RED4SEC

Security Audit Report

05/12/2022

Flamingo FUSD

RED4SEC

Content

Introduction

FUSD is the first *FToken* - a NEP-17 stablecoin that is pegged to a specific currency. As its name suggests, FUSD tracks the US dollar.

A FToken is an over-collateralized stablecoin with its value backed by various collateral tokens, including FLUND, bNEO, fWETH, and fWBTC. Anyone with a position in one of these collateral tokens can post the tokens as collateral and mint FTokens. The minted FToken will continuously accrue interest until it has been fully repaid. Where the solvency of the protocol is guaranteed by maximum loan-to-value (LTV) ratios and liquidation mechanisms.

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** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of Flamingo fUSD. The performed analysis shows that the smart contract did contain a critical vulnerability and other issues that were properly fixed by the **Flamingo** team.

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 neither "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 its 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 fUSD** 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**:

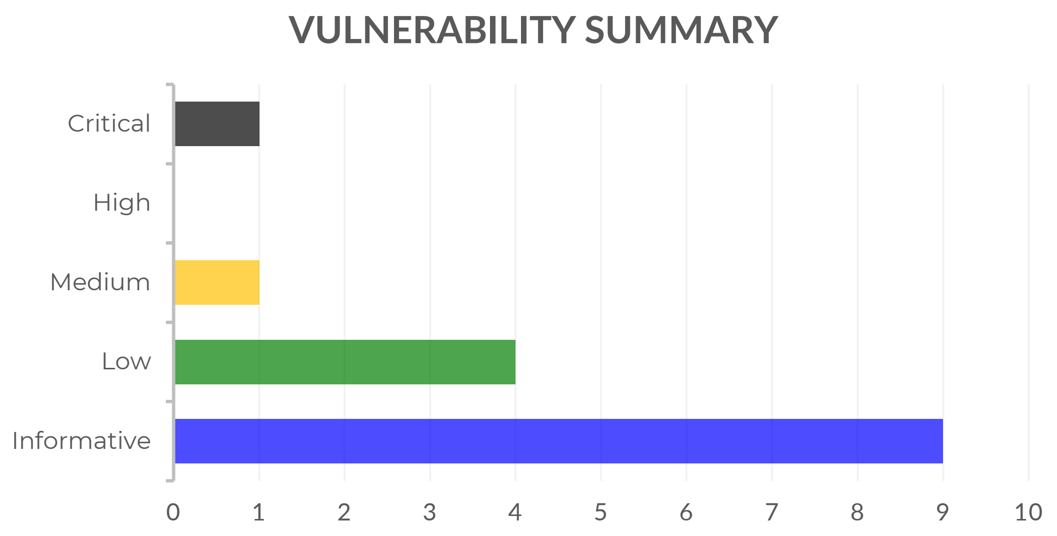
- Repository: https://github.com/flamingo-finance/flamingo-fusd/
    - Commit: `9bb52f6960913e6b05c4c9d0cdd17ca9e2240e7d`
    - Remediations review: `2ce7161344f13279898ca0545bea1422df053cf8`

Executive Summary

The security audit against **Flamingo fUSD** has been conducted between the following dates: **24/10/2022** and **07/11/2022**.

Once the analysis of the technical aspects has been completed, the performed analysis shows that the audited source code contained critical and non-critical vulnerabilities that were mitigated by the Flamingo team.

During the analysis, a total of **15 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.



Conclusions

To this date, **05/12/2022**, all issues found during the audit have been fixed by the **Flamingo** team and the code currently has no known vulnerabilities. However, the general conclusion resulting from the conducted audit, is that the business logic of the project is not completely clear and has various deficiencies that should be studied and resolved by the development team to guarantee the total viability, integrity and availability of the project objectives.

