RED4SEC

Security Audit Report

07/04/2025

Flamingo lp-staking

RED4SEC

Content

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 makes it easy to buy/sell crypto, invest and earn revenue directly on the blockchain.

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 FUSD lp-staking** project.

This report presents detailed findings from the security audit of the Flamingo smart contracts. The analysis concluded that no critical vulnerabilities were identified.

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 lp-staking** 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 security assessment encompasses the following codebases and commits, as provided by **Neo**:

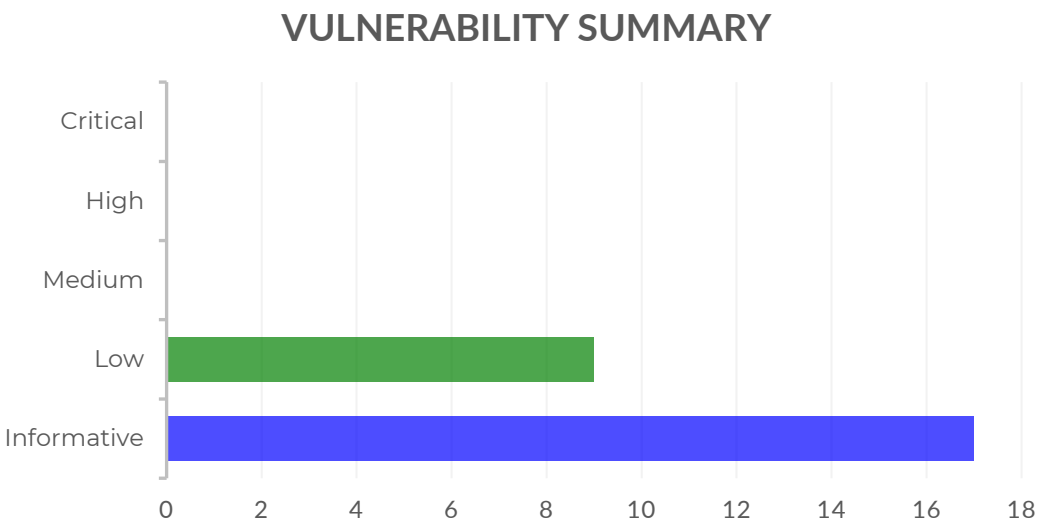
- https://github.com/flamingo-finance/flamingo-fusd/commits/feat/LPToken
    - `Commit: 99288ee0e624ff2002c7c9e8e87d82e8504ac3d8`
- https://github.com/flamingo-finance/flamingo-sc-monorepo/tree/features/lpAsCollateral/src/Flamingo.Staking
    - `Commit: e2a7a680c9039f0f4dee5f11605dc573db613ca3`
- https://github.com/flamingo-finance/FUSD-LP-Staking/tree/features/lpAsCollateral/fusd-lp-staking
    - `Commit: b65b65fa30df10f06591a1f26cb9fb732e205de2`

Executive Summary

The security audit against **Flamingo lp-staking** has been conducted between the following dates: **06/12/2025** and **07/04/2025**.

Once the analysis of the technical aspects has been completed, the performed analysis shows Upon completion of the technical assessment, the audit identified several non-critical vulnerabilities within the audited source code. While these findings do not pose an immediate threat to the integrity of the system, it is strongly recommended that they be addressed in a timely manner to reduce potential attack surfaces and ensure best security practices are maintained.

Throughout the audit process, **26 distinct vulnerabilities were identified**. Each has been classified based on the predefined risk levels outlined in the Vulnerabilities Severity Annex, providing a structured assessment of their potential impact



Conclusions

The Flamingo Finance lp-staking system, composed of `FTokenVault`, `FlamingoPriceFeed`, `Flamingo.Staking`, and `FUSD-LP-Staking`, appears to function appropriately as per the project's objectives. However, the audit has revealed several areas of concern that need to be addressed to ensure the system's security and reliability.

While no severe vulnerabilities were found, a significant number of issues were detected, all of which are of low or no impact. These issues range from a lack of input validation, unsafe numeric type usage, inconsistent reentrancy protection, to missing update data handling and more. Although these risks are considered to have a low impact, they still indicate areas that can be improved to enhance the system's overall security.

In terms of code quality and organization, the overall impression could be improved. The developed code does not conform to standard coding practices and lacks critical security measures, as evident from the numerous issues found. Most of the issues found relate to codebase quality, data validation, and design weaknesses, pointing to a lack of robustness in the software. These issues include inconsistent storage prefix types, unsafe key concatenation, incorrect boolean check, unsafe numeric type usage, and a lack of input validation.

Additionally, the system exhibits several inconsistencies and typos within the codebase. It also incorrectly applies GAS optimizations, uses insecure FLM address authority, and fails in event logging with missing event emissions.

The project also lacks comprehensive documentation, making it challenging to verify if the current implementation aligns with the project's objectives and requirements. Furthermore, the system is not designed to function with non-standard tokens, such as tokens with fees or deflationary tokens.

One significant concern is that the Flamingo team maintains control over specific centralized components, which allows them to alter certain contract values at will. This capacity could potentially lead to an advantageous position in certain situations, which could pose a risk to the system's users.

