



Security Audit Report

27/12/2024

Flamingo OrderBook v2 broker-contract

Content

Introduction	4
Disclaimer	4
Scope	5
Executive Summary.....	6
Conclusions	7
Vulnerabilities	8
List of vulnerabilities	8
Vulnerability details	9
Lack of Slippage Protection in Swaps.....	10
Incorrect Pair Creation Logic	12
Misassigned Token Decimals	14
Bypass Contract Caller	16
Vulnerable NoReentrantAttribute	17
Wrong Symbol in FlamingoSwapPairContract.....	18
Wrong Reentrancy Protection	19
Denial of Service in the Query Methods	20
Limit Call Rights	21
Wrong Timestamp Calculation	22
Unnecessary Trust Required.....	23
Lack of Inputs Validation	24
Wrong GetPaymentStatus Logic.....	26
Wrong NEP-17 Standard	28
FToken Requires Minter to Work	29
Reentrancy Pattern	30
More Restrictive Conditions.....	32
Missing Data Argument on Update Method	33
Duplicated Pools	34
Emit Events on State Changes.....	35
Outdated Framework	37
GAS Optimization	38
Bad Coding Practices.....	44
Safe Storage Access.....	46
Lack of Safe Method Attribute	47
Contracts Management Risks	49
Wrong Return in Methods	51
Unsecured Ownership Transfer	52

Hard-coded Values.....	53
Missing Manifest Information.....	54
Use of Assert instead of Throw	55
Code Typos.....	56
Decimal Truncation in Division.....	57
Code Improvements.....	58
Annexes.....	60
Methodology	60
Manual Analysis.....	60
Automatic Analysis.....	60
Vulnerabilities Severity.....	61

Introduction

Flamingo is a platform that helps convert tokens, be a liquidity provider and earn yield. By providing liquidity, also known as staking, you earn yield by collecting fees and getting minted FLM as a reward. Flamingo Finance aims to make it easy to buy/sell crypto, invest and earn revenue directly on the blockchain.



OrderBook v2 is designed to be used in conjunction with the AMM. The takers are forced to buy and sell both in the AMM and in the OrderBook. Forcing the trading to happen in both places has the objective to ensure that arbitragers always have to trigger limit orders when trading.

As solicited by **Neo** and as part of the vulnerability review and management process, Red4Sec has been requested to perform a security code audit in order to evaluate the security of the **Flamingo OrderBook** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of **Flamingo OrderBook broker-contract**. The performed analysis shows that the smart contract does contain critical vulnerabilities.

Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered either "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project, nor as guarantee on the operation or viability of the implemented financial product.

Red4Sec makes full effort and applies every resource available for each audit, however it does not warrant the function, nor the safety of the project and it cannot be deemed a sufficient assessment of the code's utility and safety, bug-free status, or any other declarations of the project. Additionally, Red4Sec makes no security assessments or judgments about the underlying business strategy, or the individuals involved in the project.

Blockchain technology and cryptographic assets come with their own new risks and challenges, where the ecosystem, platform, its programming language, and other software related to said technology can have vulnerabilities that could lead to exploits. As a result, the audit cannot guarantee the explicit security of the audited projects.

The audit reports can be used to improve the code quality of smart contracts, to help limit the vectors of attack and to lower the high level of risks associated with utilizing new and continually changing technologies such as cryptographic tokens and blockchain, but they are unable to detect any future security concerns with the related technologies.

Scope

Red4Sec Cybersecurity has made a thorough audit of the **Flamingo OrderBook** security level against attacks, identifying possible errors in the design, configuration, or programming; therefore, guaranteeing the availability, integrity and confidentiality of the project and the possible assets treated and stored.

The scope of this evaluation includes the following items provided by **Neo**:

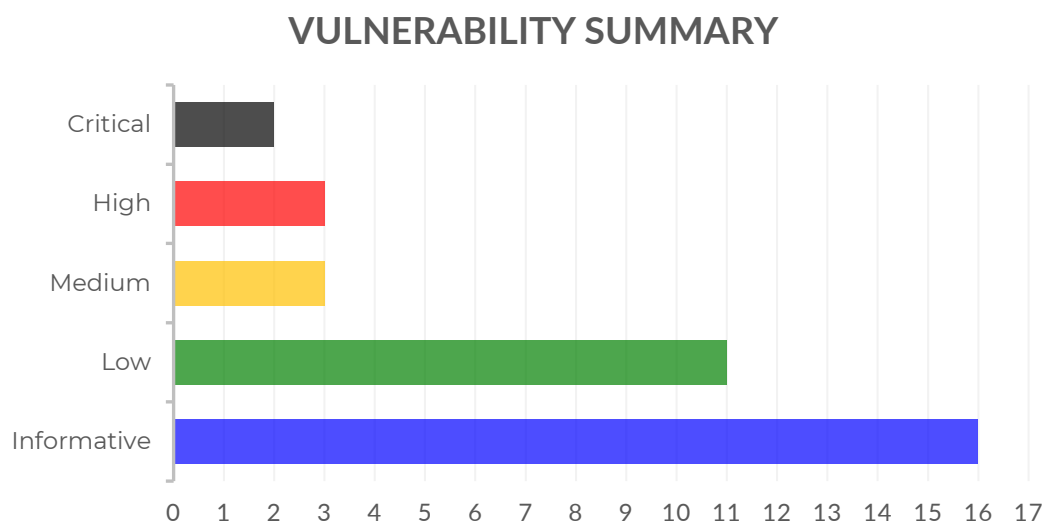
- <https://github.com/flamingo-finance/flamingo-sc-monorepo/tree/feature/broker-contract>
 - Commit August: 614eaa52bf5e0a84a21455c1f432c0c914482070
 - Commit December: 605bd9a1660a7c916b3f72fe7a5dc69942037c28

Executive Summary

The security audit against **Flamingo OrderBook** has been conducted between the following dates: **07/08/2024** and **27/12/2024**.

Once the analysis of the technical aspects has been completed, the performed analysis shows that the audited source code contains critical vulnerabilities that should be mitigated as soon as possible.

During the analysis, a total of **35 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.



Conclusions

Based on the audit performed, the **OrderBook** project appears to possess numerous vulnerabilities across various risk levels, from critical to informative. The overall security status of the project is concerning and requires immediate attention and rectification to ensure the integrity and security of the system.

The most critical vulnerabilities include the Lack of Slippage Protection in Swaps and Incorrect Pair Creation Logic. These can lead to significant losses and manipulation of the contract, which could undermine the entire project. Additionally, risks such as Misassigned Token Decimals, the potential to Bypass Contract Caller, and a Vulnerable NoReentrantAttribute present high risks, opening the door for exploitation by malicious actors. It should be noted that both critical vulnerabilities have been properly mitigated by the development team, however, there are still two remaining high-risk vulnerabilities that need to be addressed as soon as possible.

Medium risks such as Wrong Fee Calculation, Wrong Symbol in FlamingoSwapPairContract, and Wrong Reentrancy Protection can cause functional issues and potential financial discrepancies. The low-risk vulnerabilities, including Denial of Service in the Query Methods, Limit Call Rights, and Wrong Timestamp Calculation, although less severe, can still cause operational inefficiencies and minor security issues.

The informative risks, while not directly impacting the security of the contract, indicate poor coding practices, outdated frameworks, and potential gas optimization issues, which can lead to inefficiencies and future vulnerabilities if left unaddressed.

In light of the audit findings, it is recommended to address the critical and high-risk vulnerabilities immediately. This includes implementing slippage protection, rectifying the pair creation logic, ensuring the correct assignment of token decimals, securing the contract caller, and strengthening the NoReentrantAttribute.

For medium and low-risk vulnerabilities, a systematic approach to their rectification should be adopted, prioritizing those that can have the most significant impact on the project.

As for the informative risks, a thorough review of the codebase should be undertaken to improve coding practices, update frameworks, and optimize gas usage. These improvements can help prevent future vulnerabilities and improve the overall efficiency and performance of the smart contract.

Finally, it should be noted that the issue *FOB2-06 Wrong Fee Calculation* has been considered discarded and removed from the report. As well as the issue *FOB2-20 Duplicated Pools* has been marked as closed since it was verified that the logic was intended.

Overall, even though certain issues have been addressed the project still has various vulnerabilities that must be fixed as soon as possible in order to make the project safe.

Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered upon the security audit.

List of vulnerabilities

Below, we have gathered a complete list of the vulnerabilities detected by Red4Sec, presented, and summarized in a way that can be used for risk management and mitigation.