The general conclusions of the performed audit are:

- The security level of the application can be improved and it included hazardous vulnerabilities, a critical vulnerability was detected during the security audit, this vulnerability has been partially fixed and its risk appropriately controlled.
- The quantity and quality of the Unit Tests have been shown to be insufficient and could be improved. The development cycle needs to be reviewed since we have found that unit tests are not covering all the logic of the contracts or all the possible cases of operation, such as: failed cases, large price variations, different vaults and times.
- The logic of the contract allows the owner to alter certain values of the contract at will, allowing to obtain an advantageous position in certain situations and requires user trust in the project. While the contract has features to relinquish some of the owner's privileges throughout the life of the contract, this decision initially relies with the project itself.
- The project currently lacks proper and detailed documentation. There is no business logic or economic model documentation to verify whether the current implementation meets the needs and purpose of the project. Therefore, the economics of the FUSD project could not be audited, and there may be risks to the viability and sustainability of the project, such as discrepancies between the supply and the acquired debt or the correct collateralize of the interest generated.
- It is important to highlight that both, the new N3 blockchain and the compiler of the smart contracts neow3j are new technologies, which are in constant development and have less than a year functioning. For this reason, they are prone to new bugs and breaking changes. Therefore, this audit is unable to detect any future security concerns with neo smart contracts.
- It should be noted that the current audit was performed on a product under development, which is a work in progress. This partial audit consists of the commitments mentioned in the scope section of the report.

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
FUSD02-01	Price Oracle Manipulation	**Critical**	**Fixed**
FUSD02-02	Interdependence of Oracles	**Medium**	**Fixed**
FUSD02-03	Unfair Emergency Liquidations	**Low**	**Fixed**
FUSD02-04	Insecure Collateral and FToken removal	**Low**	**Fixed**
FUSD02-05	Incompatible with non-decimal assets	**Low**	**Closed**
FUSD02-06	Unbounded Loops	**Low**	**Fixed**
FUSD02-07	Invalid Error Message Format	**Low**	**Fixed**
FUSD02-08	Improvable Oracle Path Management	**Informative**	**Fixed**
FUSD02-09	Contracts Management Risks	**Informative**	**Assumed**
FUSD02-10	Safe Storage Access	**Informative**	**Fixed**
FUSD02-11	Safe Contract Update/Destroy	**Informative**	**Assumed**
FUSD02-12	Unsecured Ownership Transfer	**Informative**	**Fixed**
FUSD02-13	Lack of Safe Method Attribute	**Informative**	**Fixed**
FUSD02-14	Lack of Inputs Validation	**Informative**	**Fixed**
FUSD02-15	Incomplete OnNEP17Payment logic	**Informative**	**Fixed**
FUSD02-16	GAS Optimization	**Informative**	**Fixed**

Vulnerability details

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

- Category
- Active
- Risk
- Description
- Recommendations

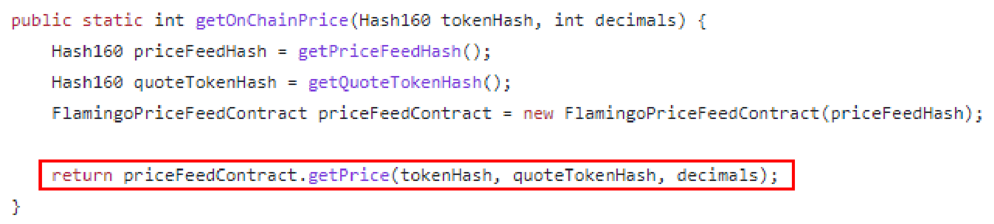
Price Oracle Manipulation

Identifier	Category	Risk	State
FUSD02-01	Design Weaknesses	**Critical**	**Fixed**

The *on-chain* method for obtaining prices of the different assets is susceptible to manipulation by altering the liquidity in the Flamingo pools used to determine the price of the collaterals.

The public method `getPrice` of `FlamingoPriceFeed` uses the balance of the different Flamingo Swap `Pairs` through the `getRatio` function to query the value of the tokens.

The `FTokenVault` of the `getOnChainPrice` uses this contract to obtain the price of collaterals, this price is calculated based on the amount of assets held by the reserve, spot price, but the assets held by the reserve change as users make swaps in the `Pair`, so swaps of a large number of tokens unbalance the pool and offer a ratio that is not adjusted to the reality of the price.