Given these findings, it is recommended that the Flamingo team take immediate action to rectify these issues. The code should be reviewed and updated to adhere to standard coding practices, and security measures should be implemented. Comprehensive documentation should also be created to facilitate system understanding and verification. Finally, the team should consider decentralizing control over the system to enhance its security and reliability.

In conclusion, while the Flamingo FUSD lp-staking system is functioning as intended, there are several areas of concern that need to be addressed to ensure its longevity and security. These issues, although small, can accumulate and potentially cause significant problems in the future if not rectified promptly.

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
FUSD03-01	Incorrect Percentage Sum Validation	**Low**	**Open**
FUSD03-02	Non-Standard Mint Event Ordering	**Low**	**Open**
FUSD03-03	Insecure FLM Address Authority	**Low**	**Open**
FUSD03-04	Unsafe Numeric Type Usage	**Low**	**Open**
FUSD03-05	Wrong ByteString Conversion	**Low**	**Open**
FUSD03-06	Incorrect Boolean Check in isLPToken	**Low**	**Open**
FUSD03-07	Inconsistent Reentrancy Protection	**Low**	**Open**
FUSD03-08	Unsafe Key Concatenation	**Low**	**Open**
FUSD03-09	Missing Event Emissions	**Low**	**Open**
FUSD03-10	Lack of Inputs Validation	**Informative**	**Open**
FUSD03-11	Limit Call Rights	**Informative**	**Open**
FUSD03-12	Reserved Storage Prefix Usage	**Informative**	**Open**
FUSD03-13	Inconsistent Storage Prefix Types	**Informative**	**Open**
FUSD03-14	Outdated Frameworks	**Informative**	**Open**
FUSD03-15	Missing Safe Attribute	**Informative**	**Open**
FUSD03-16	Safe Storage Access	**Informative**	**Open**
FUSD03-17	Missing Update Data Handling	**Informative**	**Open**
FUSD03-18	Deprecated Method Usage	**Informative**	**Open**
FUSD03-19	Incomplete Contract Manifest Metadata	**Informative**	**Open**
FUSD03-20	Bad Coding Practices	**Informative**	**Open**
FUSD03-21	Safe Contract Update	**Informative**	**Open**
FUSD03-22	Codebase Inconsistencies and Typos	**Informative**	**Open**
FUSD03-23	Unsecured Ownership Transfer	**Informative**	**Open**
FUSD03-24	Lack of Documentation	**Informative**	**Open**

FUSD03-25	Incorrect Timestamp Documentation	**Informative**	**Open**
FUSD03-26	GAS Optimizations	**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

Incorrect Percentage Sum Validation

Identifier	Category	Risk	State
FUSD03-01	Business Logic	**Low**	**Open**

An incorrect validation has been identified in the `BatchSetPercentageByAsset` and `SetPercentageByAsset` functions, which assign asset-specific percentages for a user. Both functions use the condition `newSum <= 100` to ensure that the total percentage does not exceed 100%. However, this check permits inconsistent configurations where the sum is less than 100%, potentially leading to logical errors, unexpected behavior, or incomplete fund distribution during subsequent processes such as `BatchClaim`.

The absence of a strict equality check (`== 100`) allows users to define partial distribution configurations that do not represent a full 100% allocation, which may result in invalid states or fragmented logic in components expecting an exact distribution. In automated financial systems, percentage precision is critical to ensure traceability, fairness, and operational transparency.

Recommendations

- Modify the validations in both methods to require `newSum` to be exactly equal to 100, not merely less than or equal.
- Implement a preliminary check to reject empty arrays or arrays containing negative values, preventing nonsensical divisions or inconsistencies.
- Emit a configuration event when new percentages are set, including the user, the associated assets, and the assigned values, to enable off-chain traceability.

Source Code References

- fusd-lp-staking/Staking.Percent.cs#L26
- fusd-lp-staking/Staking.Percent.cs#L34

Non-Standard Mint Event Ordering

Identifier	Category	Risk	State
FUSD03-02	Auditing and Logging	**Low**	**Open**

It was observed that the `mint` method in the `FToken` contract emits the `Transfer` event (with `from = null`) prior to the `Mint` event. This sequence diverges from commonly accepted patterns in the implementation of standard-compliant tokens, such as those adhering to NEP-17, where the `Mint` event—if emitted—typically precedes or replaces the `Transfer` event.

Emitting the `Transfer` event before the `Mint` event may lead to inconsistencies in off-chain indexing systems, block explorers, and wallets that assume event semantics aligned with ecosystem standards. This can result in balance tracking errors, duplicate transaction records, or failures in correctly recognizing token creation. Moreover, the emission of both events may be considered redundant, as a `Transfer` event from `null` already conventionally signifies a minting operation.

Recommendations

- Reverse the order of event emission so that the `onMint` event is triggered before the `onTransfer` event, aligning with standard practices.
- Evaluate whether the `onMint` event is strictly necessary, given that a `Transfer` from `null` is the widely accepted pattern for representing token minting.

