ZEALYNX SECURITY

Web3 Security & Smart Contract Development

# WEDEFIN

## Security Assessment


May 28th, 2024 - Prepared by Zealynx Security

ZealynxSecurity@protonmail.com

| | |
|---|---|
| Secoalba | @Seecoalba |
| Bloqarl | @TheBlockChainer |

# Contents


1. About Zealynx
2. Disclaimer
3. Overview
   - 3.1 Project Summary
   - 3.2 About Wedefin
   - 3.3 Audit Scope
4. Audit Methodology
5. Severity Clasification
6. Executive Summary
7. Audit Findings
   - 7.1. High Findings
     - (H-1) RefundETH() missing
     - (H-2)Reentrancy risk on WEDXIndexPortfolio's deposit() and withdraw()
     - (H-3) WEDXTreasury's ETH balance can be drained completely by anyone
   - 7.2 Low Findings
     - (L-1) Unnecessary gas consumption: Lack a check for zero input
     - (L-2) Not following CEI pattern
     - (L-3) Potential Mismanagement of Array Lengths in WEDXManager
     - (L-4) Sandwich Attacks During Rebalancing
     - (L-5) Unintended Reset of Portfolio with Empty newAssets Array
     - (L-6) Unnecessary Swapping of WNATIVE to Itself Allowed in swapNative Function
     - (L-7) Unexpected Matching Inputs
   - 7.3 Informational Suggestions
   - 7.4 Gas Optimization Suggestions

# 1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Shieldify, AuditOne, Bastion Wallet, Side.xyz, Possum Labs, and Aurora (NEAR Protocol-based).

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our blog, Bloqarl's blog, and Sergio's blog.

Zealynx has achieved public recognition, including a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

# 2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. I cannot assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

# 3. Overview

## 3.1 Project Summary

A time-boxed independent security assessment of the Wedefin protocol was done by Zealynx Security, with a focus on the security aspects of the application's implementation. We performed the security assessment based on the agreed scope, following our approach and methodology. Based on our scope and our performed activities, our security assessment revealed 3 High, and 8 Low severity security issues. Additionally, different informational and gas optimization suggestions were also made which, if resolved appropriately, may improve the quality of Wedefin's Smart contracts.

## 3.2 About Wedefin

Website: https://www.wedefin.com/
Docs: https://docs.wedefin.com/
Portfolio builder: https://main-builder-v0.wedefin.com/
Index fund: https://main-index-v0.wedefin.com/

Wedefin is a decentralized index fund where the asset allocation is dictated by the community via competition. Meaning professional can build their own portfolio, and a smart contract will rank their performance according to liquidity of their assets, volatility, and profits. Then according to the rank, we will take the top 20 people and build the asset allocation of the index fund with weight average.

## 3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes 1294 nLOC- normalized source lines of code (only source-code lines).
The core contracts on the scope are the following:

- WEDXBasePortfolio
- WEDXProPortfolio
- WEDXIndexPortfolio
- distroMath
- IWEDXInterfaces
- WEDXGroup
- WEDXswap
- WEDXlender
- WEDXlenderSingle
- WEDXManager
- WEDXRanker
- WEDXTreasury
- WEDXConstants
- WEDXDeployerPro
- WEDXDeployerIndex

# 4. Audit Methodology:

### Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough unit and fuzz testing, the usage of Formal Verification tools, and meticulous manual security reviews.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

# 5. Severity Clasification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

# 6. Executive Summary

## Summary of Findings

| Vulnerability | Severity |
|---|---|
| refundETH() missing | High |
| Reentrancy risk on WEDXIndexPortfolio's deposit() and withdraw() | High |
| WEDXTreasury's ETH balance can be drained completely by anyone | High |
| Unnecessary gas consumption: Lack a check for zero input | Low |
| Not following CEI pattern | Low |
| Potential Mismanagement of Array Lengths in WEDXManager | Low |
| Sandwich Attacks During Rebalancing | Low |
| Unintended Reset of Portfolio with Empty newAssets Array | Low |
| Unnecessary Swapping of WNATIVE to Itself Allowed in swapNative Function | Low |
| Unexpected Matching Inputs | Low |
| _wedxGroupAddress needs to be passed in constructor | Low |