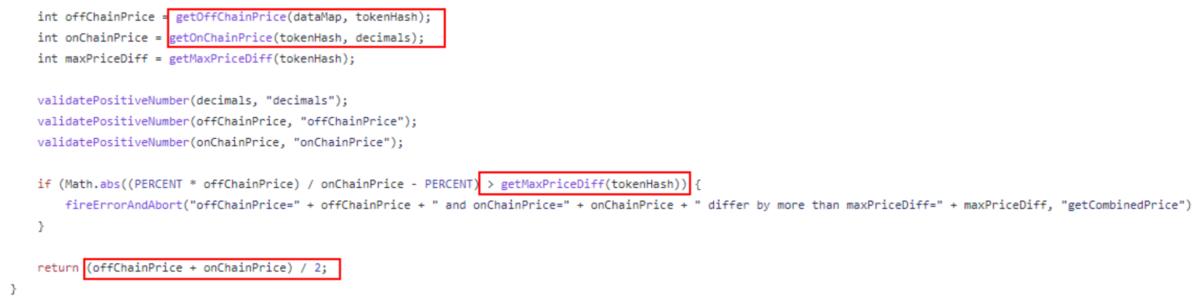
```
public static int getOnChainPrice(Hash160 tokenHash, int decimals) {
    Hash160 priceFeedHash = getPriceFeedHash();
    Hash160 quoteTokenHash = getQuoteTokenHash();
    FlamingoPriceFeedContract priceFeedContract = new FlamingoPriceFeedContract(priceFeedHash);

    return priceFeedContract.getPrice(tokenHash, quoteTokenHash, decimals);
}
```

An attacker can manipulate the oracle price in his favor and use this to mint `FTokens`, or to liquidate vaults, with less collateral than expected.

- First, the attacker uses a flash loan to make a large swap in Flamingo towards the token that will later be used as collateral, for example ETH. (`Token -> ETH`).
- With the price of ETH increased, the attacker mints `FTokens` that, when querying the price of the necessary collateral, obtains a higher price from the Oracle and allows to mint more tokens than those initially collateralized for the real price of ETH.
- Subsequently, it performs the reverse swap in Flamingo swap to rebalance the `Pair` and recover its assets. (`ETH -> Tokens`).
- The attacker returns the flash loan and obtains the benefits, paying only its fees.
- At this point the attacker already makes a profit by undoing the `FUSD` Vault with the difference between the opening price and the current price but could go further and unbalance the `Pair` of the Flamingo Swap in the opposite direction before repaying `FUSD` and raising its benefit by having to make a repay at a lower price.

It must be considered that this entire operation can be carried out in a single transaction and that it does not imply an expense for the attacker greater than the transaction fees, in addition, since a flash loan can be used, initial capital is not required either, although this is not strictly necessary.

Although `FTokenVault` has certain protections such as limiting the price difference between the *on-chain* and *off-chain* (5%) oracles and calculating the average price, which limits this advantage to half the difference limit (2.5%), this limit is per operation and if the `mint` and `repay` of the `FToken` are performed simultaneously, by manipulating the *on-chain* oracle, the full advantage is obtained (5%). Subsequently, the operation can be repeated and the advantage obtained can be sold in other markets or this same attack can be used on the `FTokens` liquidations and the collaterals of `FTokenVault` can be drained.



```
        int offChainPrice = getOffChainPrice(dataMap, tokenHash);
        int onChainPrice = getOnChainPrice(tokenHash, decimals);
        int maxPriceDiff = getMaxPriceDiff(tokenHash);

        validatePositiveNumber(decimals, "decimals");
        validatePositiveNumber(offChainPrice, "offChainPrice");
        validatePositiveNumber(onChainPrice, "onChainPrice");

        if (Math.abs((PERCENT * offChainPrice) / onChainPrice - PERCENT) > getMaxPriceDiff(tokenHash)) {
            fireErrorAndAbort("offChainPrice=" + offChainPrice + " and onChainPrice=" + onChainPrice + " differ by more than maxPriceDiff=" + maxPriceDiff, "getCombinedPrice")
        }

        return (offChainPrice + onChainPrice) / 2;
    }
```

Since spot prices are easy and relatively inexpensive to manipulate, most protocols use a rolling 30-minute time window for TWAP (time-weighted average prices) to calculate the price.

Recommendations

- Use better price sources with manipulation-resistant functions like time-weighted average prices (TWAPs).

References

- https://consensys.github.io/smart-contract-best-practices/attacks/oracle-manipulation
- https://samczsun.com/so-you-want-to-use-a-price-oracle

Source Code References

- FlamingoPriceFeed.java#L238-L239
- FTokenVault.java#L1671-L1677
- FTokenVault.java#L1652

Fixes Review

After discussing with the development team and with the new limits imposed (max minting per block, max price difference and max LTV), the benefit of price manipulation is kept below the cost of the attack, reducing the viability of this scenario.