Source Code References

- ftoken/FToken.java#L197-L200

Insecure FLM Address Authority

Identifier	Category	Risk	State
FUSD03-03	Access Controls	**Low**	**Open**

A risk has been identified regarding the management of the FLM token address within the audited contract. Currently, the `SetFLMAddress` method allows modification of the FLM address by any entity with the `author` role, without enforcing exclusivity to the contract `owner` or imposing restrictions on the frequency of updates. This lack of strict control over a critical variable such as the central token address introduces a significant manipulation risk.

Allowing any `author` to dynamically alter the FLM address means that, in the event of role compromise or abuse, asset flows could be redirected or contract behavior altered, severely impacting protocol integrity. The absence of a one-time assignment constraint further exacerbates the risk by enabling arbitrary modifications throughout the contract's lifecycle. Given that the FLM address serves as a reference for value and incentive mechanisms, unauthorized changes could result in serious financial consequences and undermine trust in the contract.

Recommendations

- Restrict invocation of `SetFLMAddress` exclusively to the contract owner, rather than allowing users with the generic `author` role.
- Implement protective logic to allow the FLM address to be set only once, preventing any subsequent unauthorized modifications.

Source Code References

- fusd-lp-staking/Staking.FLM.cs#L24

Unsafe Numeric Type Usage

Identifier	Category	Risk	State
FUSD03-04	Undefined Behavior	**Low**	**Open**

In the Java implementation of several smart contracts, native numeric types such as `int`, `uint`, and `long` are used for arithmetic operations. This practice is discouraged in the context of NEO blockchain development using the `neow3j` DevPack, as these native types do not accurately represent the execution behavior of the NEO Virtual Machine (NeoVM).

At compile time, literals and arithmetic expressions are translated into instructions operating on `BigInteger` in the bytecode generated by `neow3j`. However, during Java development, these expressions may be executed using primitive types like `int`, introducing risks of arithmetic *overflow* or *underflow* that do not occur in the NeoVM. This mismatch creates a dangerous discrepancy between development-time behavior and on-chain execution.

Expressions such as `(FLOAT_MULTIPLIER * diffTimestamp * annualInterest)` may yield invalid or truncated results in Java, leading to confusion and subtle logic errors that are difficult to detect.

Recommendations

- Replace all primitive numeric types (`int`, `long`, etc.) with `BigInteger` in Java smart contracts to ensure semantic consistency with NeoVM execution.
- Systematically validate literal conversions and assignments to ensure that represented values are accurately preserved on-chain.

Source Code References

- ftoken/FTokenVault.java
- ftoken/FlamingoPriceFeed.java
- ftoken/FToken.java
- ftoken/interfaces/FlamingoSwapPairContract.java#L18-L19
- ftoken/interfaces/FTokenContract.java#L15-L17

Wrong ByteString Conversion

Identifier	Category	Risk	State
FUSD03-05	Business Logic	**Low**	**Open**

When using `ToByteString` on an address, its binary content is converted into a `ByteString`, but this may contain non-printable data, which could cause errors when logging it due to invalid UTF-8 sequences. The correct approach is to use the `ToAddress` method of the `UInt256` object and `itoa` to convert numbers.

```
/// <summary>
/// The implementation of System.Runtime.Log.
/// Writes a log.
/// </summary>
/// <param name="state">The message of the log.</param>
1 reference
protected internal void RuntimeLog(byte[] state)
{
    if (state.Length > MaxNotificationSize) throw new ArgumentException("Message is too long.", nameof(state));
    try
    {
        string message = state.ToStrictUtf8String();
        Log?.Invoke(this, new LogEventArgs(ScriptContainer, CurrentScriptHash, message));
    }
    catch
    {
        throw new ArgumentException("Failed to convert byte array to string: Invalid or non-printable UTF-8 sequence detected.", nameof(state));
    }
}
```

This can cause an exception when using `StringUTF8`, producing a different error than expected. The correct way to convert numbers is to use the `ToAddress` method of the `UInt256` object and `itoa`.

Recommendations

- Use the existing `ToAddress` method on the `UInt160` type.

References

- https://github.com/neo-project/neo-devpack-dotnet/blob/6375fe668573c6e8ba0cf38fbfeaa69b245b862c/src/Neo.SmartContract.Framework/UInt160.cs#L63

Source Code References

- FLM/FLM.Asset.cs#L23
- FLM/FLM.Asset.cs#L29
- FLM/FLM.Asset.cs#L36-L42
- FLM/FLM.Owner.cs#L31-L32
- FLM/FLM.Owner.cs#L58-L81
- fusd-lp-staking/Staking.Pause.cs#L30-L71
- fusd-lp-staking/Staking.WhiteList.cs#L34-L42
- fusd-lp-staking/Staking.Owner.cs#L25-L63
- fusd-lp-staking/Staking.Owner.cs#L126

Incorrect Boolean Check in isLPToken

Identifier	Category	Risk	State
FUSD03-06	Data Validation	**Low**	**Open**

An incorrect logic has been identified in the `isLPToken()` function, which uses the `getBoolean()` method to verify whether a given token is classified as an LP token in storage. This approach is flawed, as the stored value associated with LP tokens is of type `Hash160`, while `getBoolean()` only returns `false` if the value is exactly `0x00` (e.g., a null or improperly initialized `Hash160`). Any other value, regardless of type or validity, will be interpreted as `true`, even if it does not represent a valid or expected value.