Table of vulnerabilities			
ID	Vulnerability	Risk	State
FOB2-01	Lack of Slippage Protection in Swaps	Critical	Fixed
FOB2-02	Incorrect Pair Creation Logic	Critical	Fixed
FOB2-03	Misassigned Token Decimals	High	Fixed
FOB2-04	Bypass Contract Caller	High	Open
FOB2-05	Vulnerable NoReentrantAttribute	High	Open
FOB2-06	Wrong Fee Calculation	Medium	Discarded
FOB2-07	Wrong Symbol in FlamingoSwapPairContract	Medium	Open
FOB2-08	Wrong Reentrancy Protection	Medium	Fixed
FOB2-09	Denial of Service in the Query Methods	Low	Open
FOB2-10	Limit Call Rights	Low	Open
FOB2-11	Wrong Timestamp Calculation	Low	Open
FOB2-12	Unnecessary Trust Required	Low	Open
FOB2-13	Lack of Inputs Validation	Low	Open
FOB2-14	Wrong GetPaymentStatus Logic	Low	Open
FOB2-15	Wrong NEP-17 Standard	Low	Open
FOB2-16	FToken Requires Minter to Work	Low	Open
FOB2-17	Reentrancy Pattern	Low	Open
FOB2-18	More Restrictive Conditions	Low	Fixed
FOB2-19	Missing data Argument on Update Method	Informative	Assumed
FOB2-20	Duplicated Pools	Informative	Closed
FOB2-21	Emit Events on State Changes	Informative	Partially Fixed
FOB2-22	Outdated framework	Informative	Open
FOB2-23	GAS Optimization <small>Updated</small>	Informative	Open
FOB2-24	Bad Coding Practices	Informative	Open

FOB2-25	Safe Storage Access	Informative	Open
FOB2-26	Lack of Safe Method Attribute	Informative	Partially Fixed
FOB2-27	Contracts Management Risks	Informative	Open
FOB2-28	Wrong Return in Methods	Informative	Open
FOB2-29	Unsecured Ownership Transfer	Informative	Open
FOB2-30	Hard-coded Values	Informative	Open
FOB2-31	Missing Manifest Information	Informative	Open
FOB2-32	Use of Assert instead of Throw	Informative	Open
FOB2-33	Code Typos	Informative	Open
FOB2-34	Decimal Truncation in Division <small>New</small>	Low	Open
FOB2-35	Code Improvements <small>New</small>	Informative	Open

Vulnerability details

In this section, we provide the details of each of the detected vulnerabilities indicating the following aspects:

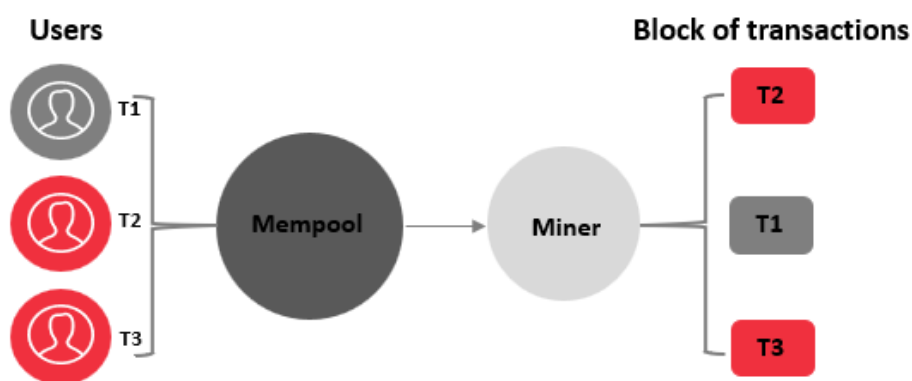
- Category
- Active
- Risk
- Description
- Recommendations

Lack of Slippage Protection in Swaps

Identifier	Category	Risk	State
FOB2-01	Timing and State	Critical	Fixed

An issue has been detected in `FlamingoBroker` where protections against sandwich attacks have not been implemented.

A Sandwich Attack is conducted with the objective of front-running a cryptocurrency in DeFi projects. A user makes a transaction to trade a cryptocurrency, X asset for a Y asset with the intent of making a large purchase. Immediately a front-running bot detects it and buys the asset before the large trade is executed, front-running the victim. This action sandwiches the transaction of the victim and raises the price of the Y asset for the user, consequently increasing the slippage.



The `ProcessSellOrder` and `ProcessBuyOrder` methods of the `FlamingoBroker` contract handle the buy and sell operations of tokens. These methods internally call the helper methods `AMMHelper.SafeSwapTokenInForTokenOut` and `AMMHelper.SafeSwapTokenOutForTokenIn`, which interact with the Automated Market Maker (AMM) for the exchange of tokens. However, there is a critical issue with how slippage protection is handled in these methods.

In `SafeSwapTokenInForTokenOut`, the `amountOutMin` parameter is set to 0.

```
Contract.Call(GetAMMRouter(), "swapTokenInForTokenOut", CallFlags.All, new object[] {
    amountIn, 0, paths, Runtime.Time + 1 });
```

This means that the swap will be executed regardless of how many output tokens are received, disabling slippage protection.

On the other hand, in `SafeSwapTokenOutForTokenIn`, the `amountInMax` parameter is set to a very large number (9999999999999999) producing the same result as previously mentioned.

```
Contract.Call(GetAMMRouter(), "swapTokenOutForTokenIn", CallFlags.All, new object[] {
    amountOut, 9999999999999999, paths, Runtime.Time + 1 });
```

Based on the current implementation, several issues arise that can lead to the following risks:

- **Front-running attacks:** Malicious users could exploit this lack of protection by observing pending transactions and manipulating the price in their favor before the transaction is mined, causing users to receive a significantly worse exchange rate.
- **Excessive slippage:** In volatile markets or pools with low liquidity, users could end up receiving much less than expected due to price fluctuations between the time the transaction is sent and when it is executed.
- **Losses to users:** Without adequate slippage limits, users are exposed to greater risks of financial loss, which undermines trust in the platform.

Recommendations

- It is advisable to implement adequate protection against slippage by allowing users to specify acceptable limits, or by calculating reasonable values within the contract.

Source Code References

- [src/Flamingo.FLUND/FLUND.cs#L546](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L199](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L213](#)
- [src/Flamingo.Broker/FlamingoBroker.Sell.cs#L203](#)
- [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L253](#)

Fixes Review

This issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/pull/65>

Incorrect Pair Creation Logic

Identifier	Category	Risk	State
FOB2-02	Business Logic	Critical	Fixed

The OrderBook contract has been identified as having flawed logic during pair creation, which prevents more than one pair from being properly configured within the contract.

In fact, if more than one pair is created, the values of the first pair (pairId 1) will be replaced by the following ones, affecting all existing orders, potentially triggering significant losses of tokens for contract users.

The main problem is that when the pairId is stored, it is relative to the pair of tokens that are intended to be stored, instead of using a general counter, so new pairs receive an initial counter of 1.

```
public static class PairIdStorage
{
    private static readonly byte[] pairIdPrefix = new byte[] { 0x10 };

    internal static BigInteger Get(UInt160 tokenA, UInt160 tokenB, BigInteger version)
    {
        StorageMap pairToPairIdMap = new StorageMap(Storage.CurrentReadOnlyContext, pairIdPrefix);
        var value = pairToPairIdMap[tokenA + tokenB + version];
        return value != null ? (BigInteger)value : 0;
    }

    internal static BigInteger Current(UInt160 tokenA, UInt160 tokenB, BigInteger version)
    {
        return Get(tokenA, tokenB, version);
    }

    internal static BigInteger Add(UInt160 tokenA, UInt160 tokenB, BigInteger version)
    {
        var pairId = Get(tokenA, tokenB, version) + 1;
        StorageMap pairToPairIdMap = new StorageMap(Storage.CurrentContext, pairIdPrefix);
        pairToPairIdMap.Put(tokenA + tokenB + version, pairId);
        return pairId;
    }
}
```

The pair id is stored relative to tokenA, tokenB and version

Proof of Concept

To illustrate this vulnerability, the unit test for FlamingoOrderBookTests has been modified so that two pairs are created during initialization. It is subsequently verified that the unit test fails because for the first pair it returns the baseToken and the quoteToken of the last pair added.

```
InitializeContract([
    (fWBTCContract.Hash, fUSDTContract.Hash, treeBitLength, pricePrecision)
    , (FLMContract.Hash, fUSDTContract.Hash, treeBitLength, pricePrecision)
]);

Assert.AreEqual(fWBTCContract.Hash, Contract.GetBaseToken(1));
Assert.AreEqual(fUSDTContract.Hash, Contract.GetQuoteToken(1));
```

This results in an error, as the value for FLM has been stored in the location previously occupied by fWBTC.

```
Failed InitializeTest [806 ms]
Error Message:
    Assert.AreEqual failed. Expected:<0xabdf11d969c0982bf0eeec40b9f3a5a0af8809c2>. Actual
: <0xfa41012407cfc56bb2778d61e10193dc93813ba2>.
```

Recommendations

- It is advisable to implement a global counter for pair IDs in order to ensure that each new pair receives a unique identifier, preventing any potential overwriting issues.
- Update the logic in `PairIdStorage` so that the `pairId` does not depend on the tokens `tokenA` and `tokenB` but is instead derived from a global counter that ensures the uniqueness of each pair.

Source Code References

- [src/Flamingo.OrderBook/FlamingoOrderBook.Owner.cs#L125](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Storage.cs#L221-L235](#)

Fixes Review

This issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/pull/20>

Misassigned Token Decimals

Identifier	Category	Risk	State
FOB2-03	Business Logic	High	Fixed

In the current `FlamingoOrderBook` contract code, the decimals of the `baseToken` and `quoteToken` tokens are incorrectly assigned, causing the decimal values to be inverted. Specifically, the value of `quoteTokenDecimals` is being assigned using `baseToken`, and the value of `baseTokenDecimals` is being assigned using `quoteToken`. This inversion in the assignment of decimals can result in erroneous calculations in token operations, affecting the accuracy of the transactions.

```
Pair pair = new Pair()
{
    Id = pairId,
    BaseToken = baseToken,
    QuoteToken = quoteToken,
    TreeWidth = (int)GetTreeBitLength(pairId),
    PricePrecision = GetPricePrecision(pairId),
    QuoteTokenDecimals = GetDecimals(baseToken),
    BaseTokenDecimals = GetDecimals(quoteToken),
};
```

Decimals of inverted tokens

This issue can cause incorrect calculations in token operations because the decimals are incorrectly assigned. This could lead to erroneous transactions, loss of precision in the managing of tokens and possible financial losses for users, impacting the integrity of operations and trust in the system. It is essential to correct it to ensure accuracy in calculations and correct representation of token values.

Recommendations

- Modify the code to ensure that the `quoteTokenDecimals` are assigned using `quoteToken` and the `baseTokenDecimals` are assigned using `baseToken`. This correction ensures that future calculations are accurate and transactions are completed correctly.
- Improve the specific unit tests that verify the correct assignment of decimals for each token in the pairs. This will ensure that any similar errors in the future are detected before they can impact the system.