In addition, it has been decided to add these limits to the liquidation process and create unit tests to cover any possible changes to these limits in the future and their impact on the system.

This issue has been addressed in the following commits:

- https://github.com/flamingo-finance/flamingo-fusd/commit/cc9163756a61922bdf1e9e25f9b48c45d9cc5a0c
- https://github.com/flamingo-finance/flamingo-fusd/commit/0f0bc04e99f386dd54b3a7445fdc47edbd0e0c1b
- https://github.com/flamingo-finance/flamingo-fusd/commit/2ce7161344f13279898ca0545bea1422df053cf8

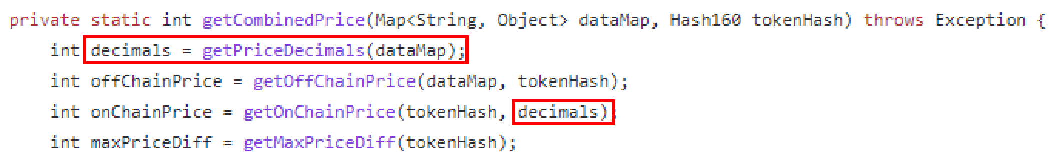
It is important to mention that this scenario is still valid for whitelisted liquidators, but the FUSD team certifies that this role will only be assigned to trusted agents and will guarantee their security.

Interdependence of Oracles

Identifier	Category	Risk	State
FUSD02-02	Design Weaknesses	**Medium**	**Fixed**

The Flamingo FUSD project queries two different Oracles, *on-chain* and *off-chain*, to eliminate the risk of a single point of failure (SPOF), however, certain values of these are interconnected, so an attack on the *off-chain* oracle grants control over the prices of the *on-chain* oracle.

The `getCombinedPrice` method obtains the value of `decimals` from the values provided by the *off-chain* oracle, however, this value is used to query the *on-chain* oracle.


```
private static int getCombinedPrice(Map<String, Object> dataMap, Hash160 tokenHash) throws Exception {
    int decimals = getPriceDecimals(dataMap);
    int offChainPrice = getOffChainPrice(dataMap, tokenHash);
    int onChainPrice = getOnChainPrice(tokenHash, decimals);
    int maxPriceDiff = getMaxPriceDiff(tokenHash);
```

Therefore, if an attacker manages to gain access to the *off-chain* oracle, through his private key or by manipulating the response from the *off-chain* oracle, he can alter the `decimals` value of the tokens so that the prices returned by the *on-chain* oracle are also manipulated.

Due to this problem, the prices of the *on-chain* oracle are dependent on the *off-chain* response, facilitating price manipulation and making it less resilient to *off-chain* attacks.

Recommendations

- The value of `decimals` of each token must be consulted to the *on-chain* oracle or directly to the tokens to eliminate this interdependence of the oracles.

References

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

Source Code References

- FTokenVault.java#L1639-L1641
- FTokenVault.java#L1676
- FlamingoPriceFeed.java#L176

Fixes Review

This issue has been addressed in the following commit:

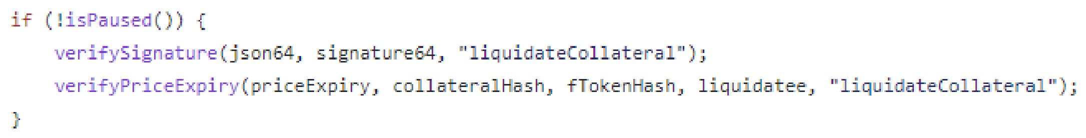
- https://github.com/flamingo-finance/flamingo-fusd/commit/06aa888c29ed53be07e967855cd151c425670d9b

Unfair Emergency Liquidations

Identifier	Category	Risk	State
FUSD02-03	Governance	**Low**	**Fixed**

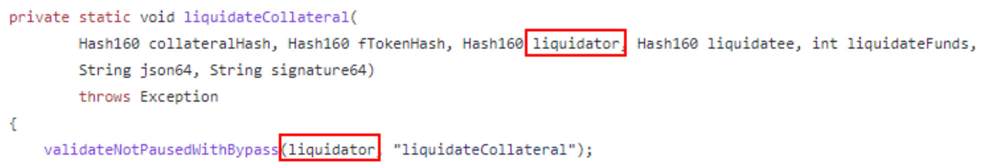
The `FTokenVault` contract contains the necessary methods to perform emergency liquidations by the Owner while the contract is *paused*.