This faulty validation may compromise critical functions such as `getTrueOnChainPrice()` and others related to staking or collateralization, where accurate identification of LP tokens is essential.

Recommendations

- To avoid silent errors, replace `getBoolean()` with a more reliable check such as `get() != null`, ensuring the presence of a valid stored value rather than relying on type-based coercion.

Source Code References

- ftoken/FTokenVault.java#L1926
- ftoken/FTokenVault.java#L1898

Inconsistent Reentrancy Protection

Identifier	Category	Risk	State
FUSD03-07	Timing and State	**Low**	**Open**

An inconsistent use of reentrancy protection mechanisms has been identified across the audited contracts. While some contracts implement a manual approach—writing an execution marker to storage (`EnteredStorage.Set(tx.Hash)`)—others correctly utilize native DevPack attributes such as `NoReentrant`. The manual implementation presents several issues: if execution terminates unexpectedly without clearing the marker (`EnteredStorage.Delete`), it may leave residual data that permanently blocks functionality or produces false positives. Furthermore, manual protection is prone to logic errors and is harder to maintain.

The NEO DevPack, as of version 3.7.4, provides official mechanisms such as `NoReentrant` (contract-wide protection) and `NoReentrantMethod` (method-level protection), which are safer, more efficient, and easier to maintain. The coexistence of multiple approaches to such a critical safeguard introduces unnecessary complexity.

Although no current exploit has been identified in the `FToken` **contract (Java), it lacks any reentrancy protection** on critical functions.

Recommendations

- Standardize the use of reentrancy protection across the project by adopting the official `NoReentrant` and `NoReentrantMethod` attributes.
- Remove manual storage-based implementations (`EnteredStorage`) to prevent residual data and reduce code complexity.
- Incorporate reentrancy safeguards in `FToken` as well, using secure and consistent protection patterns.

References

- https://github.com/neo-project/neo-devpack-dotnet/blob/master/docs/framework/ReentrantAttack.md

Source Code References

- fusd-lp-staking/Staking.cs#L227
- fusd-lp-staking/Staking.cs#L162
- Flamingo.Staking/FlamingoStaking.cs#L208
- Flamingo.Staking/FlamingoStaking.cs#L142
- ftoken/FToken.java

Unsafe Key Concatenation

Identifier	Category	Risk	State
FUSD03-08	Data Validation	**Low**	**Open**

The use of direct byte array concatenation for generating storage keys has been identified, without the inclusion of delimiters or a robust serialization scheme. This technique is employed, for instance, through expressions such as `(byte[])asset.Concat(timestamp.ToByteArray())`. The absence of separators or length prefixes in key construction introduces a latent risk of collisions when concatenated elements have variable lengths.

In smart contract contexts, where storage keys must be unique to ensure correct data identification and retrieval, this approach may lead to overwrites or incorrect accesses if different inputs produce identical concatenation results. This issue is particularly critical when using types such as `timestamp.ToByteArray()`, whose binary representation may vary in length depending on the specific value. Exploiting this weakness requires no special privileges and may result in logic errors, data loss, or contract integrity vulnerabilities.

Recommendations

- Implement an explicit serialization scheme using methods such as `StdLib.Serialize()` and `StdLib.Deserialize()` to construct and parse storage keys securely.
- Include delimiters or length prefixes between each key component when concatenating byte arrays.
- Use fixed-length formats to represent key components (e.g., convert timestamps to a standard 8-byte format).

Source Code References

- Flamingo.Staking/FlamingoStaking.Storage.cs#L237-L244

Missing Event Emissions

Identifier	Category	Risk	State
FUSD03-09	Auditing and Logging	**Low**	**Open**

The absence of events in several critical administrative functions that modify essential smart contract parameters has been identified. Methods such as `setAnnualInterest`, `setMaxLoanToValue`, `SetLendContract`, `setVaultScriptHash`, `setSwapFactoryHash`, `setFLUNDHash`, `SetFlocks`, `SetLPConfig`, and functions related to pausing and resuming operations (`Pause`, `UnPause`, `PauseStaking`, `UnPauseStaking`, `PauseRefund`, `UnPauseRefund`) perform significant state changes without emitting notification events.

Omitting events in such operations severely compromises the contract's traceability and transparency. On blockchain platforms like NEO, events are essential for off-chain monitoring, change auditing, and integration with external systems. Without event emissions, it becomes unreliable to track when, how, or by whom specific parameters were modified, undermining administrative control and hindering the detection of misconfigurations or malicious activity.

Recommendations

- Emit custom events for all functions that modify critical system parameters, including the name of the modified parameter, its previous and new values, and the transaction invoker if relevant.
- Include events for all pause and resume operations, specifying the affected component and the resulting state.
- Establish a consistent naming convention for administrative events to ensure they are easily indexable and uniform across external monitoring tools.