Source Code References

- [src/Flamingo.OrderBook/FlamingoOrderBook.Sell.cs#L42-L43](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Sell.cs#L96-L97](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Sell.cs#L142-L143](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Buy.cs#L42-L43](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Buy.cs#L109-L110](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Buy.cs#L159-L160](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Buy.cs#L227-L228](#)

Fixes Review

This issue has been addressed in the following commit:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/commit/e6e8fb13ec03ac76ae25ceb3b8863cef44b73e1a>

Bypass Contract Caller

Identifier	Category	Risk	State
FOB2-04	Access Controls	High	Open

The `AddLiquidity`, `RemoveLiquidity`, `SwapTokenInForTokenOut`, and `SwapTokenOutForTokenIn` methods in the `FlamingoSwapRouterContract` unnecessarily duplicate code in overloads that do not include the `sender` argument. Furthermore, in the methods that do have it, the restriction of verifying that the `sender` is a contract is not implemented. This allows an external contract to exploit this bypass to evade the expected check and make calls that should be restricted to contracts.

```
//验证权限
var caller = Runtime.CallingScriptHash;
Assert(ContractManagement.GetContract(caller) != null, "Forbidden");
```

Non-existent condition in certain overloads

It should be noted that the `RemoveLiquidity` method is slightly different and transfers the funds first to the contract, and then transfers them from the contract to the `pairContract`, something redundant that has been optimized in the other overload.

Recommendations

- Reuse the code calling with `CallingScriptHash` as `sender` and add the check that it is not a contract in the function that has the `sender` argument.

Source Code References

- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L78](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L163](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L325](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L373](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/34>

Vulnerable NoReentrantAttribute

Identifier	Category	Risk	State
FOB2-05	Timing and State	High	Open

Using the `NoReentrant` attribute using SDK version **3.7.4** or lower is vulnerable to reentrancy attacks where the attacker does not call the same method but abuses the contract by calling another method also protected by the `NoReentrant` attribute. The problem is that this attribute uses a storage key based on the name of the method it protects, unlike Solidity where it is global for the entire contract.

Imagine a scenario where there are `swap`, `deposit`, and `withdraw` methods, each protected by the `NoReentrant` attribute. An attacker could exploit this by calling the `swap` method and, during a reentrancy, invoke either the `deposit` or `withdraw` method. Although all methods are individually protected by the `NoReentrant` attribute, the attacker would still be able to call each method once more. This loophole could allow for a reentrancy attack, where the attacker manipulates liquidity changes in real-time, creating an economic imbalance and benefiting from it.

Recommendations

- Use versions higher than **3.7.4** of `neo-devpack-dotnet`, where the behavior of this attribute has been modified to be global.

References

- <https://github.com/neo-project/neo-devpack-dotnet/issues/1182>
- <https://github.com/neo-project/neo-devpack-dotnet/pull/1181>

Source Code References

- [src/Flamingo.Broker/Flamingo.Broker.csproj#L10](#)
- [src/Flamingo.FToken/Flamingo.FToken.csproj#L10](#)
- [src/Flamingo.TokenMigrationRouter/Flamingo.TokenMigrationRouter.csproj#L10](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/44>

Wrong Symbol in FlamingoSwapPairContract

Identifier	Category	Risk	State
FOB2-07	Business Logic	Medium	Open

In the **FlamingoSwapPairContract** contract, the `Symbol` is defined statically in the contract. This means that it does not change depending on the tokens that make up the exchanging pair, because the values of **Token0** and **Token1** are defined during the `_deploy` method.

This creates a problem as all pair contracts on **FlamingoSwap** will have the same symbol, which does not correctly reflect the tokens being exchanged.

Ideally, the symbol of the pair should be generated dynamically based on the **Token0** and **Token1** tokens associated with that particular pair. For example, by concatenating the symbols of both tokens in the format `"FLP-Token0-Token1"`, so that each pair can be easily identifiable.

```
[Safe]
public static string Symbol() => "FLP-fWBTC-fUSDT"; //symbol of the token
```

Hardcoded Symbol value

This situation can cause Flamingo users to become confused when interacting with the different exchanging pairs, since they would all have the same symbol. In addition, it can also have an impact on Flamingo's GUI.

Recommendations

- It is advisable to modify the contract so that during deployment, the symbol is automatically generated using the token symbols of **Token0** and **Token1**.

Source Code References

- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L19](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/35>

Wrong Reentrancy Protection

Identifier	Category	Risk	State
FOB2-08	Business Logic	Medium	Fixed

The **FToken** contract implements reentrancy protection using the block hash (`Ledger.CurrentHash`) to prevent repeated executions within the same block.

```
Storage.Put(Storage.CurrentContext, Prefix_Tx.Concat(Ledger.CurrentHash), 1);
```

However, this solution is inadequate since the block hash is the same for all transactions within the same block, which will prevent other legitimate users whose transactions are in the same block from using the contract.

Recommendations

- Instead of using `Ledger.CurrentHash`, `Runtime.Transaction.Hash` should be used, which is the unique hash of the current transaction.
 - `Storage.Put(Storage.CurrentContext, Runtime.Transaction.Hash, 1);`

Source Code References

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/commit/cf57c9d2a83f4dd7c29bc762035915e1dc9cbc62>

Fixes Review

This issue has been addressed in the following commit:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/commit/e21a092cdab4e7c0e4e1b5a17fcef8874b56e6c7>

Denial of Service in the Query Methods

Identifier	Category	Risk	State
FOB2-09	Denial of Service	Low	Open

Different query methods of the contract return a list of characteristics without considering that Neo's virtual machine has a limitation of 2048 elements in the return of the operations and a Denial of Service could occur with higher values.

Taking into consideration a scenario where these query methods fail at some point by exceeding the limits of the virtual machine and returning FAULT in its execution, a Denial of Service would be produced, this limits the affected methods to fewer than 600 entries.

Source Code References

- [src/Flamingo.FLUND/FLUND.cs#L297](#)
- [src/Flamingo.FLUND/FLUND.cs#L308](#)
- [src/Flamingo.FLUND/FLUND.cs#L383](#)
- [src/Flamingo.FLUND/FLUND.cs#L394](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L107](#)
- [src/Flamingo.Broker/FlamingoBroker.Account.cs#L63](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/47>

Limit Call Rights

Identifier	Category	Risk	State
FOB2-10	Design Weaknesses	Low	Open

It is important to highlight that in certain cases, the witnesses scope extends beyond the invoked contracts and that there is a possibility that the invoked contract makes a reentrancy. Therefore, it is advisable to use the principle of least privilege (PoLP) during all the external processes or the calls to the contracts.

Thus, when making the call to any contract it is expected to be read-only; as it is the case of obtaining the user's balance, this call should always be made with the `ReadOnly` flag instead of `CallFlags.All`.

```
private static bool GetReversed(UInt160 fromToken, UInt160 toToken, UInt160 pair)
{
    var token1 = (UInt160) Contract.Call(pair, "getToken1", CallFlags.All);
    return fromToken == token1;
}
```

Recommendations

- Use `ReadOnly` whenever possible for any external call to a contract where permissions to modify states are not required.

References

- Principle of Least Privilege

Source Code References

- [src/Flamingo.Lend/FlamingoPriceFeed.cs#L168](#)
- [src/Flamingo.Lend/FlamingoPriceFeed.cs#L176](#)
- [src/Flamingo.FLUND/Router.cs#L81](#)
- [src/Flamingo.FLUND/FLUND.cs#L39](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/50>

Wrong Timestamp Calculation

Identifier	Category	Risk	State
FOB2-11	Business Logic	Low	Open

In the **FlamingoSwapPairContract** contract, a 32-bit timestamp is used to calculate the time between updates, limiting the value to a range of 0 to 2^{32} (4294967296). This approach can cause problems when the timestamp exceeds that limit and is reset, which can lead to incorrect calculations of elapsed time, preventing cumulative prices from updating. The problem lies in the lack of proper handling of this reset, which affects the accuracy and correct functioning of the contract.

```
private static void Update(BigInteger balance0, BigInteger balance1, ReservesData reserve)
{
    BigInteger blockTimestamp = Runtime.Time / 1000 % 4294967296;
    var priceCumulative = Cumulative;
    BigInteger timeElapsed = blockTimestamp - Cumulative.BlockTimestampLast;
    if (timeElapsed > 0 && reserve.Reserve0 != 0 && reserve.Reserve1 != 0)
    {
        priceCumulative.Price0CumulativeLast += reserve.Reserve1 * FIXED * timeElapsed / reserve.Reserve0;
        priceCumulative.Price1CumulativeLast += reserve.Reserve0 * FIXED * timeElapsed / reserve.Reserve1;
        priceCumulative.BlockTimestampLast = blockTimestamp;
        Cumulative = priceCumulative;
    }
    reserve.Reserve0 = balance0;
    reserve.Reserve1 = balance1;

    ReservePair = reserve;
    Synced(balance0, balance1);
}
```

Wrong timeElapsed computation

This logic failure can lead to the following problems:

- **Timestamp overflow:**

Calculating `blockTimestamp` with module 4294967296 can cause problems if the time between `blockTimestampLast` and current exceeds that limit and the timestamp "flips over". In that case, `timeElapsed` would be negative, preventing the cumulative prices from updating, even if time has passed.

- **Reset time:**

Using 32-bit timestamps means that time will "reset" every 2^{32} seconds, which is approximately 136 years. Although this seems like a long time, it is important to note that if the contract has a long lifespan or is run in quick simulations (such as testing), this restart can occur and cause incorrect calculations.

Recommendations

- It is convenient to eliminate the `timeElapsed` calculation module since the neo virtual machine works with `BigIntegers`.