Although this functionality contains several bypasses of the contract's verifications, such as Verifying the Oracle's signature or the expiration of the prices, it may be necessary in emergency cases, however it still diminishes the decentralization of the contract.



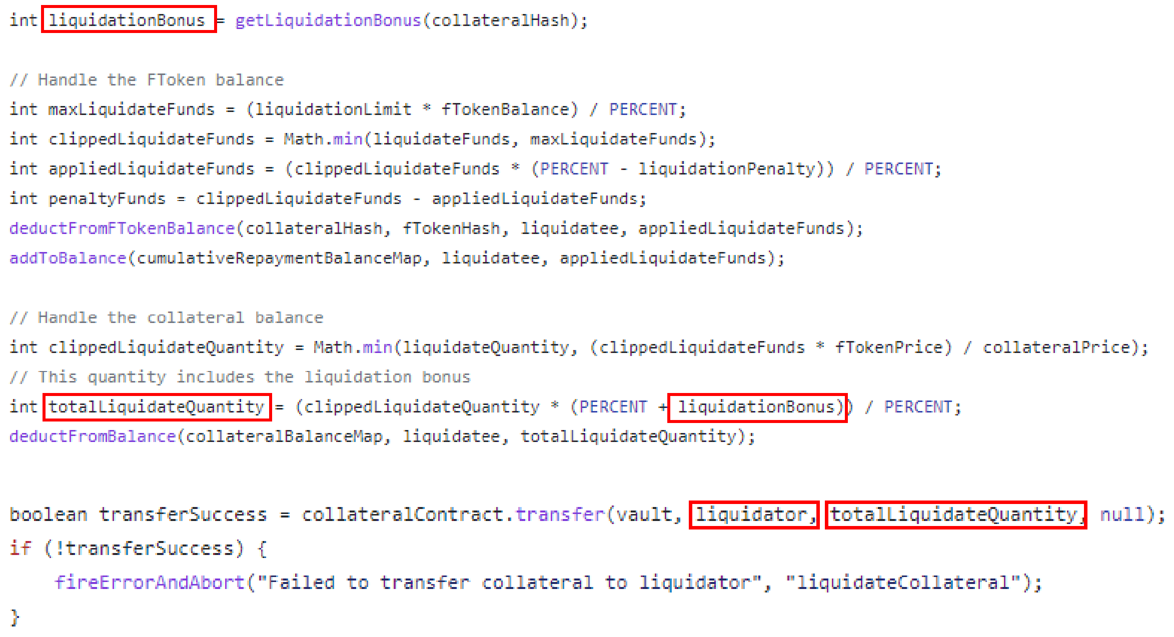
```
if (!isPaused()) {
    verifySignature(json64, signature64, "liquidateCollateral");
    verifyPriceExpiry(priceExpiry, collateralHash, fTokenHash, liquidatee, "liquidateCollateral");
}
```

Additionally, since the `liquidator` argument is used in `validateNotPausedWithBypass`, the only way to fufill this condition when the Vault is *paused* is to be invoked by the Owner.



```
private static void liquidateCollateral(
        Hash160 collateralHash, Hash160 fTokenHash, Hash160 liquidator, Hash160 liquidatee, int liquidateFunds,
        String json64, String signature64)
        throws Exception
{
    validateNotPausedWithBypass(liquidator, "liquidateCollateral");
```