Source Code References

- ftoken/FTokenVault.java#L1398
- ftoken/FTokenVault.java#L1232
- fusd-lp-staking/Staking.Owner.cs#L96
- ftoken/FToken.java#L123
- ftoken/FlamingoPriceFeed.java#L104
- ftoken/FlamingoPriceFeed.java#L116
- ftoken/FTokenVault.java
- Flamingo.Staking/FlamingoStaking.Pause.cs
- Flamingo.Staking/FlamingoStaking.Owner.cs

Lack of Inputs Validation

Identifier	Category	Risk	State
FUSD03-10	Data Validation	**Informative**	**Open**

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

In certain methods it is convenient to check that the value is not `IsZero`, leaving the verification as: `hash.IsValid && !hash.IsZero`.

- fusd-lp-staking/Staking.FLM.cs#L20
- fusd-lp-staking/Staking.FLM.cs#L28
- fusd-lp-staking/Staking.Record.cs#L16
- fusd-lp-staking/Staking.cs#L153
- fusd-lp-staking/Staking.cs#L159
- fusd-lp-staking/Staking.cs#L222
- Flamingo.Staking/FlamingoStaking.cs#L121
- Flamingo.Staking/FlamingoStaking.cs#L127
- Flamingo.Staking/FlamingoStaking.cs#L203
- Flamingo.Staking/FlamingoStaking.FLM.cs#L26

Additionally, a lack of whitelist verification has been found in the following references:

- fusd-lp-staking/Staking.cs#L163
- fusd-lp-staking/Staking.cs#L222

No upper bound validation for `liquidationBonus` percentage:

- ftoken/FTokenVault.java#L1320

Lack of numeric input validation:

- Flamingo.Staking/FlamingoStaking.Owner.cs#L105

Check that `data` is not `null`:

- ftoken/FTokenVault.java#L326
- Flamingo.Staking/FlamingoStaking.cs#L36

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.

Limit Call Rights

Identifier	Category	Risk	State
FUSD03-11	Design Weaknesses	**Informative**	**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. So, it is advisable to use the principle of least privilege (PoLP) during all the external processes or the calls to the contracts.

Therefore, 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`.

Recommendations

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

References

- https://en.wikipedia.org/wiki/Principle_of_least_privilege

Source Code References

- fusd-lp-staking/Staking.FUSD.cs#L24

Reserved Storage Prefix Usage

Identifier	Category	Risk	State
FUSD03-12	Codebase Quality	**Informative**	**Open**

Some audited contracts have been found to use storage prefixes within the reserved range `0x00` to `0x0F`. These values are designated exclusively for system contracts and development framework components in the NEO ecosystem, such as `TokenContract`, as defined in `Neo.SmartContract.Framework.TokenContract`. Their use by other contracts constitutes a violation of established ecosystem conventions.

```
// Keys
private static final byte[] OWNER_KEY = new byte[] { 0x00 };
private static final byte[] SUPPLY_KEY = new byte[] { 0x01 };
private static final byte[] VAULT_HASH_KEY = new byte[] { 0x02 };
```

Use of reserved prefixes

Prefix collisions may lead to the overwriting of critical data or the reading of incompatible information from another contract or system module, compromising state integrity. These errors are often difficult to diagnose, as the resulting faulty behavior may not manifest immediately. Moreover, using reserved prefixes restricts future compatibility with newer framework versions, increasing the contract's technical risk and complicating long-term maintenance.

Recommendations

- Avoid using storage prefixes within the `0x00` to `0x0F` range, reserving this space exclusively for framework use.
- Assign custom prefixes outside the reserved range, preferably starting from `0x10`, ensuring uniqueness and traceability by module or functionality.
- Clearly document all prefixes used within the contract to support future audits, integrations, and maintenance.

References

- https://github.com/neo-project/neo-devpack-dotnet/blob/master/src/Neo.SmartContract.Framework/TokenContract.cs

Source Code References

- ftoken/FToken.java#L50-L52

Inconsistent Storage Prefix Types

Identifier	Category	Risk	State
FUSD03-13	Codebase Quality	**Informative**	**Open**

Different `storage` prefixes are used throughout the contract, mixing `String` prefixes and `Byte` prefixes. It is convenient to use the practice of unifying all the prefixes to the same type and of the same length to avoid possible collisions and injections in the use of storage keys.

The use of prefixes in string mode is discouraged, and even more so if they are text strings of different sizes amongst them, since this will cause the keys to have a different sizes, If the code is vulnerable, there is a possibility of collisions and/or storage injections, however not in this particular case.

Recommendations

- Standardize the use of fixed-length byte-formatted storage prefixes to ensure consistency and prevent variable key structures.
- Avoid using string-formatted prefixes, particularly if they lack normalized length.
- Assign prefixes in a logical order and document them clearly, specifying their purpose and usage within the contract to facilitate understanding and future audits.