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/36>

Unnecessary Trust Required

Identifier	Category	Risk	State
FOB2-12	Design Weaknesses	Low	Open

The design of the `FlamingoBroker` contract requires users to explicitly call the `deposit` method in order to make deposits. This means that the `OnNEP17Payment` method does not automatically process deposits on behalf of users transferring tokens.

```
/// <summary>
/// Called when NEP17 tokens are transferred to the contract.
/// </summary>
/// <param name="from">The address from which the transfer is happening.</param>
/// <param name="amount">The amount transferred.</param>
/// <param name="data">The array of data to be passed as arguments [action: BigInteger] always first.</param>
public static void OnNEP17Payment(UInt160 from, BigInteger amount, object[] data)
{
    // Allow receiving NEP17 tokens only if deposit enabled for them
    ExecutionEngine.Assert(IsTokenDepositEnabled(Runtime.CallingScriptHash), "Token deposit not enabled");
}
```

Method `OnNep17Payment`

This design introduces several potential issues:

- **Loss of tokens by users:** Users who directly transfer tokens to the contract will lose their tokens, since no accounting or traceability of said deposit is performed by the contract. Therefore, any user who transfers tokens directly to the contract will have left the tokens locked in it.
- **Increased trust requirements:** It forces users to sign transactions with a specific `Scope` where they give the broker's contract access to use their signature for the token to be transferred. This `Scope` does not specify how the user's signature will be used, so the contract can transfer more tokens than the user initially thought or believed to have signed.
- **Loss of versatility:** The contract is much more flexible, versatile, and manageable, if to deposit the tokens they only have to be transferred by the user to the smart contract, although the deposit method is maintained, having the possibility of making it simpler, and with fewer trust requirements will always facilitate the use of the contract by users.

Recommendations

- Allow deposits to the contract using the `OnNEP17Payment` method.

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.External.cs#L23](#)
- [src/Flamingo.Broker/FlamingoBroker.Account.cs#L26](#)

Lack of Inputs Validation

Identifier	Category	Risk	State
FOB2-13	Data Validation	Low	Open

Certain methods of the different contracts in the Flamingo project do not properly check the arguments, which can lead to major errors.

In some cases the inputs are not being checked nor the integrity of some the `UInt160` types, and in some cases, there are integers with negative value.

It is advisable to check the `UInt160` types (`hash.IsValid && !hash.IsZero`) whenever possible and using only `IsValid` in cases where it is strictly necessary.

Additionally, depending on the method's requirements, it is important to verify whether the value represents a contract, such as when working with `exchangeContractHash`.

Recommendations

It is advisable to always check the format of the arguments before using their value, otherwise, a user could send unexpected values through these arguments, being able to make injections or arbitrary reads from the storage, either intentionally or not.

Source Code References

- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.cs#L51](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L61](#)
- [src/Flamingo.SwapPairWhiteList/FlamingoSwapPairWhiteList.cs#L29](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs#L59](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs#L88](#)
- [src/Flamingo.FToken/FToken.cs#L83](#)
- [src/Flamingo.FToken/FToken.cs#L70](#)
- [src/Flamingo.FLUND/FLUND.cs#L599](#)
- [src/Flamingo.Broker/FlamingoBroker.Account.cs#L59](#)
- [src/Flamingo.Broker/FlamingoBroker.Account.cs#L65](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L38](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L62](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L74](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L85](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L96-L120](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L145](#)
- [src/Flamingo.Broker/FlamingoBroker.cs#L168-L174](#)
- [src/Flamingo.Broker/FlamingoBroker.Order.cs#L240](#)
- [src/Flamingo.Broker/FlamingoBroker.Order.cs#L246](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L27](#)

- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L57](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L77](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L99](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L121](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L144](#)
- The Order argument might not have all the expected fields:
 - [src/Flamingo.Broker/FlamingoBroker.Order.cs#L114](#)
- It should be checked that the address is a contract:
 - [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L58](#)
- Is not checked that both arrays have the same length:
 - [src/Flamingo.FLUND/Router.cs#L141](#)
- Lack of IsValid:
 - [src/Flamingo.Lend/FlamingoPriceFeed.cs#L299](#)
- Lack of zero check:
 - [src/Flamingo.FLUND/FLUND.cs#L619](#)
 - [src/Flamingo.Broker/Utils/ContractUtils.cs#L13](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/37>

Wrong GetPaymentStatus Logic

Identifier	Category	Risk	State
FOB2-14	Business Logic	Low	Open

An issue has been identified in the logic of the `GetPaymentStatus` method. The method is currently designed to check if a stored value equals 1 (numeric), but due to a type handling bug (casting), it always returns `false`. This happens because a numeric value is being compared with a `ByteString`, causing the condition to fail even when the value stored by the `Enable` method is correct. As a result, the method never returns `true`.

```
public static void Enable() => new StorageMap(Storage.CurrentContext, mapName).Put("enable", 1);

public static void Disable() => new StorageMap(Storage.CurrentContext, mapName).Put("enable", 0);

public static void Reduce(UInt160 key, BigInteger value)
{
    var oldValue = Get(key);
    if (oldValue == value)
        Remove(key);
    else
        Put(key, oldValue - value);
}

public static void Put(UInt160 key, BigInteger value) => new StorageMap(Storage.CurrentContext, mapName).Put(key, value);

public static BigInteger Get(UInt160 key)
{
    var value = new StorageMap(Storage.CurrentContext, mapName).Get(key);
    return value is null ? 0 : (BigInteger)value;
}

public static bool GetPaymentStatus() => new StorageMap(Storage.CurrentContext, mapName).Get("enable").Equals(1);
```

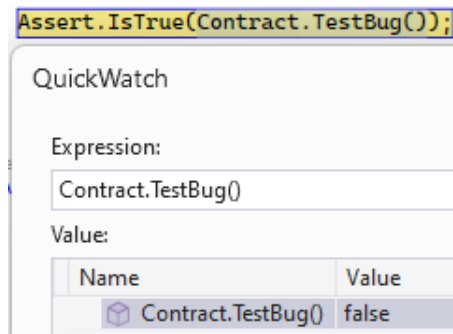
Verification of `GetPaymentStatus`

Proof of Concept

If we create a contract with the following code:

```
public static bool TestBug()
{
    var storage = new StorageMap(Storage.CurrentContext, 0x01);
    storage.Put("enable", 1);
    return storage.Get("enable").Equals(1);
}
```

When we run the unit test, we can see that it is not `true`, confirming that the `GetPaymentStatus` method returns an incorrect value.



Unit test execution

The risk of this issue has been reduced from critical to low, as the `GetPaymentStatus` method is currently unused in the project (dead code). However, it still presents a potential risk, as future integration of this method into the contract's logic could lead to significant failures due to its malfunction.

Source Code References

- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Nep17.cs#L106](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/38>

Wrong NEP-17 Standard

Identifier	Category	Risk	State
FOB2-15	Codebase Quality	Low	Open

The NEP17 standard establishes the methods to be implemented by the project, however a discrepancy has been found in the audited project.

It is important to highlight that the NEP17 standard establishes the following:

"The function MUST return false if the from account balance does not have enough tokens to spend."

However, this function has been developed with the return of an error instead, through the Assert shown below:

```
Assert(AssetStorage.Get(from) >= amount, "Insufficient balance.");
```

Recommendations

- Return false if the user has insufficient balance to complete the transfer.

References

- <https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki>

Source Code References

- [Flamingo.SwapPair/FlamingoSwapPairContract.Nep17.cs#L34](#)
- [src/Flamingo.FLUND/FLUND.cs#L121](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/39>

FToken Requires Minter to Work

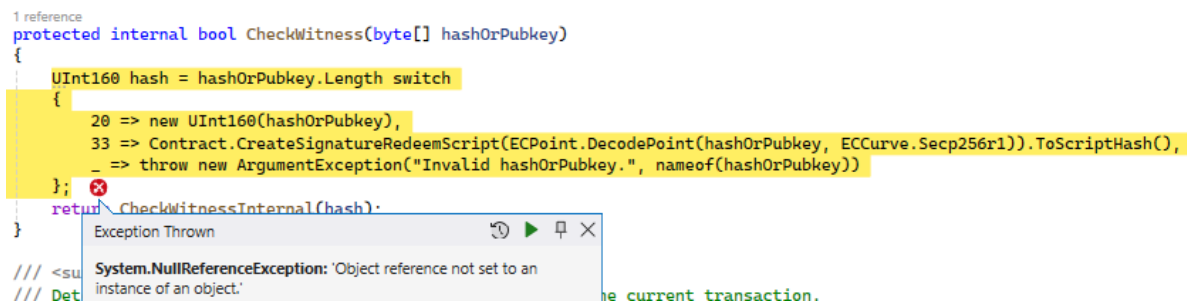
Identifier	Category	Risk	State
FOB2-16	Denial of Service	Low	Open

The Flamingo.FToken.FToken contract performs the following check `IsMinter() || IsOwner()` in order to verify whether it has the necessary authorization to mint. However, if the minter has not been defined, the function is denied even for the administrator, preventing it from being called.

The problem is that if the minter is not defined, the `IsMinter` function (which will be called to check if it is authorized) performs a check of the *Witness* with `null`.

```
private static bool IsMinter() => Runtime.CheckWitness(GetMinter());
```

Throwing an error in the syscall.



```
1 reference
protected internal bool CheckWitness(byte[] hashOrPubkey)
{
    UInt160 hash = hashOrPubkey.Length switch
    {
        20 => new UInt160(hashOrPubkey),
        33 => Contract.CreateSignatureRedeemScript(ECPPoint.DecodePoint(hashOrPubkey, ECCurve.Secp256r1)).ToScriptHash(),
        _ => throw new ArgumentException("Invalid hashOrPubkey.", nameof(hashOrPubkey))
    };
    return CheckWitnessInternal(hash);
}

/// <summary>
/// Details the current transaction.
```

Recommendations

- Define the minter in `_deploy` or modify the authentication logic.

