");

    // ...
}
```

Tool Used

Manual Review

Recommendation

Convert `flashLoanAmount` to 18 decimals before subtraction:

```
function closeTroveFromCollateral(CloseTroveFromCollateralParams calldata _params)
 ↪  external {
    // ...

    LatestTroveData memory trove = troveManager.getLatestTroveData(_params.troveId);

    // Convert flashLoanAmount from 8 decimals to 18 decimals
    uint256 flashLoanAmountWrapped = _params.flashLoanAmount * 1e10;
    uint256 collLeft = trove.entireColl - flashLoanAmountWrapped;

    require(collLeft >= _params.minExpectedCollateral, "Insufficient collateral");

    // ...
}
```

Issue M-2: WBTCZapper approves wrong token for exchange [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/21

Summary

The `WBTCZapper` constructor approves wWBTC (wrapped WBTC) for the exchange, but the Curve pool uses WBTC directly, causing swap transactions to fail.

Vulnerability Detail

In the constructor, the zapper approves wWBTC to the exchange contract. However, the Curve NG pool on Gnosis holds WBTC (8 decimals), not wWBTC (18 decimals). When `closeTroveFromCollateral()` tries to swap collateral for EVRO, it will fail because WBTC is not approved.

Impact

The `closeTroveFromCollateral()` function will fail when trying to swap WBTC for EVRO, preventing users from closing their troves using the flash loan mechanism.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/Zappers/WBTCZapper.sol#L29

```
constructor(IAddressesRegistry _addressesRegistry, IFlashLoanProvider
↪  _flashLoanProvider, IExchange _exchange)
    BaseZapper(_addressesRegistry, _flashLoanProvider, _exchange)
{
    // Approve wWBTC to exchange
    collToken.approve(address(_exchange), type(uint256).max);  // @audit - wrong
    ↪  token!
}
```

Tool Used

Manual Review

Recommendation

Approve WBTC instead of wWBTC for the exchange:

```
constructor(IAddressesRegistry _addressesRegistry, IFlashLoanProvider
↪  _flashLoanProvider, IExchange _exchange)
    BaseZapper(_addressesRegistry, _flashLoanProvider, _exchange)
{
    // Approve WBTC to exchange (for closeTroveFromCollateral)
    IWBTC(wbtcWrapper.WBTC()).approve(address(_exchange), type(uint256).max);

    // Approve wWBTC for borrowerOperations (existing functionality)
    collToken.approve(address(borrowerOperations), type(uint256).max);
}
```

Issue M-3: WBTCZapper precision loss will lead to reverts [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/24

Summary

When users input collateral amounts that are not evenly divisible by 1e10, the WBTC to wWBTC conversion causes precision loss that may result in transaction reverts.

Vulnerability Detail

The WBTCWrapper converts between 8-decimal WBTC and 18-decimal wWBTC by multiplying/dividing by 1e10. When a user provides `collAmount` that is not divisible by 1e10:

1. User deposits 1.5e10 wWBTC worth of collateral
2. Actual WBTC transferred = 1.5e10 / 1e10 = 1 WBTC (truncated)
3. Zapper wraps 1 WBTC → 1e10 wWBTC
4. Zapper tries to use `collAmount` (1.5e10) but only has 1e10 wWBTC
5. Transaction reverts

Impact

Users must ensure their input amounts are divisible by 1e10 or transactions will revert.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/Zappers/WBTCZapper.sol#L45

```
function openTroveWithWBTC(OpenTroveParams calldata _params) external {
    // User provides _params.collAmount in wWBTC (18 decimals)
    // But WBTC transfer truncates to 8 decimals
    IWBTC(wbtcWrapper.WBTC()).transferFrom(msg.sender, address(this),
    ↪  _params.collAmount / 1e10);

    // Wrap back to wWBTC - may have less than _params.collAmount
    wbtcWrapper.depositFor(address(this), _params.collAmount / 1e10);

    // This may revert if there's not enough wWBTC
    borrowerOperations.openTrove(..., _params.collAmount, ...);
}
```

Tool Used

Manual Review

Recommendation

Add input validation or round up the wrapped amount:

```
function openTroveWithWBTC(OpenTroveParams calldata _params) external {
    require(_params.collAmount % 1e10 == 0, "Amount must be divisible by 1e10");
    // ... rest of function
}
```

Or round up to handle dust:

```
uint256 wbtcAmount = _params.collAmount / 1e10;
uint256 actualWrappedAmount = wbtcAmount * 1e10;
// Use actualWrappedAmount instead of _params.collAmount
```

Issue M-4: API3 oracle future timestamp causes temporary DoS via underflow [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/26

Summary

API3 oracles can return timestamps up to 1 hour in the future, causing an arithmetic underflow in the staleness check that temporarily DoS's the price feed.

Vulnerability Detail

The API3 DataFeedServer allows timestamps to be up to 1 hour in the future:

```
// From API3 DataFeedServer.sol
require(
    timestamp < block.timestamp + 1 hours,
    "Timestamp not valid"
);
```

However, `MainnetPriceFeedBase._isValidChainlinkPrice()` is as follows:

```
return chainlinkResponse.success && block.timestamp - chainlinkResponse.timestamp <
↪  _stalenessThreshold
    && chainlinkResponse.answer > 0;
```

When `chainlinkResponse.timestamp > block.timestamp`, the subtraction `block.timestamp - chainlinkResponse.timestamp` underflows in Solidity 0.8.x, causing a revert. This doesn't trigger a shutdown (since it reverts before returning false), but it does DoS the price feed until `block.timestamp` catches up to the oracle timestamp.

Impact

Price feed operations (borrowing, redemptions, liquidations) are temporarily DoS'd for up to 1 hour when API3 returns a future timestamp. While not a permanent issue, this could prevent time-sensitive liquidations during volatile market conditions.

Code Snippet

https://github.com/api3dao/contracts/blob/main/contracts/api3-server-v1/DataFeedServer.sol#L30-L38

```
function _updateDapiWithBeacons(
    bytes32[] memory beaconIds,
```

```
    int224[] memory values,
    uint32[] memory timestamps
) internal returns (int224 updatedValue, uint32 updatedTimestamp) {
    // ...
    require(
        timestamp < block.timestamp + 1 hours,  // @audit - allows future timestamps
        "Timestamp not valid"
    );
}
```

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/PriceFeeds/MainnetPriceFeedBase.sol#L114

```
function _isValidChainlinkPrice(ChainlinkResponse memory chainlinkResponse, uint256
↪  _stalenessThreshold)
    internal
    view
    returns (bool)
{
    return chainlinkResponse.success && block.timestamp -
    ↪  chainlinkResponse.timestamp < _stalenessThreshold  // @audit - underflows
    ↪  if timestamp > block.timestamp
        && chainlinkResponse.answer > 0;
}
```

Tool Used

Manual Review

Recommendation

See their docs. Handle future timestamps gracefully by checking before subtraction:

```
function _isValidChainlinkPrice(ChainlinkResponse memory chainlinkResponse, uint256
↪  _stalenessThreshold)
    internal
    view
    returns (bool)
{
    if (!chainlinkResponse.success || chainlinkResponse.answer <= 0) return false;

    // Handle future timestamps from API3 (allowed up to 1 hour ahead)
    if (chainlinkResponse.timestamp > block.timestamp) return true;

    return block.timestamp - chainlinkResponse.timestamp < _stalenessThreshold;
}
```

Issue M-5: Missing min/max price check [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/28

Summary

Missing min/max price check

Vulnerability Detail

In Evro, most token prices are fetched from API3, except EURO. The EURO token price will be fetched from Chainlink(https://gnosisscan.io/address/0xab70BCB260073d036d1660201e9d5405F5829b7a).

The aggregator for EURO/USD price feed is https://gnosisscan.io/address/0x759be90a34E426042ed7d17916B78a5cD2567dd1.

In this aggregator, the min/max answer is set to `1000000/100000000000(https://gnosisscan.io/unitconverter?wei=100000000000)`. It means that once the actual price is lower than min answer or higher than max answer, the price feed will return incorrect price.

Although it's quite impossible to trigger this, considering this is EUR token, one min/max answer check in contract is still suggested.

Impact

Fetch the incorrect price from the Chainlink, users may be liquidated incorrectly.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/e047c2c73d7b208974b561012ac97e6cf90bf7ff/evro/contracts/src/PriceFeeds/WBTCPriceFeed.sol#L52

Tool Used

Manual Review

Recommendation

Add min/max answer check when the EUR price is fetched.

Issue L-1: SDAIPriceFeed missing constructor price feed check [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/22

Summary

The `SDAIPriceFeed` constructor does not call `_fetchPricePrimary()` to initialize `lastGoodPrice`, leaving it at zero and causing potential issues on first oracle failure.

Vulnerability Detail

Unlike other price feeds (GNOPriceFeed, WBTCPriceFeed, etc.), `SDAIPriceFeed` does not call `_fetchPricePrimary()` in its constructor. This means:

1. `lastGoodPrice` starts at 0
2. If the first `fetchPrice()` call encounters an oracle failure, `_shutDownAndSwitchToLastGoodPrice()` returns 0
3. A price of 0 could cause division by zero or allow infinite borrowing

Impact

If oracles fail on the first price fetch after deployment, the system returns a price of 0.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/PriceFeeds/SDAIPriceFeed.sol#L13-L21