Source Code References

- fusd-lp-staking/Staking.Storage.cs#L23
- Flamingo.Staking/FlamingoStaking.Storage.cs#L12

Outdated Frameworks

Identifier	Category	Risk	State
FUSD03-14	Outdated Software	**Informative**	**Open**

The project's smart contracts have been found to rely on outdated versions of the official NEO frameworks, both in the C# and Java implementations. Specifically, the C# code is based on NEO framework version 3.1.0, while the Java implementation uses the `io.neow3j.gradle-plugin` version 3.19.0. More recent versions are available—3.8.1 for NEO C# and 3.24.0 for `neow3j`—which include significant improvements in performance, compatibility, new features, and critical bug and vulnerability fixes present in earlier releases.

```
<ItemGroup>
  <PackageReference Include="Neo.SmartContract.Framework" Version="3.1.0" />
</ItemGroup>
```

The use of outdated compilers and development tools poses a significant risk for deployed smart contracts on blockchain platforms, as it limits the ability to leverage important optimizations and prevents the application of critical security patches.

Recommendations

- Update the NEO C# framework from version 3.1.0 to the latest stable release (3.8.1 at the time of analysis) to benefit from security enhancements and functional improvements.
- Upgrade the `io.neow3j.gradle-plugin` in Java projects from version 3.19.0 to 3.24.0 to ensure compatibility with the latest tools in the NEO ecosystem.

References

- https://www.nuget.org/packages/Neo.SmartContract.Framework
- https://plugins.gradle.org/plugin/io.neow3j.gradle-plugin

Source Code References

- FLM/FLM.csproj#L8
- fusd-lp-staking/fusd-lp-staking.csproj#L14
- build.gradle#L3-L4
- Flamingo.Staking/Flamingo.Staking.csproj#L10

Missing Safe Attribute

Identifier	Category	Risk	State
FUSD03-15	Configuration	**Informative**	**Open**

Several read-only methods in the audited contracts have been identified as missing the `Safe` attribute. 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. Thus, it is convenient to establish our query methods as *Safe* to keep the principle of least privilege.

Although it is worth noting that in the case of the `GetHistoryUintSProfitSum` method, it seems it should be a private method. It would be advisable to review the intended visibility of this method or add the `Safe` attribute.

Recommendations

- It is convenient to add the `Safe` attribute to methods that do not make any changes to the storage.
- Review the visibility of the `GetHistoryUintSProfitSum` method to ensure it aligns with its intended usage.

References

- https://en.wikipedia.org/wiki/Principle_of_least_privilege

Source Code References

- fusd-lp-staking/Staking.Record.cs#L16
- ftoken/FToken.java#L102
- ftoken/FlamingoPriceFeed.java#L82
- ftoken/FTokenVault.java#L1881

Safe Storage Access

Identifier	Category	Risk	State
FUSD03-16	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` in all read-only methods that do not modify state, reinforcing the principle of least privilege.

References

- https://en.wikipedia.org/wiki/Principle_of_least_privilege

Source Code References

- fusd-lp-staking/Staking.Storage.cs#L29
- ftoken/FTokenVault.java#L2119-L2151
- Flamingo.Staking/FlamingoStaking.Storage.cs#L18

Missing Update Data Handling

Identifier	Category	Risk	State
FUSD03-17	Design Weaknesses	**Informative**	**Open**

The audited contracts do not implement handling of the `data` parameter in the `update` method, which is part of the smart contract upgrade standard in NEO N3. Although this parameter is not currently used explicitly, its purpose is to enable execution of specific actions during the upgrade process by passing it to the `_deploy` method.

Omitting this mechanism significantly restricts the contract's future upgradeability. Without support for `data`, activating new functionality during an update would require changes to both the contract code and its `manifest.json`. Additionally, decentralized applications (dApps) interacting with these contracts would need to be adapted, incurring extra maintenance and integration costs, and potentially disrupting operational continuity.

Recommendations

- Include the `data` parameter in the `update` method definition across contracts, even if not used immediately, to ensure forward compatibility.
- Prepare the `_deploy` function to interpret optional initialization data when received during a contract update.

Source Code References

- ftoken/FToken.java#L91
- ftoken/FlamingoPriceFeed.java#L74
- ftoken/FTokenVault.java#L275
- Flamingo.Staking/FlamingoStaking.Upgrade.cs#L11

Deprecated Method Usage

Identifier	Category	Risk	State
FUSD03-18	Codebase Quality	**Informative**	**Open**

The `liquidateCollateralOCP` method has been found to use the `getOnChainPrice` function, which has been marked for deprecation, instead of the updated `getPrice` method. Although both methods provide equivalent functionality, relying on deprecated functions introduces maintenance risks and may cause confusion during future contract updates or changes in the execution environment.

```
// Unfortunately we can't rename this due to external clients
// We will migrate them to #getPrice and deprecate this method
@Safe
public static int getOnChainPrice(Hash160 tokenHash, int decimals) throws Exception {
    // FUSD price is always 1 for the purposes of computing LTV
    if (isFTokenSupported(tokenHash)) {
        return Helper.pow(10, decimals);
    }
    return getTrueOnChainPrice(tokenHash, decimals);
}
```

Deprecated method

This behavior is also observed in `FlamingoStaking`, where `Runtime.ScriptContainer` is used instead of the recommended `System.Runtime.Transaction`.