Source Code References

- [src/Flamingo.FToken/FToken.cs#L27](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/49>

Reentrancy Pattern

Identifier	Category	Risk	State
FOB2-17	Timing and State	Low	Open

The Reentrancy attack is a vulnerability that occurs when external contract calls can make new calls to the calling contract before the initial execution is completed. For a function, this means that the state of the contract could change in the middle of its execution as a result of a call to an untrusted contract or the use of a low-level function with an external address.

This attack is possible in N3 because the NEP17 and NEP11 standards establish that after sending tokens to a contract, the `onNEPXXPayment()` method must be invoked. Therefore, transfers to contracts will invoke the execution of the payment method of the recipient, and the recipient may redirect the execution to himself or to another contract.

One way to mitigate this vulnerability is to use a checks-effects-interactions pattern in the contract design. In this pattern, the contract first performs all the necessary checks to ensure that the function call is valid and then performs all the state changes before interacting with other contracts or accounts. Another solution may be to implement modifiers that prevent reentry, such as the native `NoReentrant` modifier.

In the case of the `Start` method of the FLUND contract, the tokens are transferred before switch the start flag, so the admin can exploit a reentrancy in this method.

```
public static void Start()
{
    ExecutionEngine.Assert(IsOwner(), "verify fail");
    ExecutionEngine.Assert(!IsStart(), "Started");
    SafeTransfer(FLMHash, Owner, Runtime.ExecutingScriptHash, 10000000000);
    StoragePut(IsStartPrefix, 1);
    Mint(Runtime.ExecutingScriptHash, 10000000000);
}
```

The risk of this vulnerability has been reduced from critical to low, because it is only possible to exploit it by the admin

Recommendations

- **Implementation of the Checks-Effects-Interactions Pattern:** It is essential to always make the state changes in the storage before making transfers or calls to external contracts, in addition to implementing the necessary measures to avoid duplicated calls or chain calls to the methods. A good practice to avoid reentrancy is to organize the code according to the following pattern:
 - The first step is conducting all necessary **checks**.
 - The next step is the application of all **effects**.
 - All external **interactions** take place in the last step.
- **Using the NoReentrant Attribute:** Implement the `NoReentrant` attribute to prevent recursive method invocations. This practice can be particularly effective in protecting key contract functions from reentry vulnerability.

References

- <https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki>
- <https://github.com/neo-project/proposals/blob/master/nep-11.mediawiki>
- <https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern>
- <https://github.com/neo-project/neo-devpack-dotnet/pull/756>

Source Code References

- [src/Flamingo.FLUND/FLUND.cs#L595](#)
- [src/Flamingo.FLUND/FLUND.cs#L441](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/48>

More Restrictive Conditions

Identifier	Category	Risk	State
FOB2-18	Data Validation	Low	Fixed

It is vital to ensure that values not only meet a specific condition but are within a safe and manageable range of values, so it is important to implement conditions that cover uncontrolled cases that could lead to unexpected behavior.

During the code review, a certain lack of Asserts has been detected that can help with contract resilience. Adding conditions that must not be violated under any circumstances helps to avoid cases where an attacker takes advantage of a logic flaw. For example: subtractions with negative numbers, producing increments instead of decrements.

Source Code References

- Check that after the swap the base balance decreases and the quote balance increases, ensuring that the exchange expectations are respected.
 - [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L293-L296](#)
 - [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L262](#)
- It is important to check that the returned values are positive. This prevents undesired behavior.
 - totalBaseAmountReceived and totalQuoteAmountSpent
 - [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L111](#)
 - [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L48](#)
 - [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L224](#)
 - quoteAmountPlaced
 - [src/Flamingo.Broker/FlamingoBroker.Buy.cs#L169](#)
 - totalBaseAmountSpent and totalQuoteAmountReceived
 - [src/Flamingo.Broker/FlamingoBroker.Sell.cs#L48](#)
 - quoteAmountToPlace
 - [src/Flamingo.Broker/FlamingoBroker.Sell.cs#L447](#)
 - totalBaseAmountSpent and totalQuoteAmountReceived
 - [src/Flamingo.Broker/FlamingoBroker.Sell.cs#L105](#)
- It is recommended to cover any negative values with `nodeIndexToCheck < 0`. This ensures that any other negative values will be handled correctly.
 - [src/Flamingo.Broker/FlamingoBroker.Order.cs#L116](#)

Fixes Review

This issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/pull/63>

Missing Data Argument on Update Method

Identifier	Category	Risk	State
FOB2-19	Design Weaknesses	Informative	Assumed

The logic for updating and deploying contracts is centralized in the `_deploy` method. This method accepts a `data` argument from the `Update` method; however, it has been observed that `null` is consistently passed as this value. Consequently, any update relying on the `data` value within the `_deploy` method becomes ineffective.

```
public static void Update(ByteString neffFile, string manifest)
{
    Assert(Verify(), "No authorization.");
    ContractManagement.Update(neffFile, manifest, null);
}
```

Recommendations

- Add the `data` argument to the call made to the `ContractManagement.Update` method.

Source Code References

- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Admin.cs#L72](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L167](#)
- [src/Flamingo.Lend/FTokenVault.cs#L220](#)
- [src/Flamingo.Lend/FlamingoPriceFeed.cs#L48](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs#L70](#)
- [src/Flamingo.FToken/FToken.cs#L94](#)
- [src/Flamingo.FLUND/FLUND.cs#L676](#)

Duplicated Pools

Identifier	Category	Risk	State
FOB2-20	Business Logic	Informative	Closed

A vulnerability has been identified in the `AddPair` method of the Flamingo smart contracts. This method does not ensure a specific order when adding new token pairs (`baseToken` and `quoteToken`) to the storage, allowing the same pair to be added in two different sequences. This can lead to the creation of duplicate pools, which poses several risks, including storage inconsistencies and potential trading errors.

The `AddPair` method technically adds token pairs after performing various validations and related data storage. However, it lacks a mechanism to enforce a consistent order for the tokens, which means a pair could be added as either `(tokenA, tokenB)` or `(tokenB, tokenA)`. This inconsistency allows for the creation of duplicate entries in storage, particularly in `PairIdStorage`, which can cause mechanisms that rely on the unique identification of pairs to fail, thereby compromising the integrity of the **OrderBook** and trading operations.

The primary risk is that these duplicates could be exploited to manipulate the market, carry out unauthorized operations, divide liquidity, or create unfavorable trading conditions. This could undermine the trust and functionality of the Flamingo platform, potentially leading to significant financial and reputational damage.

Recommendations

- Implement an order of the tokens before adding them to storage. For example, always store the pair as `(tokenA, tokenB)` where `tokenA` is the token with the lowest address.
- Add a previous validation that checks if the token pair already exists in any order before proceeding to add a new pair to the storage.
- Modify the methods in `PairIdStorage` so that they always manipulate token pairs in an orderly and consistent manner, ensuring the uniqueness and consistency of pair identifiers.

Source Code References

- [src/Flamingo.OrderBook/FlamingoOrderBook.Owner.cs#L125](#)
- [src/Flamingo.OrderBook/FlamingoOrderBook.Storage.cs#L233-L239](#)

Fixes Review

After reviewing the issue, it has been determined that this behavior is intentional as per the project's design. As a result, the issue is now considered resolved and closed. For further information find the following link:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/19>

Emit Events on State Changes

Identifier	Category	Risk	State
FOB2-21	Auditing and Logging	Informative	Partially Fixed

It is a good practice to emit events when there are significant changes in the states of the contract that can affect the result of its execution by the users.

The changes during the project should emit events so that the potential actors monitoring the blockchain; such as dApps, automated processes and users, can be notified of these significant state changes.

List of main methods that make significant changes to smart contract parameters and should emit events:

- **FlamingoSwapPair:**
 - `SetAdmin.`
 - `SetGASAdmin.`
 - `SetWhitelListContract.`
 - `SetFundAddress.`
 - `Enable.`
 - `Disable.`
- **FlamingoBroker.Owner:**
 - `AddPair.`
 - `SetPairMakerFee.`
 - `SetPairTakerFee.`

Recommendations

- Consider issuing events to notify of changes in the contract values that may affect the relationship of the users with the contract.

Source Code References

- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L53](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L80](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L107](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L146](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L164](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Nep17.cs#L85-L87](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L123](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L203](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L217](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/51>

And the issue has been partially addressed in the pull request:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/pull/67>

Outdated Framework

Identifier	Category	Risk	State
FOB2-22	Outdated Software	Informative	Open

The N3 C# compiler frequently launches new versions of the framework. Using an outdated version can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version.

It has been verified that the project uses Neo's framework 3.6.2-CI00581. This version can be updated containing major improvements.

```
<ItemGroup>
  <PackageReference Include="Neo.SmartContract.Framework" Version="3.6.2-CI00581" />
</ItemGroup>
```

The compiler version can be updated, containing major improvements, as can be seen in the following image.

neo - Neo.SmartContract.Framework 3.7.4-CI00649

Neo.SmartContract.Framework

[NuGet \(PM Console\)](#) [NuGet.exe](#) [.NET CLI](#) [.csproj](#) [Paket](#) [Chocolatey](#) [PowerShellGet](#)

```
PM> Install-Package Neo.SmartContract.Framework -Version 3.7.4-CI00649 -Source
https://www.myget.org/F/neo/api/v3/index.json
```

Recommendations

- Use the latest version of the framework.

References

- <https://www.nuget.org/packages/Neo.SmartContract.Framework/>

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/40>

GAS Optimization

Identifier	Category	Risk	State
FOB2-23	Codebase Quality	Informative	Open

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On the N3 blockchain, GAS is an execution fee which is used to compensate the network for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to arithmetical operations and comparisons.

CheckWitness Already Verifies CallingScriptHash

The CheckWitness method already verifies if the argument is the CallingScriptHash, making any additional checks in the code redundant and unnecessary.