| Test Name | FUZZ | FV |
|---|---|---|
| testFuzzNormalizeWithNonZeroSum | ✅ | |
| testFuzzNormalizeWithVariousNonZeroValues | ✅ | |
| testFuzzNormalizeWithZeroArray | ✅ | |
| testFuzzNormalizeWithLargeValues | ✅ | |
| testFuzzNormalizeWithSmallValues | ✅ | |
| testFuzzIsInNewAssets | ✅ | |
| testNormalizeProportionality | ✅ | |
| testNormalizeNoModificationWithZeroSum | ✅ | |
| testNormalizeDistribution | ✅ | |
| testFuzzCollectToken | ✅ | |
| testFuzzGetTokenBalance | ✅ | |
| testFuzzIsLoanPossible | ✅ | |
| testIsLoanPossibleForKnownTokens | ✅ | |
| testFuzzUpdateTraderData | ✅ | |
| testFuzzComputeRanking | ✅ | |
| testFuzzDepositWithSimpleFee | ✅ | |
| testFuzzWithdraw | ✅ | |
| testRankingWithEqualLengthSeries | ✅ | |
| testRankingNoDivisionByZero | ✅ | |
| testRankingSigmaCalculation | ✅ | |
| testRankingCalculation | ✅ | |
| testFuzzDepositGeneralFee | ✅ | |
| testFuzzDepositRewardFee | ✅ | |
| testFuzzRedeem | ✅ | |
| testFuzzMultipleDepositsAndRedeems | ✅ | |
| testFuzzRankRewardDistribution | ✅ | |
| testFuzzMinimumDeposit | ✅ | |
| testFuzzMaximumDeposit | ✅ | |
| check_FuzzNormalizeWithNonZeroSum | | ✅ |
| check_FuzzNormalizeWithVariousNonZeroValues | | ✅ |
| check_FuzzNormalizeWithZeroArray | | ✅ |
| check_FuzzNormalizeWithMixedValues | | ✅ |
| check_FuzzNormalizeWithSmallValues | | ✅ |
| check_FuzzIsInNewAssets | | ✅ |
| check_FuzzIsNotInNewAssets | | ✅ |
| check_FuzzIsFirstAssetInNewAssets | | ✅ |
| check_FuzzIsLastAssetInNewAssets | | ✅ |
| check_FuzzNormalizeWithVariableLength | | ✅ |
| check_NormalizeProportionality | | ✅ |
| check_NormalizeNoModificationWithZeroSum | | ✅ |
| check_NormalizeDistribution | | ✅ |
| check_RankingWithEqualLengthSeries | | ✅ |
| check_RankingNoDivisionByZero | | ✅ |
| check_RankingSigmaCalculation | | ✅ |
| check_RankingCalculation | | ✅ |
| check_testBasicDepositReflectsInCWETH | | ✅ |
| check_FuzzDepositGeneralFee | | ✅ |
| check_FuzzDepositRewardFee | | ✅ |

# 7. Findings

## 7.1. High Findings

### [H-1] refundETH() missing

#### Summary

Potential loss of user funds due to unhandled refunds of unspent ETH in the Uniswap V3 **ISwapRouter** after executing the swapNative() function.

#### Vulnerability Description

**Location**: **WEDXswap::swapNative()**

SwapRouter allows swapping of ETH for ERC20 tokens. The difference between selling ETH and an ERC20 token is that the contract can compute and request from the user the exact amount of ERC20 tokens to sell, but, when selling ETH, the user has to send the entire amount when making the call (i.e. before the actual amount was computed in the contract).

As swaps made via SwapRouter can be partial, there are scenarios when ETH can be spent partially. However, the contract doesn't refund unspent ETH in such scenarios:

1. When sqrtPriceLimitX96 is set (SwapRouter.sol#L80, SwapRouter.sol#L164), the swap will be interrupted when the limit price is reached, and some ETH can be left unspent.
2. A swap can be interrupted earlier when there's not enough liquidity in a pool.
3. Positive slippage can result in more efficient swaps, causing exact output swaps to leave some ETH unspent (even when it was pre-computed precisely by the caller).