Therefore, it is not understood that during these exceptional cases the liquidation bonus is collected by the Owner of the contract.



```
int liquidationBonus = getLiquidationBonus(collateralHash);

// Handle the FToken balance
int maxLiquidateFunds = (liquidationLimit * fTokenBalance) / PERCENT;
int clippedLiquidateFunds = Math.min(liquidateFunds, maxLiquidateFunds);
int appliedLiquidateFunds = (clippedLiquidateFunds * (PERCENT - liquidationPenalty)) / PERCENT;
int penaltyFunds = clippedLiquidateFunds - appliedLiquidateFunds;
deductFromFTokenBalance(collateralHash, fTokenHash, liquidatee, appliedLiquidateFunds);
addToBalance(cumulativeRepaymentBalanceMap, liquidatee, appliedLiquidateFunds);

// Handle the collateral balance
int clippedLiquidateQuantity = Math.min(liquidateQuantity, (clippedLiquidateFunds * fTokenPrice) / collateralPrice);
// This quantity includes the liquidation bonus
int totalLiquidateQuantity = (clippedLiquidateQuantity * (PERCENT + liquidationBonus)) / PERCENT;
deductFromBalance(collateralBalanceMap, liquidatee, totalLiquidateQuantity);


boolean transferSuccess = collateralContract.transfer(vault, liquidator, totalLiquidateQuantity, null);
if (!transferSuccess) {
    fireErrorAndAbort("Failed to transfer collateral to liquidator", "liquidateCollateral");
}
```

Recommendations

- Reconsider the distribution of the bonuses and the fines during emergency liquidations, as well as the bypass of the Oracle verifications.

Source Code References

- FTokenVault.java#L524
- FTokenVault.java#L601-L614
- FTokenVault.java#L654

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/5aff1e99f9b01f0c864e993aa6acfa704c1249a2

Insecure Collateral and FToken removal

Identifier	Category	Risk	State
FUSD02-04	Business Logic	**Low**	**Fixed**

The `unsupportFToken` and `unsupportCollateral` methods of the `FTokenVault` contract allow the owner to remove collaterals or `FTokens` from accepted assets, however the consequences of these operations on open vaults are not considered.

If an owner decides to delist a collateral or an active FToken, the `validateCollateral` and `validateFToken` verifications will deny the use of the `depositCollateral`, `mintFToken`, `repayFToken`, `liquidateCollateralInternal` and `withdrawCollateral` methods on vaults containing the retired asset.

Any operation on the affected vaults will be blocked; in the case of a collateral removal, users cannot repay the debt and in the case of an FToken removal, users cannot recover their collateral. In both cases, leaving assets blocked in the contract and causing economic losses to users.

Recommendations

- Make sure there are no conflicts with active and open vaults before preventing the use of a collateral or a `Ftoken`.

Source Code References

- FTokenVault.java#L941-L946
- FTokenVault.java#L963-L968

Fixes Review

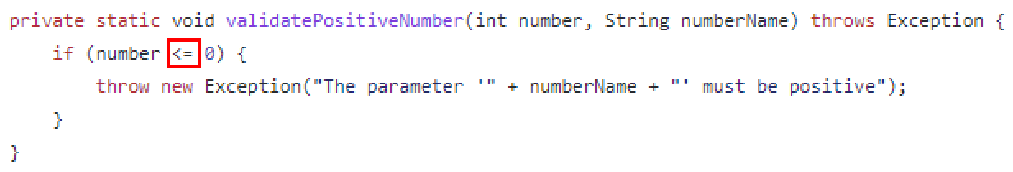
This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/d9aafc1b90d2b3c4762eb8d1cd8680540d55f73c

Incompatible with non-decimal assets

Identifier	Category	Risk	State
FUSD02-05	Business Logic	**Low**	**Closed**

In the `FlamingoPriceFeed` contract, the `getPrice` method uses `validatePositiveNumber` to check that `decimals` is a positive number, however, this also prevents decimals from being equal to zero, so the contract is unable to return token prices without decimals, as is the case of the native token NEO.



```
private static void validatePositiveNumber(int number, String numberName) throws Exception {
    if (number <= 0) {
        throw new Exception("The parameter '" + numberName + "' must be positive");
    }
}
```

Despite the fact that in the past it has been widely discussed whether or not to add decimals to the neo token, these issues have been closed, so the division of neo requires external projects that force us to delegate part of the project's security. In order to limit external risks, it is convenient to allow tokens without decimals in the implementation of our projects or oracles.

Recommendations

- Make sure the token can have 0 decimals.

References

- https://docs.neo.org/docs/en-us/basic/concept/blockchain/token_model.html
- https://github.com/neo-project/neo/issues/2118

Source Code References

- FlamingoPriceFeed.java#L179

Fixes Review

This issue has been discarded after further explanation from the development team.