Source Code References

- [src/Flamingo.FLM/FLM.Asset.cs#L29](#)
- [src/Flamingo.Nep17/Nep17Token.cs#L56](#)
- [src/Flamingo.Nep17/Nep17Token.cs#L99](#)

Static Variables

The static variables in N3 are processed at the beginning of the execution of the smart contract, so they must be carefully used. Although these static variables are not used for the call that will be made, their initialization will be processed anyway, and they will occupy elements in the stack with the corresponding cost that this entails.

It is a good practice to reduce the static variables, either through constants or through methods that return the desired value.

Source Code References

- [src/Flamingo.FLM/FLM.Storage.cs](#)
- [src/Flamingo.Lend/FTokenVault.cs](#)
- [src/Flamingo.Lend/FlamingoPriceFeed.cs](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs](#)

- [src/Flamingo.Lend/Ftoken.cs](#)
- [src/Flamingo.FLUND/FLUND.cs](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs](#)

Dead Code

In programming, a part of the source code that is never used is known as dead code. The execution of this type of code consumes more GAS during deployment in something that is not necessary.

The following methods are not used during the execution of the contract, so it would be convenient to either remove it or to use it.

Source Code References

- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Helper.cs#L114](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Helper.cs#L115-L150](#)
- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.Helper.cs#L63-L75](#)
- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.Helper.cs#L83](#)
- [src/Flamingo.FLUND/FLUND.cs#L55](#)
- [src/Flamingo.FLUND/FLUND.cs#L65-L80](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L183](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L205](#)
- [src/Flamingo.Broker/FlamingoBroker.Helpers.cs#L219](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L315](#)
- [src/Flamingo.Broker/FlamingoBroker.Order.cs#L120](#)

Unnecessary Assert Method

Duplicate and unnecessary code has been identified due to the implementation of an Assert method that invokes a native Assert which makes this implementation redundant. It is advisable to remove this custom method and directly use the native method.

Source Code References

- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Helper.cs#L18-L40](#)
- [src/Flamingo.SwapPairWhiteList/FlamingoSwapPairWhiteList.Helper.cs#L15](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Helper.cs#L22-L44](#)
- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.Helper.cs#L15-L37](#)
- [src/Flamingo.Lend/FlamingoPriceFeed.cs#L291](#)

Unnecessary Return

The methods mentioned below always return true, so it is more optimal to eliminate the return in terms of GAS.

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.External.cs#L36](#)

Storage Optimization

It has been identified that `Decrease` methods can be optimized, saving GAS accordingly. When the value of `newAmount` turns out to be 0, the record remains in storage, taking up unnecessary space and affecting storage efficiency. Keeping data without significant value can lead to inefficient use of storage resources.

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L103](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L187](#)

Redundant Code

It has been identified that redundant checks are being performed, for example first checking if it is a contract using `ContractUtils.IsContract(token)`, and then checking if it is a valid address using `ContractUtils.IsValidAddress(token)`. The second check, `IsValidAddress`, is completely redundant, since if the token is a valid contract, it already has a valid address.

This introduces unnecessary duplication in the code, which can affect clarity and unnecessary GAS cost, without providing any additional value.

```
public static void EnableTokenDeposit(UInt160 token)
{
    ExecutionEngine.Assert(HasOwnerWitness(), "No owner witness");
    ExecutionEngine.Assert(ContractUtils.IsContract(token), "Token not a contract");
    ExecutionEngine.Assert(ContractUtils.IsValidAddress(token), "Invalid token address");

    TokenDepositEnabledStorage.Put(token, true);
}
```

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L45](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L58](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L72](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L85](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L98](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L111](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L127](#)

Logic Optimization

The logic related to the `RequestTransfer` function in the `FlamingoSwapRouterContract.Helper` file can be optimized if, the condition that checks the result of `approvedTransfer` is moved one line above, so that it is made immediately after the call, since if this method returns `false`, it must not be continued in any way.


```
private static void RequestTransfer(UInt160 token, UInt160 from, UInt160 to, BigInteger amount)
{
    try
    {
        var balanceBefore = (BigInteger)Contract.Call(token, "balanceOf", CallFlags.ReadOnly, new object[] { to });
        var result = (bool)Contract.Call(from, "approvedTransfer", CallFlags.All, new object[] { token, to, amount, null });
        var balanceAfter = (BigInteger)Contract.Call(token, "balanceOf", CallFlags.ReadOnly, new object[] { to });
        Assert(result, "Transfer Not Approved in Router", token);
        Assert(balanceAfter == balanceBefore + amount, "Unexpected Transfer in Router", token);
    }
    catch (Exception)
    {
        Assert(false, "Transfer Error in Router", token);
    }
}
```

Verification before continuing

Local variables have also been found that are defined before their use. Moving these variables can optimize GAS in certain cases.

```
var me = Runtime.ExecutingScriptHash;

Assert(amount0Out >= 0 && amount1Out >= 0, "Invalid AmountOut");
Assert(amount0Out > 0 || amount1Out > 0, "Invalid AmountOut");

var r = ReservePair;
var reserve0 = r.Reserve0;
var reserve1 = r.Reserve1;

//转出量小于持有量
Assert(amount0Out < reserve0 && amount1Out < reserve1, "Insufficient Liquidity");
//禁止转到token本身的地址
Assert(toAddress != (UInt160)Token0 && toAddress != (UInt160)Token1 && toAddress != me, "INVALID_TO");
```

When INITIAL_SUPPLY is 0 (currently it is), the `_deploy` method should not emit a transfer event or write to the owner's BALANCE_MAP.

```
public static void _deploy(object data, bool update)
{
    if (!update)
    {
        Storage.Put(ctx, SUPPLY_KEY, INITIAL_SUPPLY);
        Storage.Put(ctx, OWNER_KEY, INITIAL_OWNER);

        BALANCE_MAP.Put(INITIAL_OWNER, INITIAL_SUPPLY);
        OnTransfer(null, INITIAL_OWNER, INITIAL_SUPPLY);
    }
}
```

Source Code References

- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Helper.cs#L90](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.cs#L146](#)
- [src/Flamingo.Lend/Ftoken.cs#L61-L62](#)

New Variable Cache

In the code, repetitive calculations or calls are being made to retrieve values that remain constant within the same context. This approach unnecessarily increases execution costs. To optimize performance, it is recommended to cache the results of these repetitive calculations or calls by storing them in variables. This allows the values to be reused without re-executing the operations.

For example, storing the result of the key, in order to avoid performing repeated operations, leading to significant GAS savings.

```
internal static void Put(ByteString treeKey, BigInteger nodeIndex, BigInteger baseAmount, BigInteger quoteAmount)
{
    StorageMap storageMap = new StorageMap(Storage.CurrentContext, cancelledBaseAmountAtPricePrefix);
    storageMap.Put(Key(treeKey, nodeIndex) baseAmount);

    StorageMap storageMap2 = new StorageMap(Storage.CurrentContext, cancelledQuoteAmountAtPricePrefix);
    storageMap2.Put(Key(treeKey, nodeIndex) quoteAmount);
}
```

Source Code References

- [Flamingo.Broker/FlamingoBroker.Storage.cs#L794-L798](#)
- [Flamingo.Broker/FlamingoBroker.Storage.cs#L807-L810](#)

New Redundant Check Operations

During the audit, redundant checks were identified that do not provide additional value, as certain conditions are implicitly validated by others. These redundancies not only increase code complexity but can also lead to unnecessary GAS costs.

For example, in the `DisableTokenWithdraw` method, the `IsContract` method, which internally calls `ContractManagement.GetContract`, already verifies that the address is valid.

```
public static void DisableTokenWithdraw(UInt160 token)
{
    ExecutionEngine.Assert(HasOwnerWitness(), "No owner witness");
    ExecutionEngine.Assert(ContractUtils.IsContract(token), "Token not a contract");
    ExecutionEngine.Assert(ContractUtils.IsValidAddress(token), "Invalid token address");

    TokenWithdrawEnabledStorage.Put(token, false);
}
```

Source Code References

- [Flamingo.Broker/FlamingoBroker.Owner.cs#L20](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L33](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L46](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L59](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L72](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L86](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L99](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L112](#)

- [Flamingo.Broker/FlamingoBroker.Owner.cs#L125](#)
- [Flamingo.Broker/FlamingoBroker.Owner.cs#L141](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/62>

Bad Coding Practices

Identifier	Category	Risk	State
FOB2-24	Codebase Quality	Informative	Open

During the smart contract audit, certain bad practices have been detected throughout the code that should be improved. It is strongly advised to consider the adoption of enhanced coding styles and best practices for overall code improvement.

Compressive Initial Values

When you want to define a `BigInteger` value as a static initial value, it is convenient to use `IntegerAttribute` instead of `InitialValueAttribute`, since otherwise the initial value must be in hexadecimal, reducing the readability and understanding of the code.

References

- [src/Neo.SmartContract.Framework/Attributes/IntegerAttribute.cs](#)

Source Code References

- [src/Flamingo.FLM/FLM.cs#L17](#)
- [src/Flamingo.FLUND/FLUND.cs#L24](#)

Homogeneous Verification

It has been observed that the method for verifying if the caller is the owner varies throughout the code. In occasions, the `Verify` method is used, while other times the admin's `Witness` is checked. Although both methods produce the same result, for better readability, maintainability, and gas efficiency, it is advisable to standardize on a single verification approach.