#### Impact

- Consequence: The SwapRouter can retain leftover ETH after a swap. This ETH can be withdrawn by anyone using the SwapRouter.refundETH() function, leading to potential loss for the SwapRouter user.

#### Mitigation

- Solution: Ensure the **refundETH()** function of Uniswap V3's router is called to return any unspent ETH. This prevents funds from being inadvertently left in the swap router, safeguarding user assets.

# (H-2) Reentrancy risk on WEDXIndexPortfolio's deposit() and withdraw()

## Summary

The **supplyLendToken** and **withdrawLendToken** functions in the **WEDXIndexPortfolio** contract don't implement the CEI Pattern and hence are vulnerable to reentrancy attacks when interacting with ERC777 tokens. These tokens can execute custom logic during transfer operations, which can potentially allow a malicious contract to re-enter and manipulate the state of the **WEDXIndexPortfolio** contract.

## Vulnerability Description

Both the **supplyLendToken** and **withdrawLendToken** functions interact with external contracts for token transfers and lending operations. Specifically, these functions call into Aave's lending pool to supply or withdraw assets. If an ERC777 token is used, its hooks (**tokensToSend** and **tokensReceived**) can be exploited to trigger reentrant calls back into these functions, potentially leading to unexpected behaviors such as multiple withdrawals or supplies in a single transaction, manipulating balances or state in a manner advantageous to the attacker.

## Impact

If exploited, an attacker could potentially manipulate account balances, duplicate assets, or cause financial loss to users by triggering unintended actions within the contract. This could undermine the integrity of the lending protocol and lead to loss of user funds.

## Migitation

To mitigate this vulnerability, consider the following changes:

1. **Use Reentrancy Guards**: Implement reentrancy guards in the **supplyLendToken**` and **withdrawLendToken**` functions to prevent reentrant calls.

2. **External Call Placement** : Review and adjust the order of external calls and state updates in accordance with the checks-effects-interactions pattern to minimize the impact of any reentrant calls.

---

## (H-3) WEDXTreasury's ETH balance can be drained completely by anyone

### Summary

The redeem function in the WEDXTreasury contract is vulnerable to reentrancy attacks, allowing attackers to drain the contract's ether by repeatedly calling the redeem function before their token balance is updated.

### Vulnerability Description

The redeem function sends ether to the caller before burning their tokens. This allows a malicious contract to re-enter the redeem function through a fallback or receive function, bypassing the balance check and draining the contract's funds.

### Proof of Concept (PoC)

```
contract Attack {
    WEDXTreasury public treasury;
    uint256 public amountToRedeem;

    constructor(WEDXTreasury _treasury) {
        treasury = _treasury;
    }

    receive() external payable {
        if (address(treasury).balance >= amountToRedeem) {
            treasury.redeem(amountToRedeem);
        }
    }

    function attack(uint256 _amount) external {
        amountToRedeem = _amount;
        treasury.redeem(amountToRedeem);
    }
}
```

By running the attack function, anyone would be able to steal all the WEDXTreasury funds because when sending ether to the contract it would enter the receive() function and call again redeem() getting yet another transfer with the same amount of ETH sent and this would continue in a loop until the treasury's contract balance is emptied.

## Mitigation

**1.** Apply the Checks-Effects-Interactions Pattern:

```
function redeem(uint256 amount) public returns (uint256) {
    require(balanceOf(msg.sender) >= amount, "Insufficient balance");
    _burn(msg.sender, amount); // Burn tokens before sending ether

    address payable sender = payable(msg.sender);
    (bool success, ) = sender.call{value: amount}("");
    require(success, "Withdrawal failed");

    return amount;
}
```

**2**. Use **nonReentrant** modifier from OpenZeppelin

**Link Test** 

## 7.2. Low Findings

### (L-1) Unnecessary gas consumption: Lack a check for zero input

#### Summary

The **redeem** , **lendToken** , **collectToken** and more... functions lack a check for zero input, which allows them to be executed with an amount of zero, leading to unnecessary gas consumption without any meaningful transaction effect.

#### Vulnerability Description

While the functions have other checks, they do not verify that the **amount** parameter is greater than zero before proceeding.