Recommendations

- Replace the use of `getOnChainPrice` with `getPrice` in the `liquidateCollateralOCP` method to align the code with the current oracle contract version.
- Review other references to the deprecated method throughout the codebase and replace them consistently to eliminate unwanted future dependencies.

Source Code References

- ftoken/FTokenVault.java#L2285
- ftoken/FTokenVault.java#L686
- ftoken/FTokenVault.java#L1771
- Flamingo.Staking/FlamingoStaking.cs#L205
- Flamingo.Staking/FlamingoStaking.cs#L253

Incomplete Contract Manifest Metadata

Identifier	Category	Risk	State
FUSD03-19	Codebase Quality	**Informative**	**Open**

Some smart contracts in the project have been found to lack proper specification of required information in their 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, increasing the trust and improve the SEO of the contract.

```
[ManifestExtra("Author", "")]
[ManifestExtra("Email", "")]
[ManifestExtra("Description", "")]
```

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

- FLM/FLM.cs#L10-L12
- Flamingo.Staking/FlamingoStaking.cs#L20-L22

Bad Coding Practices

Identifier	Category	Risk	State
FUSD03-20	Codebase Quality	**Informative**	**Open**

The audited smart contract code exhibits a significant lack of organization, structural clarity, and stylistic consistency. While this does not constitute a direct vulnerability, it represents a technical weakness that severely impairs code readability, comprehension, analysis, and long-term maintainability. In dynamic and continuously evolving projects, code quality is a critical factor for both security and sustainability.

Several suboptimal patterns and practices were identified during the review, including:

Assignment of initial values using `InitialValue` instead of safer and more explicit attributes such as `IntegerAttribute` or `Hash160Attribute`.

- fusd-lp-staking/Staking.Owner.cs#L15
- Neo.SmartContract.Framework/Attributes/IntegerAttribute
- Neo.SmartContract.Framework/Attributes/Hash160Attribute

Use of variable-length storage keys.

- fusd-lp-staking/Staking.Storage.cs#L262
- fusd-lp-staking/Staking.Storage.cs#L424-L431

Unordered use of storage prefixes.

- fusd-lp-staking/Staking.Storage.cs#L42

Use of `Array` instead of more robust structures like `List`, which enables clearer semantics such as returning `null` or using `Append`.

- FLM/FLM.Storage.cs#L118

Safe Contract Update

Identifier	Category	Risk	State
FUSD03-21	Design Weaknesses	**Informative**	**Open**

It is important to mention that the owner of the contract has the possibility of updating the contract, which implies a possible change in the logic and in the functionalities of the contract, subtracting part of the concept of decentralized trust.

Although this is a recommended practice in these early phases of the N3 blockchain where significant changes can still take place, it would be convenient to include some protections to increase transparency for the users so they can act accordingly.

Recommendations

There are a few good practices that help mitigate this problem, such as; add a `TimeLock` to start the `Update` operations, emit events when the `Update` operation is requested, temporarily disable contract functionalities and finally issue a last notification and execute the `Update` operation after the `TimeLock` is completed.

Source Code References

- ftoken/FToken.java#L91
- ftoken/FlamingoPriceFeed.java#L74
- ftoken/FTokenVault.java#L275
- Flamingo.Staking/FlamingoStaking.Upgrade.cs#L11

Codebase Inconsistencies and Typos

Identifier	Category	Risk	State
FUSD03-22	Codebase Quality	**Informative**	**Open**

A significant lack of order, structure, and consistency has been identified in the audited smart contract codebase, severely hindering its readability, analysis, and maintainability. These deficiencies include typographical errors in key identifiers (e.g., `StakingReocrd` instead of `StakingRecord`) and repeated misuse of incorrect names (e.g., `benifactor` instead of `benefactor`), observed across multiple project files.

Furthermore, inconsistent naming conventions have been found in classes related to persistent storage, such as `FrozenBalanceStoragetorage` and `LPStakeFromLendContractStorageContractStorage`.

Misspelled Identifiers

- `StakingReocrd` struct name misspelled (should be `StakingRecord`).
- `benifactor` repeated in parameters/variables (should be `benefactor`).

This inconsistency appears in multiple files and could impact code readability and maintenance.

Source References

- Flamingo.Staking/FlamingoStaking.cs#L10
- Flamingo.Staking/FlamingoStaking.cs#L60
- Flamingo.Staking/FlamingoStaking.cs#L143

Inconsistent Storage Naming

Storage classes exhibit inconsistent naming (e.g., `FrozenBalanceStoragetorage`, `LPStakeFromLendContractStorageContractStorage`). This reduces code readability and maintainability.
Recommendations:

- Standardize storage class suffixes (e.g., *Storage).
- Remove redundant terms like 'Storage' in nested contexts.
- Align with Neo's official storage examples.

Source References

- Flamingo.Staking/FlamingoStaking.Storage.cs#L470
- Flamingo.Staking/FlamingoStaking.Storage.cs#L500

Unsecured Ownership Transfer

Identifier	Category	Risk	State
FUSD03-23	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.

Recommendations

- Implement a system for the governance transfer, in which the new account is assigned to accept the proposal, in order to validate that it is a completely valid account.