Similarly, there are times where in the verification of `tokenA` and `tokenB` it is checked that the entry is an `IsAddress`, and in other places it is only verified that they are different from each other. Standardizing these checks will improve code consistency and clarity.

```
public static ByteString GetExchangePair(UInt160 tokenA, UInt160 tokenB)
{
    Assert(tokenA != tokenB, "Identical Address", tokenA);
    return StorageGet(GetPairKey(tokenA, tokenB));
}

/// <summary>
/// 增加Nep17资产的exchange合约映射
/// </summary>
/// <param name="tokenA">Nep5 tokenA</param>
/// <param name="tokenB">Nep5 tokenB</param>
/// <param name="exchangeContractHash"></param>
/// <returns></returns>
public static bool CreateExchangePair(UInt160 tokenA, UInt160 tokenB, UInt160 exchangeContractHash)
{
    Assert(Runtime.CheckWitness(GetAdmin()), "Forbidden");
    Assert(tokenA.IsAddress() && tokenB.IsAddress(), "Invalid Address");
    Assert(tokenA != tokenB, "Identical Address", tokenA);
    var key = GetPairKey(tokenA, tokenB);
    var value = StorageGet(key);
    Assert(value == null || value.Length == 0, "Exchange Already Existed");

    StoragePut(key, exchangeContractHash);
    onCreateExchange(tokenA, tokenB, exchangeContractHash);
    return true;
}
```

Source Code References

- Verify ownership in the same way:
 - [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Admin.cs#L72](#)
 - [src/Flamingo.SwapPairWhiteList/FlamingoSwapPairWhiteList.Admin.cs#L50](#)
 - [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L166](#)
 - [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L149](#)
- Perform the different token checks in the same way:
 - [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.cs#L31](#)
 - [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.cs#L45](#)
 - [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.cs#L69](#)
- Maintain the verifications in a homogeneous manner, that is, always in private or public methods.
 - [src/Flamingo.FLUND/FLUND.cs#L356](#)

Refactor File Organization

A poor organization of the different `getter` methods of the contract has been identified. Currently, these methods are located in the file dedicated to the logic related to the owner, which is not the best place, these methods are usually general purpose and not necessarily related to the specific functionalities of the owner, so it is advisable to move these methods to a different file dedicated to data access logic or general utilities.

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L238-L308](#)

Use `NoReentrant` Attribute

Instead of implementing a custom reentrancy prevention system, it is advisable to use the `NoReentrant` attribute. This attribute ensures that storage is cleaned when exiting the method, regardless of the exit path. Using your own method introduces the risk of human error, which could potentially leave the method or contract permanently locked if the flag is not correctly reverted.

References

- <https://github.com/neo-project/neo-devpack-dotnet/blob/master/src/Neo.SmartContract.Framework/Attributes/NoReentrantAttribute.cs>

Source Code References

- [src/Flamingo.SwapPair/FlamingoSwapPairContract.cs#L115](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/52>

Safe Storage Access

Identifier	Category	Risk	State
FOB2-25	Design Weaknesses	Informative	Open

N3 contains different types of storage access, being `CurrentReadOnlyContext` the most appropriate one for the read-only methods; using a read-only context prevents any malicious change to the states. As in the rest of the cases, it is important to follow the principle of least privilege (PoLP) in order to avoid future problems.

Recommendations

- Use `CurrentReadOnlyContext` whenever the context is used only for reading purposes.

References

- Principle of Least Privilege

Source Code References

- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.Helper.cs#L45-L55](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.cs#L123](#)
- [src/Flamingo.SwapPairWhiteList/FlamingoSwapPairWhiteList.Helper.cs#L29-L40](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Helper.cs#L101](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Helper.cs#L83-L88](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Nep17.cs#L74](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Nep17.cs#L102](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Nep17.cs#L106](#)
- [src/Flamingo.FToken/FToken.cs#L47](#)
- [src/Flamingo.FLUND/FLUND.cs#L57-L62](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/53>

Lack of Safe Method Attribute

Identifier	Category	Risk	State
FOB2-26	Design Weaknesses	Informative	Partially Fixed

In N3 there is a `Safe` attribute which defines that the call to the contract will create an execution context where the storage will not be modifiable or able to produce notifications. This characteristic turns the `Safe` methods into secure query methods.

```
if (method.Safe)
{
    flags &= ~(CallFlags.WriteStates | CallFlags.AllowNotify);
}
```

Additionally, it will provide the wallets and dApps with the necessary information to identify it as a query method and to make a reading invocation with no GAS costs. Therefore, it is convenient to establish our query methods as `Safe` to keep the Principle of Least Privilege.

Recommendations

- It is convenient to add the `Safe` attribute to methods that do not make any changes to the storage.

References

- Principle of Least Privilege

Source Code References

- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.Admin.cs#L19](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L35](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L42](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L74](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L96](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L120](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L135](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Admin.cs#L23](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Admin.cs#L29](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Admin.cs#L48](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L196](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L210](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L228](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L243](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L264](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.cs#L285](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Helper.cs#L48](#)

- [src/Flamingo.FLUND/Router.cs#L49](#)
- [src/Flamingo.FLUND/Router.cs#L79](#)
- [src/Flamingo.FLUND/Router.cs#L92](#)
- [src/Flamingo.FLUND/Router.cs#L111](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs#L73](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs#L83](#)
- [src/Flamingo.FLUND/FLUND.cs#L27-L37](#)
- [src/Flamingo.FLUND/FLUND.cs#L462](#)
- [src/Flamingo.FLUND/FLUND.cs#L471](#)
- [src/Flamingo.FLUND/FLUND.cs#L505](#)
- [src/Flamingo.FLUND/FLUND.cs#L524](#)
- [src/Flamingo.FLUND/FLUND.cs#L529](#)
- [src/Flamingo.FLUND/FLUND.cs#L610](#)
- [src/Flamingo.FLUND/FLUND.cs#L633](#)
- [src/Flamingo.FLUND/FLUND.cs#L661](#)
- [src/Flamingo.Broker/FlamingoBroker.External.cs#L12](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/54>

And the issue has been partially addressed in the pull request:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/pull/68>

Contracts Management Risks

Identifier	Category	Risk	State
FOB2-27	Governance	Informative	Open

The logical design of the `FlamingoSwapRouterContract` contract introduces certain minor risks that should be reviewed and considered for their improvement.

Owner Front-running

It has been identified that the owner of the contract can modify important values of the contract without the contract being paused. This opens the possibility of a **front-running** attack. The owner could react to user transactions and change the conditions of the contract in their favor before the user transactions are processed, thus managing to steal funds or benefit from third-party operations.

Implementing a time lock for critical changes will protect users against these types of attacks, as well as giving users enough time to react to any changes that could negatively affect their funds or interests.

Source Code References

- [src/Flamingo.Lend/FlamingoPriceFeed.cs#L163](#)

Owner Arbitrary Burn and Mint

The current contract allows the owner to arbitrarily burn and mint user tokens without restrictions. This burn and mint feature, even if intended, puts the integrity of the project and user funds at risk, as the owner or a malicious user who gains access to such an account could arbitrarily alter the tokens of any account.

Source Code References

- [src/Flamingo.Lend/Ftoken.cs#L178](#)
- [src/Flamingo.FToken/FToken.cs#L83](#)

Not compatible with all NEP-17

The NEP17 standard defines the data argument as an optional argument in token transfers. This allows the token to react in different ways based on its value. However, the `SafeTransfer` method always sends null as the value of said argument, which could prevent the use of the contract with tokens whose logic depends on the assigned data value.

```
private static void SafeTransfer(UInt160 token, UInt160 from, UInt160 to, BigInteger amount)
{
    try
    {
        var result = (bool)Contract.Call(token, "transfer", CallFlags.All, new object[] { from, to, amount, null });
        Assert(result, "Transfer Fail in Router", token);
    }
    catch (Exception)
    {
        Assert(false, "Transfer Error in Router", token);
    }
}
```

Source Code References

- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Helper.cs#L67](#)
- [src/Flamingo.TokenMigrationRouter/TokenMigrationRouter.cs#L104](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/55>

Wrong Return in Methods

Identifier	Category	Risk	State
FOB2-28	Business Logic	Informative	Open

The `RemoveExchangePair` or `RemoveRouter` methods in the **FlamingoSwapFactoryContract** and **FlamingoSwapPairWhiteList** contracts currently always return `true`. If no modifications are made because the record is not found, these methods should return `false` instead. Alternatively, if the return value is not necessary, consider removing it from the function.

```
public static bool RemoveExchangePair(UInt160 tokenA, UInt160 tokenB)
{
    Assert(tokenA.IsAddress() && tokenB.IsAddress(), "Invalid Address");
    Assert(Runtime.CheckWitness(GetAdmin()), "Forbidden");
    var key = GetPairKey(tokenA, tokenB);
    var value = StorageGet(key);
    if (value?.Length > 0)
    {
        StorageDelete(key);
        onRemoveExchange(tokenA, tokenB);
    }
    return true;
}
```



RemoveExchangePair method optimization

Source Code References

- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.cs#L96-L98](#)
- [src/Flamingo.SwapPairWhiteList/FlamingoSwapPairWhiteList.cs#L42](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/56>

Unsecured Ownership Transfer

Identifier	Category	Risk	State
FOB2-29	Governance	Informative	Open

The process of modifying an owner is very delicate, as it directly affects the governance of our contract and, consequently, the entire project's integrity. For this reason, it is advised to adjust the owner's modification logic, to one that allows to verify that the new owner is in fact valid and does exist.

In N3, the owner modification process can be conducted in a single transaction without unnecessary prolongation. Transactions in N3 can be signed by multiple accounts and CheckWitness could be called with the proposed owner and the current owner simultaneously, provided the transaction scope is properly configured.