#### Impact

If called with an amount of zero, the function unnecessarily consumes gas for validation checks and state modifications. This behavior does not lead to direct financial loss but can result in wasted gas if abused or called erroneously with zero. It also poses potential disruption or inconvenience through minor network load increase due to such calls.

#### Mitigation

To prevent unnecessary gas usage and ensure only meaningful transactions are processed, the function should include a check to verify that the **amount** is greater than zero:

```
require(amount != 0, "Amount must be greater than zero");
```

# (L-2) Not following CEI pattern

## Summary

The **WEDXBasePortfolio** smart contract exhibits multiple instances where the Check-Effects-Interactions (CEI) pattern is not properly implemented. This pattern is crucial for preventing reentrancy attacks and ensuring state consistency.

Although the user should only have access to his own portfolio not being able to withdraw any additional funds than the ones he deposited, it is recommended to correct the implementation.

## Vulnerability Description

1. **Withdraw Function (withdraw)**: This function interacts with external contracts for token swaps before updating the internal **totalAssets** state. This sequence could allow reentrancy or manipulation of state through callbacks from called contracts.
2. **Brute Force Withdraw Function (withdrawBruteForced):** Similar to the **withdraw** function, assets are transferred before the **totalAssets** state is updated, which exposes the contract to reentrancy attacks.
3. **Portfolio Reset Function (_setPortfolio)**: The function performs external interactions with potentially untrusted contracts to swap tokens before updating the **totalAssets**. This could be exploited if the external contracts called are malicious or contain reentrant functions.

## Impact

If exploited, these vulnerabilities could allow an attacker, potentially even the owner, to manipulate contract balances, extract funds unduly, or influence the contract operations to their benefit.

The risk is low as mentioned earlier, yet, we recommend to fix it.

## Mitigation

- Adhere Strictly to the CEI Pattern: Ensure that all state changes occur before external calls or token transfers. This rearrangement of operations will prevent reentrancy and preserve state integrity.
- Implement Reentrancy Guards: Utilize OpenZeppelin's **ReentrancyGuard** to block reentrant calls across all state-changing public and external functions.

---

# (L-3) Potential Mismanagement of Array Lengths in WEDXManager

## Summary

The WEDXManager smart contract, responsible for tracking and updating trader data and computing rankings, has a potential vulnerability related to the management of trader data arrays.

The issue arises in the handling of the array lengths during updates to trader data, which, if not strictly managed, could lead to data inconsistencies or unexpected behavior, especially during interactions with external contracts like Uniswap.

## Vulnerability Description

The **updateTraderData** function in WEDXManager is designed to update trader data whenever a trader rebalances their portfolio, or deposits or withdraws assets.

This function manipulates several arrays (**performances**, **timestamps**, **minLiquidity**) that track trader activities. When the array length exceeds **_nPoints**, the oldest entries are supposed to be shifted out to maintain a fixed length. However, there's a brief moment where the array length exceeds the intended fixed length (**_nPoints**) before the older entries are removed.

This happens because new data is pushed to the arrays before the excess data is popped out. Although this is resolved within the same transaction and the arrays are corrected before the function execution completes, the brief inconsistency poses a theoretical risk, especially when combined with external calls.

## Impact

The primary risk is related to potential misuse or unexpected behavior when interacting with external systems or during data retrieval and manipulation. If external functions depend on the assumption that these arrays never exceed their maximum length (**_nPoints**), even temporary violations of this assumption could lead to errors or vulnerabilities, especially in high-stress scenarios or attacks aimed at contract's state integrity.