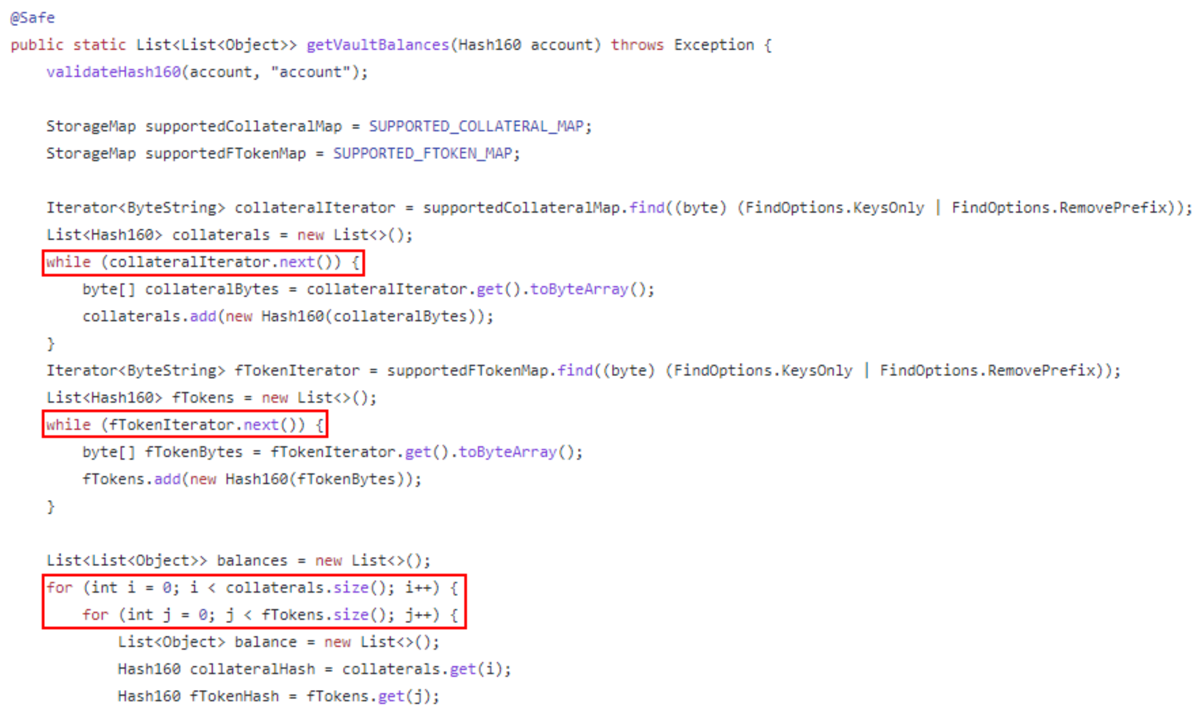
Unbounded Loops

Identifier	Category	Risk	State
FUSD02-06	Denial of Service	**Low**	**Fixed**

A few logics of the contract execute loops that can make too many iterations, which can trigger a Denial of Service (DoS) by GAS exhaustion because it iterates without any limit.

The logic executed in the `getVaultBalances` method might trigger a denial of service (DoS) by GAS exhaustion because it iterates without limits over the storage entries.

Loops without limits are considered a bad practice in the development of Smart Contracts, since they can trigger a Denial of Service or overly expensive executions, this is the case affecting Network Contract.



```java
@Safe
public static List<List<Object>> getVaultBalances(Hash160 account) throws Exception {
    validateHash160(account, "account");

    StorageMap supportedCollateralMap = SUPPORTED_COLLATERAL_MAP;
    StorageMap supportedFTokenMap = SUPPORTED_FTOKEN_MAP;

    Iterator<ByteString> collateralIterator = supportedCollateralMap.find((byte) (FindOptions.KeysOnly | FindOptions.RemovePrefix));
    List<Hash160> collaterals = new List<>();
    while (collateralIterator.next()) {
        byte[] collateralBytes = collateralIterator.get().toByteArray();
        collaterals.add(new Hash160(collateralBytes));
    }
    Iterator<ByteString> fTokenIterator = supportedFTokenMap.find((byte) (FindOptions.KeysOnly | FindOptions.RemovePrefix));
    List<Hash160> fTokens = new List<>();
    while (fTokenIterator.next()) {
        byte[] fTokenBytes = fTokenIterator.get().toByteArray();
        fTokens.add(new Hash160(fTokenBytes));
    }

    List<List<Object>> balances = new List<>();
    for (int i = 0; i < collaterals.size(); i++) {
        for (int j = 0; j < fTokens.size(); j++) {
            List<Object> balance = new List<>();
            Hash160 collateralHash = collaterals.get(i);
            Hash160 fTokenHash = fTokens.get(j);
```

`getVaultBalances` query method return a list of `balances` 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.