Source Code References

- [src/Flamingo.SwapPairWhiteList/FlamingoSwapPairWhiteList.Admin.cs#L34](#)
- [src/Flamingo.SwapFactory/FlamingoSwapFactoryContract.Admin.cs#L36](#)
- [src/Flamingo.SwapPair/FlamingoSwapPairContract.Admin.cs#L53](#)
- [src/Flamingo.SwapRouter/FlamingoSwapRouterContract.Admin.cs#L40](#)
- [src/Flamingo.Lend/FlamingoPriceFeed.cs#L64](#)
- [src/Flamingo.Lend/Ftoken.cs#L83](#)
- [src/Flamingo.FLUND/FLUND.cs#L642](#)
- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L16](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/57>

Hard-coded Values

Identifier	Category	Risk	State
FOB2-30	Codebase Quality	Informative	Open

During the smart contract audit, certain bad practices have been detected throughout the code that should be improved. It is strongly advised to consider the adoption of improved coding styles and best practices for overall code improvement.

Hard-coded values have been detected in the `FToken` contract, which reduces the flexibility of the contract. Specifically, hard-coded values have been found in:

- `Symbol`: The token symbol is hard-coded as "Test Token".
- `Factor`: The decimal factor is hard-coded as 8.

These values must be configurable or defined in a way that allows them to be modified without the need to change the source code and redeploy the contract.

Source Code References

- [src/Flamingo.FToken/FToken.cs#L31](#)
- [src/Flamingo.FToken/FToken.cs#L42](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/58>

Missing Manifest Information

Identifier	Category	Risk	State
FOB2-31	Codebase Quality	Informative	Open

The Red4Sec team has detected that the **FLUND** contract does not properly specify the information related to the Smart Contract in its manifest.

All the contract information identified in the contract's manifest (author, email, description, source) belongs to the developer of the project. Therefore, it is recommended to customize said content by adding your own information, which will also provide the users with relevant information about the project.

```
namespace Flamingo.FLUND
{
    [ManifestExtra("Author", "")]
    [ManifestExtra("Email", "")]
    [ManifestExtra("Description", "")]
    [SupportedStandards("NEP-17")]
    [ContractPermission("?", "?")]
}
```

In the case of the audited smart contracts, it has been identified that the source attribute is not being included. This value can be defined using the `ContractSourceCodeAttribute` attribute. Additionally, the author, email and description values are not properly specified.

Recommendations

- Adding all possible metadata to the contracts favors indexing, traceability, and the audit by users, which conveys greater confidence to the user.

References

- <https://github.com/neo-project/proposals/blob/master/nep-16.mediawiki#source>

Source Code References

- <src/Flamingo.FLUND/FLUND.cs#L13-L17>

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/59>

Use of Assert instead of Throw

Identifier	Category	Risk	State
FOB2-32	Design Weaknesses	Informative	Open

The native `ExecutionEngine.Assert` must be used instead of the `throw`. The `ASSERT` and `ABORT` opcodes are approximately 9 times less expensive than the `THROW` opcode, and in the case of the `ExecutionEngine.Assert` it also allows to specify an error message.

`Throw` may be captured by the `try/catch` instructions. For this reason, its use must be limited only to some initial verification in the methods when none of the values have been updated and even in these cases it is more recommended to use `ExecutionEngine.Assert`.

In fact, `throw` it is only recommended to be used within a `try` sentence in the same contract, otherwise it can end up being captured by an external contract, and it will not necessarily end the execution of the transaction.

Recommendations

- `throw` it is only recommended to be used within a `try` sentence in the same contract.

Source Code References

- [src/Flamingo.FLUND/FLUND.cs#L675](#)
- [src/Flamingo.FLUND/FLUND.cs#L681](#)

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/60>

Code Typos

Identifier	Category	Risk	State
FOB2-33	Codebase Quality	Informative	Open

A simple typo in a line of code can cause minor confusion when reading, but it does not affect execution or system security. These mistakes can be made by both users and software developers and can have significant consequences in terms of data integrity, confidentiality, and availability.

A simple typo has been found in the following:

- In the `Burn` method '*witeness*' is written instead of *witness*.

Recommendations

- It is recommended to correct the typos in the source code.

Source Code References

- <src/Flamingo.FLUND/FLUND.cs#L180>

Fixes Review

This issue is currently being addressed by the Flamingo team and is part of an ongoing work-in-progress. For further details, please refer to the corresponding issue:

- <https://github.com/flamingo-finance/flamingo-sc-monorepo/issues/61>

Decimal Truncation in Division

Identifier	Category	Risk	State
FOB2-34	Business Logic	Low	Open

Since NeoVM does not support decimal values, performing division before multiplication can result in unintended truncation, leading to a loss of precision. This issue is particularly problematic in contexts where decimal values are critical, as it can produce incorrect results. Consider the following code snippet, where an arithmetic operation divides the value by 10 before multiplying by 25:

```
var balance0Adjusted = balance0 * 1000 - amount0In * 3;  
var balance1Adjusted = balance1 * 1000 - amount1In * 3;  
var balance0Adjusted = balance0 * 1000 - amount0In * (25 / 10);  
var balance1Adjusted = balance1 * 1000 - amount1In * (25 / 10);
```

The problem lies in the expression $(25 / 10)$. This division will produce a truncated result, removing any decimal parts. Therefore, $25 / 10$ evaluates as 2 instead of the expected value 2.5. This results in an incorrect calculation and will affect the accuracy of the final result.

Recommendations

- To ensure accuracy, the multiplication must be performed before the division, as demonstrated below:

```
var balance0Adjusted = ((balance0 * 1000 - amount0In) * 25) / 10;  
var balance1Adjusted = ((balance1 * 1000 - amount1In) * 25) / 10;
```

Source Code References

- [src/Flamingo.SwapPair/FlamingoSwapPairContract.cs#L199-L200](#)

Code Improvements

Identifier	Category	Risk	State
FOB2-35	Codebase Quality	Informative	Open

During the smart contract audit, certain bad practices have been detected throughout the code that should be improved. It's strongly advised to consider the adoption of enhanced coding styles and best practices for overall code improvement.

Below, we outline specific areas for potential style, quality, and code readability improvements in the audited contracts.

Unnecessary Casting

A recurring pattern has been identified in the code where a value is unnecessarily converted from `BigInteger` to `int` and then back to `BigInteger`. This multiple casting not only serves no purpose but can also introduce potential precision and functionality issues.

For example, consider the following method, though it is not the only instance of this pattern:

```
internal static BigInteger Get(BigInteger pairId)
{
    StorageMap pairIdToGasToBurnMap = new StorageMap(Storage.CurrentReadOnlyContext, gasToBurnPrefix);
    var value = pairIdToGasToBurnMap[(ByteString) pairId];
    return value != null ? (int)(BigInteger) value : 0;
}
```

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L717](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L735](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L753](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L771](#)
- [src/Flamingo.Broker/FlamingoBroker.Storage.cs#L828](#)
- [Flamingo.Broker/FlamingoBroker.Storage.cs#L846](#)

Use of BigInteger

The `gasAmount` parameter of the `SetGasToBurn` method is declared as an `int` type. Gas may require values that exceed the bounds of the `int` type, which could result in overflow errors or unnecessary restrictions.

The most suitable data type for `gasAmount` would be `BigInteger`, as it can represent integer values of any size, ensuring that the method is able to handle all possible gas values without restrictions.

```
public static void SetGasToBurn(BigInteger pairId, int gasAmount)
{
    ExecutionEngine.Assert(HasOwnerWitness(), "No owner witness");
    ExecutionEngine.Assert(IsPairExisting(pairId), "Pair does not exists");
    ExecutionEngine.Assert(gasAmount >= 0, "Invalid gas amount");
}
```

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.Owner.cs#L271](#)

Open ToDo

For example, a To-Do comment has been detected that reflect that the code is unfinished. Thus, many unaudited changes could introduce new vulnerabilities in the future.

```
// Update totals
// TODO: This might become a negative number, so let's add a check!
baseAmountLeft -= baseAmountConsumedAtUpperNode;
ExecutionEngine.Assert(baseAmountLeft >= 0, "Invalid base amount left [upper node]");
```

Source Code References

- [src/Flamingo.Broker/FlamingoBroker.Sell.cs#L376](#)

Annexes

Methodology

A code audit is a thorough examination of the source code of a project with the objective of identifying errors, discovering security breaches, or contraventions of programming standards. It is an essential component to the defense in programming, which seeks to minimize errors prior to the deployment of the product.

Red4Sec adopts a set of cybersecurity tools and best security practices to audit the source code of the smart contract by conducting a search for vulnerabilities and flaws.

The audit team performs an analysis on the functionality of the code, a manual audit, and automated verifications, considering the following crucial features of the code:

- The implementation conforms to protocol standards and adheres to best coding practices.
- The code is secure against common and uncommon vectors of attack.
- The logic of the contract complies with the specifications and intentions of the client.
- The business logic and the interactions with similar industry protocols do not contain errors or lead to dangerous situations to the integrity of the system.

In order to standardize the evaluation, the audit is executed by industry experts, in accordance with the following procedures:

Manual Analysis

- Manual review of the code, line-by-line, to discover errors or unexpected conditions.
- Assess the overall structure, complexity, and quality of the project.
- Search for issues based on the SWC Registry and known attacks.
- Review known vulnerabilities in the third-party libraries used.
- Analysis of the business logic and algorithms of the protocol to identify potential risk exposures.
- Manual testing to verify the operation, optimization, and stability of the code.

Automatic Analysis

- Scan the source code with static and dynamic security tools to search for known vulnerabilities.
- Manual verification of all the issues found by the tools and analyzes their impact.
- Perform unit tests and verify the coverage.

Vulnerabilities Severity

Red4Sec determines the severity of vulnerabilities found in risk levels according to the impact level defined by CVSSv3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST), classifying the risk of vulnerabilities on the following scale:

Severity	Description
Critical	Vulnerabilities that possess the highest impact over the systems, services, and/or sensitive information. The existence of these vulnerabilities is dangerous and should be fixed as soon as possible.
High	Vulnerabilities that could severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.
Medium	Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.
Low	These vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.
Informative	It covers various characteristics, information or behaviors that can be considered as inappropriate, without being considered as vulnerabilities by themselves.



Invest in Security, invest in your future