## Proof of Concept

The issue can be highlighted through a series of operations that show the array exceeding its intended maximum size within the transaction execution:

---

```
traderData[traderId].performances.push(newPerformance);
traderData[traderId].timestamps.push(newTimestamp);
traderData[traderId].minLiquidity.push(newLiquidity);
// Arrays now exceed the intended size
require(traderData[traderId].performances.length == _nPoints + 1, "Array length exceeded temporarily");
// Correcting the array size
for (uint i = 0; i < _nPoints; i++) {
    // shifting logic
}
traderData[traderId].performances.pop();
traderData[traderId].timestamps.pop();
traderData[traderId].minLiquidity.pop();
```

## Mitigation

To prevent any issues associated with this temporary array size increase, it is recommended to adjust the array management strategy by ensuring the arrays are maintained at the maximum size without exceeding it at any point during the function execution. This can be achieved by popping the excess elements before pushing new data:

```
if (traderData[traderId].performances.length >= _nPoints) {
    traderData[traderId].performances.pop();
    traderData[traderId].timestamps.pop();
    traderData[traderId].minLiquidity.pop();
}
// Now safely add new data
traderData[traderId].performances.push(newPerformance);
traderData[traderId].timestamps.push(newTimestamp);
traderData[traderId].minLiquidity.push(newLiquidity);
```

This change ensures the length of the arrays is strictly controlled, eliminating any brief inconsistencies and strengthening the contract's robustness, especially in interactions involving external contract calls.

# (L-4) Sandwich Attacks During Rebalancing

## Summary

The **WEDXBasePortfolio** contract is vulnerable to sandwich attacks during rebalancing transactions. This exploit allows a malicious actor to profit by manipulating the market price of tokens involved in the rebalancing process. Despite the use of the **maxSlippage** parameter, the visibility of transactions on the blockchain enables attackers to front-run and back-run rebalancing transactions, leading to potential financial losses for the portfolio.

## Vulnerability Description

Rebalancing transactions in the **WEDXBasePortfolio** contract involve significant token swaps using Uniswap V3 pools. Due to the public nature of Ethereum transactions, these rebalancing transactions are visible in the mempool before being mined.

Malicious actors can exploit this by placing their transactions immediately before and after the rebalancing transaction. This is known as a sandwich attack, where the attacker can manipulate the token prices within the allowed slippage range to profit at the expense of the portfolio.

## Impact

The impact of this vulnerability includes:

- Financial losses for the portfolio due to manipulated token prices.
- Increased transaction costs for the portfolio as a result of slippage.
- Potential destabilization of the portfolio's value and performance.

## Proof of Concept

The issue can be highlighted through a series of operations that show the array exceeding its intended maximum size within the transaction execution:

1. **Setup**
    - **maxSlippage** is set at 5%, meaning the contract is willing to tolerate up to 5% price movement during a swap.

2. **Exploit Scenario:**
    - A malicious actor monitors the mempool for rebalancing transactions.
    - The attacker places a large buy order immediately before the rebalancing transaction, inflating the token price.
    - The rebalancing transaction occurs, accepting the inflated price within the slippage tolerance.
    - The attacker then places a sell order immediately after the rebalancing transaction, profiting from the price difference.

3. **Example Scenario:**
    - Portfolio needs to rebalance and swap 100 ETH for USDC.
    - The attacker front-runs the transaction, buying ETH and inflating the price by 5%.
    - The portfolio's swap transaction is executed at the inflated price, accepting the 5% slippage.
    - The attacker back-runs the transaction, selling ETH at the higher price, and profiting from the price difference.

## Mitigation

**Implementing a Cooldown Period:**

- Introduce a cooldown period between rebalancing transactions to reduce the likelihood of sandwich attacks.
- This involves adding a state variable to track the last transaction time and a modifier to enforce the cooldown.

```
// State variables
uint256 public cooldownPeriod = 300; // Example: 5 minutes
uint256 public lastRebalanceTimestamp;

// Modifier
modifier cooldown() {
    require(block.timestamp >= lastRebalanceTimestamp + cooldownPeriod, "Cooldown period not yet
passed");
    _;
    lastRebalanceTimestamp = block.timestamp;
}

// Apply modifier to rebalancing functions
function deposit() public virtual payable onlyOwner cooldown returns(uint256) {
    // function logic
}

function withdraw(uint256 amount) virtual public onlyOwner cooldown {
    // function logic
}

function withdrawBruteForced() virtual public onlyOwner cooldown {
    // function logic
}

function _setPortfolio(address[] memory newAssets, uint256[] memory newDistribution, uint256 fee)
internal virtual cooldown returns (uint256) {
    // function logic
}

function _changeDistribution(uint256[] memory newDistribution, uint256 fee) internal cooldown returns
(uint256) {
    // function logic
}
```

**Using Private Relays:**

- Relay sensitive transactions through private relays like Flashbots to prevent them from being visible in the public mempool.
- This can be integrated into the deployment and operational procedures of the smart contract.