Source Code References

- fusd-lp-staking/Staking.Owner.cs#L23
- ftoken/FToken.java#L107
- ftoken/FlamingoPriceFeed.java#L88
- ftoken/FTokenVault.java#L853
- Flamingo.Staking/FlamingoStaking.Owner.cs

Lack of Documentation

Identifier	Category	Risk	State
FUSD03-24	Testing and Documentation	**Informative**	**Open**

The project does not contain technical documentation of any sort, nor execution flow diagrams, class diagrams or code properly commented. This is a bad practice that complicates understanding the project and makes it difficult for the team of auditors to analyze the functionalities, since there is no technical documentation to check if the current implementation meets the needs and purpose of the project.

Although the project has excellent test coverage, it is notable that it lacks documentation. Updated and accurate documentation is crucial for open-source projects, as it directly impacts community adoption and contributions. Providing comprehensive documentation enhances the project's accessibility and encourages broader involvement.

Having updated and accurate documentation of the project is an essential aspect for open source projects, in fact it is closely related with the adoption and contribution of the project by the community.

Documentation is an integral part of the Secure Software Development Life Cycle (SSDLC), and it helps to improve the quality of the project, so it is recommended to add the code documentation with the according descriptions of the functionalities, classes and public methods.

References

- https://snyk.io/learn/secure-sdlc
- https://www.freecodecamp.org/news/why-documentation-matters-and-why-you-should-include-it-in-your-code-41ef62dd5c2f

Source Code References

- fusd-lp-staking

Incorrect Timestamp Documentation

Identifier	Category	Risk	State
FUSD03-25	Testing and Documentation	**Informative**	**Open**

A discrepancy has been identified between the comments in the project's configuration files and the actual values used for time-related fields. Specifically, the comments state that `startStakingTimeStamp` and `startClaimTimeStamp` are expressed in milliseconds (ms), whereas the numerical values provided (e.g., `1601114400`) are in fact expressed in seconds since the UNIX epoch.

This inconsistency between documentation and actual behavior can mislead developers, auditors, or external integrators, resulting in incorrect interpretations of the system's temporal logic. For instance, assuming the values are in milliseconds may lead to miscalculations, failed time validations, or unexpected behavior in functions dependent on these parameters. While not a direct vulnerability, this lack of precision undermines the clarity, maintainability, and reliability of the codebase.

Recommendations

- Update the configuration file comments to accurately reflect that the timestamp values are expressed in **seconds**, not milliseconds.
- Ensure that all functions consuming these values are aligned with the seconds format, avoiding unnecessary conversions or misinterpretations.

Source Code References

- Flamingo.Staking/buildConfig/testnet.config.js#L6-L9
- Flamingo.Staking/buildConfig/unittests.config.js#L6-L9

GAS Optimizations

Identifier	Category	Risk	State
FUSD03-26	Codebase Quality	**Informative**	**Open**

During the audit, certain code segments were found to lack optimization principles, resulting in unnecessary resource consumption during smart contract execution on the NEO N3 blockchain. In NEO N3, GAS represents the unit of computational cost, and each executed instruction carries a tangible economic value. Any redundant operation, inefficient use of data structures, or poorly structured logic increases execution costs for users.

Logic optimization

Moving the maximum size check (`MAX_SIGNERS_SIZE`) to the `setSigner` method would save GAS on all future contract executions:

- ftoken/FTokenVault.java#L898C35-L900

Eliminating the `result` variable avoids unnecessary checks, as `result` can never be `false` at line 61:

- FLM/FLM.Asset.cs#L61

The use of `CallingScriptHash` is unnecessary, `CheckWitness` already performs the verification internally:

- FLM/FLM.Asset.cs#L30
- FLM/FLM.Owner.cs#L81

Unnecessary conversion of a constant string:

- fusd-lp-staking/Staking.cs#L215-L219

The `checkValid` call is redundant, as the asset is always valid and whitelisted:

- fusd-lp-staking/Staking.cs#L66

Single-line method with a single call:

- fusd-lp-staking/Staking.FUSD.cs#L44

Redundant check, if the asset is whitelisted, it cannot be zero:

- fusd-lp-staking/Staking.Percent.cs#L41

Including `FromAddress` and `AssetId` in the value is unnecessary when already present in the key:

- fusd-lp-staking/Staking.Storage.cs#L349-L383

Move the `symbol` call inside the `else` block to avoid redundant execution:

- ftoken/FTokenVault.java#L2254

Dead code

The `FIXED18` variable is unused:

- fusd-lp-staking/Staking.cs#L16

The `frozenBalance` variable is unused:

- fusd-lp-staking/Staking.cs#L196
- Flamingo.Staking/FlamingoStaking.cs#L176

The `toToken` argument is unused:

- ftoken/FlamingoPriceFeed.java#L189

The `UpgradeTimeLockStorage` class is unused:

- Flamingo.Staking/FlamingoStaking.Storage.cs#L300

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.

- FLM/FLM.Storage.cs#L10-L18
- ftoken/FToken.java#L36-L54
- ftoken/FlamingoPriceFeed.java#L34-L48
- ftoken/FTokenVault.java#L70-L238

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 compromise 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