```
constructor(address _daiUsdOracleAddress, address _eurUsdOracleAddress, uint256
↪  _daiUsdStalenessThreshold, uint256 _usdEurStalenessThreshold, address
↪  _borrowerOperationsAddress, address _sdaiAddress)
    MainnetPriceFeedBase(_daiUsdOracleAddress, _daiUsdStalenessThreshold,
    ↪  _borrowerOperationsAddress)
{
    eurUsdOracle.aggregator = AggregatorV3Interface(_eurUsdOracleAddress);
    eurUsdOracle.stalenessThreshold = _usdEurStalenessThreshold;
    eurUsdOracle.decimals = eurUsdOracle.aggregator.decimals();

    sdai = IERC4626(_sdaiAddress);

    // @audit - missing: _fetchPricePrimary();
```

```
    // @audit - missing: assert(priceSource == PriceSource.primary);
}
```

Compare to GNOPriceFeed: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/PriceFeeds/GNOPriceFeed.sol#L17-L18

```
_fetchPricePrimary();
assert(priceSource == PriceSource.primary);
```

Tool Used

Manual Review

Recommendation

Add initialization in the constructor:

```
constructor(...) MainnetPriceFeedBase(...) {
    eurUsdOracle.aggregator = AggregatorV3Interface(_eurUsdOracleAddress);
    eurUsdOracle.stalenessThreshold = _usdEurStalenessThreshold;
    eurUsdOracle.decimals = eurUsdOracle.aggregator.decimals();

    sdai = IERC4626(_sdaiAddress);

    _fetchPricePrimary();
    assert(priceSource == PriceSource.primary);
}
```

Issue L-2: IWBTCZapper interface mismatches implementation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/23

Summary

The `IWBTCZapper` interface has incorrect function signatures and missing functions compared to the actual `WBTCZapper` implementation.

Vulnerability Detail

Several discrepancies exist between the interface and implementation:

1. `closeTroveFromCollateralWithWBTC` doesn't exist - it's `closeTroveFromCollateral`
2. `adjustTroveWithWBTC` has wrong arguments in the interface
3. Missing functions: `withdrawEvro`, `repayEvro`, `adjustZombieTroveWithWBTC`, `receiveFlashLoanOnCloseTroveFromCollateral`

Impact

External integrations using the interface will fail to compile or call the wrong functions.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/Zappers/Interfaces/IWBTCZapper.sol#L7-L13

```
interface IWBTCZapper {
    // @audit - missing: withdrawEvro, repayEvro, adjustZombieTroveWithWBTC,
    ↪  receiveFlashLoanOnCloseTroveFromCollateral

    function adjustTroveWithWBTC(...) external;  // @audit - wrong arguments

    function closeTroveFromCollateralWithWBTC(...) external;  // @audit - doesn't
    ↪  exist, should be closeTroveFromCollateral
}
```

Tool Used

Manual Review

Recommendation

Update the interface to match the implementation exactly.

Issue L-3: TroveNFT Duplicates ERC721Enumerable Logic [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/issues/25

Summary

The `TroveNFT` contract maintains a custom `_ownerToTroveIds` mapping that duplicates functionality already provided by the inherited `ERC721Enumerable` extension.

Vulnerability Detail

`ERC721Enumerable` already maintains mappings to track all tokens owned by each address via `_ownedTokens` mapping and provides `tokenOfOwnerByIndex()` function. The custom `_ownerToTroveIds` array duplicates this logic, increasing gas costs and storage usage.

Impact

Unnecessary gas costs for minting/transferring and storage overhead.

Code Snippet

https://github.com/sherlock-audit/2025-12-evro-finance-dec-26th/blob/main/evro/contracts/src/TroveNFT.sol#L24

```
mapping(address => uint256[]) private _ownerToTroveIds;  // @audit - duplicates
↪  ERC721Enumerable

function ownerToTroveIds(address owner) external view returns (uint256[] memory) {
    return _ownerToTroveIds[owner];
}
```

ERC721Enumerable already provides:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/extensions/ERC721Enumerable.sol#L18-L19

```
mapping(address => mapping(uint256 => uint256)) private _ownedTokens;
// Can iterate using tokenOfOwnerByIndex(owner, index) for index in [0,
↪  balanceOf(owner))
```

Tool Used

Manual Review

Recommendation

Remove the custom mapping and use ERC721Enumerable's built-in functionality:

```
function ownerToTroveIds(address owner) external view returns (uint256[] memory) {
    uint256 balance = balanceOf(owner);
    uint256[] memory troveIds = new uint256[](balance);
    for (uint256 i = 0; i < balance; i++) {
        troveIds[i] = tokenOfOwnerByIndex(owner, i);
    }
    return troveIds;
}
```

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.