# (L-5) Unintended Reset of Portfolio with Empty newAssets Array

## Summary

The **_setPortfolio** function in **WEDXBasePortfolio** might be called with an empty newAssets array, leading to the portfolio being reset to holding only the native token (WNATIVE). While this does not cause a significant imbalance, it might not be the intended behavior.

## Vulnerability Description

The internal **_setPortfolio** function can be called with an empty newAssets array from the public setPortfolio function in derived contracts (**WEDXIndexPortfolio** and **WEDXProPortfolio**). This causes:

• Skipping validation of asset pools.
• Resetting the tokenAddresses array to only [WNATIVE].
• Executing **_changeDistribution** with newDistribution.

This effectively resets the portfolio, which might not be intended behavior but does not cause a significant imbalance if newDistribution is correctly set.

## Mitigation

Require Non-Empty newAssets Array:

```
function _setPortfolio(address[] memory newAssets, uint256[] memory newDistribution, uint256 fee)
internal virtual returns (uint256) {
    require(newAssets.length > 0, "Assets array cannot be empty");
    ...
}
```

This ensures that the newAssets array is not empty, preventing accidental resets and maintaining the intended state of the portfolio.

# (L-6) Unnecessary Swapping of WNATIVE to Itself Allowed in swapNative Function

## Summary

The **swapNative** function allows the swapping of the native token (WNATIVE) with itself. This is generally unnecessary and could lead to confusion or unintended behavior.**Impact**

## Vulnerability Description

Allowing the swap of WNATIVE to itself does not provide any practical utility and may lead to unnecessary transaction fees. Additionally, it introduces potential vectors where users might exploit this behavior for arbitrage or other unintended activities.

## Proof of Concept (PoC)

```
function testSwapNative_WNATIVE() public {
    uint256 amountIn = 0.5 * 1e18;
    uint256 maxSlippage = 50; // 0.5% max slippage

    // Send ETH to the contract
    (bool success,) = address(swapContract).call{value: 1 ether}("");
    require(success, "Failed to send ETH to contract");

    // Perform the swap
    vm.prank(owner);
    uint256 amountOut = swapContract.swapNative(WNATIVE, maxSlippage);
```

The test shows that swapping WNATIVE to WNATIVE does not result in any meaningful output, highlighting the redundancy and potential for confusion.

## Recommendations

Add a require statement in the **swapNative** function to prevent the swapping of WNATIVE to itself. This will ensure that such unnecessary and potentially harmful operations are not performed.

By adding this require statement, the contract will reject any attempts to swap WNATIVE to WNATIVE, ensuring that only meaningful swaps are processed.

# (L-7) Unexpected Matching Inputs

## Summary

The validatePool function in the WEDXswap contract does not verify that the input tokens (tokenIn and tokenOut) are different. This can lead to unintended behavior.

## VIulnerability Description

In the WEDXswap contract, the validatePool function allows the same token to be used for both tokenIn and tokenOut. This lack of validation can result in the function processing these identical inputs incorrectly, leading to unexpected and potentially erroneous behavior.

## Impact

This vulnerability can lead to unnecessary transactions and potential confusion. While it does not pose a direct security risk, it may result in wasted resources and inefficiencies within the contract's operation.

## Proof of Concept (PoC)

The validatePool function can be called with the same token for both tokenIn and tokenOut. Without proper validation, the function processes the request incorrectly, which can disrupt the intended logic and flow of the contract.

```
function testValidatePoolWithSameTokens() public {
    // Expect revert with message "Tokens must be different"
    vm.expectRevert("Tokens must be different");
    swapContract.validatePool(address(tokenA), address(tokenA));
}
```

## Mitigation

Implement a validation check in the validatePool function to ensure that tokenIn and tokenOut are not the same.

```
function validatePool(address tokenIn, address tokenOut) public view returns (exInfo memory) {
    require(tokenIn != tokenOut, "Tokens must be different");
    // Additional function logic...
}
```

---

## 7.3 Informational Suggestions

### • [I-1] Redundant Code in validatePool

#### Description

In WEDXswap::validatePool the code below is redundant:

```
if ( tokenIn != WNATIVE ) {
        result = exInfo(j, uniAllowedFees[i], refLiquidity);
} else {
        result = exInfo(j, uniAllowedFees[i], refLiquidity);
}
```

#### Recommendation

Consider removing and leave result = exInfo(j, uniAllowedFees[i], refLiquidity);

### • [I-2] Use external instead of public when not used internally

#### Description

Functions that are not used internally should be marked as external instead of public.

#### Recommendation

Change the visibility of these functions from public to external to optimize gas usage and improve contract efficiency.

### • [I-3] Consider adding address(0) check for function parameters

#### Description

WEDXLender::lendToken
WEDXLender::collectToken

#### Recommendation

Consider adding address(0) check

- ## [I-4] Redundant conditions in WEDXIndexPortfolio

### Description

Deposit and setPortfolio functions have a redundant check in order to call supplyLendToken

```
if
(IWEDXlender(IWEDXGroup(_wedxGroupAddress).getLenderContractAddress()).isLoanPossible(tokenAddresses[i]
) == true && totalAssets[tokenAddresses[i]] > 0) {
    supplyLendToken(tokenAddresses[i]);
}
```

is unnecessary because the supplyLendToken function already includes these checks:

### Recommendation

Remove the redundant condition from the deposit and setPortfolio functions, as supplyLendToken already handles these validations internally. This will streamline the code and avoid unnecessary checks.

```
require(totalAssets[tokenAddress] > 0, "User does not have this token");
require(IWEDXlender(IWEDXGroup(_wedxGroupAddress).getLenderContractAddress()).isLoanPossible(tokenAddre
ss), "Token is not available for lending");
```

- ## [I-5] Redundant function

### Description

The getAssetsExtended function is redundant because it simply calls the getAssets function without adding any additional functionality. Both functions return the same result.

```
function getAssets() public view returns (uint256[] memory) {
    uint256[] memory totalAssetsResult = new uint256[](tokenAddresses.length);
    for(uint256 i = 0; i < tokenAddresses.length; i++) {
        totalAssetsResult[i] = totalAssets[tokenAddresses[i]];
    }
    return totalAssetsResult;
}

function getAssetsExtended() virtual public view returns (uint256[] memory) {
    return getAssets();
}
```

### Recommendation

Remove the getAssetsExtended function as it does not provide any additional value beyond what getAssets already offers

- ## [I-6] WNATIVE should be passed in constructor instead of as constant

### Description

The WNATIVE token address is currently set as a constant in the contract, which limits flexibility.

### Recommendation

Pass the WNATIVE token address as a parameter in the constructor to allow for greater flexibility and configurability during deployment.

## 7.4 Gas Optimization suggestions

- ### [G-1] Use Custom Errors

#### Description
Using custom errors can make error messages more gas efficient and descriptive.

#### Recommendation
Define and use custom errors instead of revert strings to save gas and improve clarity.

- ### [G-1] Consider using x ! = 0 instead of x > 0

#### Description
Using x != 0 is a more efficient way to check for non-zero values than x > 0.

#### Recommendation
Replace instances of x > 0 with x != 0 for gas optimization.

- ### [G-1] Consider using if + Error handlers instead of require + strings

#### Description
Using if statements with custom error handlers can be more gas efficient than using require with strings.

#### Recommendation
Replace require statements with if conditions followed by custom error handlers.

- ## [G-4] Consider using ++i instead of i++

### Description
Using the pre-increment operator ++i is slightly more gas efficient than the post-increment operator i++.

### Recommendation
Replace instances of i++ with ++i in loops and other operations.

- ## [G-5] Consider using calldata instead of memory

### Description
Using calldata for function parameters is more gas efficient than using memory.

### Recommendation
Replace memory with calldata for function parameters where appropriate.

- ## [G-6] Consider saving the array length in memory before using it in loop

### Description
Fetching the array length repeatedly in a loop is less efficient than storing it in a variable.

### Recommendation
Store the array length in a local variable before the loop to optimize gas usage.