Recommendations

- It is convenient to establish arguments that allow paging or limit the expected return.

Source Code References

- FTokenVault.java#L840
- FTokenVault.java#L846
- FTokenVault.java#L852-L853
- FTokenVault.java#L875

Fixes Review

This issue has been addressed in the following commits:

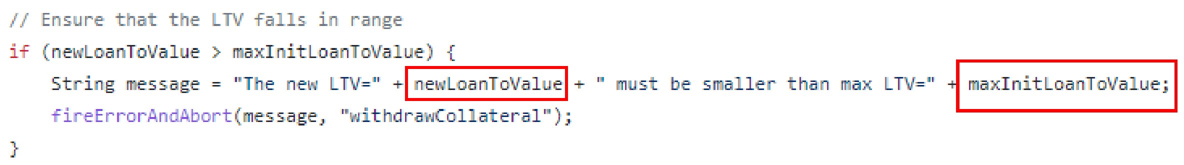
- https://github.com/flamingo-finance/flamingo-fusd/commit/d8e01fe6f24151c9113929ce2a4fb65896960aba
- https://github.com/flamingo-finance/flamingo-fusd/commit/48541da723167e352c8d2800bda2bf2af9abe24e

Invalid Error Message Format

Identifier	Category	Risk	State
FUSD02-07	Auditing and Logging	**Low**	**Fixed**

The different messages of the `FTokenVault` and `FlamingoPriceFeed` smart contracts to display out-of-range values and error messages are not properly formatted.

When constructing the error message, it is intended to display numeric values or hashes, but these are concatenated resulting in an incorrect UTF8 strings, impossible to be represented in N3 due to the `StrictUTF8` property.



```
// Ensure that the LTV falls in range
if (newLoanToValue > maxInitLoanToValue) {
    String message = "The new LTV=" + newLoanToValue + " must be smaller than max LTV=" + maxInitLoanToValue;
    fireErrorAndAbort(message, "withdrawCollateral");
}
```

The concatenated value is the raw value of the integer, not its representation as a text string, this can be fixed by using `StdLib.itoa`.

Recommendations

- Use `StdLib.itoa` for the correct representation of the numeric values as strings.

Source Code References

- FTokenVault.java#L376
- FTokenVault.java#L386
- FTokenVault.java#L443
- FTokenVault.java#L480
- FTokenVault.java#L593
- FTokenVault.java#L620
- FlamingoPriceFeed.java#L208

Fixes Review

This issue has been addressed in the following commits:

- https://github.com/flamingo-finance/flamingo-fusd/commit/7092a22ce488af39c6407b56b8277aa8857634f6
- https://github.com/flamingo-finance/flamingo-fusd/commit/17b4d12975ad3c31c7d20eb3b4c348dda1bbeb84

Improvable Oracle Path Management

Identifier	Category	Risk	State
FUSD02-08	Data Validation	**Informative**	**Fixed**

The management of routes used in `FlamingoPriceFeed` to determine the different `Pairs` used to resolve the price of an asset is subject to improvement to avoid possible failures or invalid routes.

The failures related to the introduction of routes found during the audit are as follows:

- The `setPath` method must verify that `fromToken` and `toToken` are different, being able to take its values from the first and last value of `List<Hash160> path` and saving arguments.
- The `setPath` method must verify that the intermediate elements of a path are never `fromToken` or `toToken`.
- The `setPath` method must verify that the first element of the `path` list corresponds to `fromToken`, and the last element corresponds to `toToken`.
- The `getRatio` method must verify that `baseToken` and `quoteToken` are always different.

Recommendations

- Make sure that the routes entered are valid for the specified pairs.

Source Code References

- FlamingoPriceFeed.java#L147
- FlamingoPriceFeed.java#L153-L156
- FlamingoPriceFeed.java#L176

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/b12cec2cd21719a015d0ead8a736ebf4d085a411

Contracts Management Risks

Identifier	Category	Risk	State
FUSD02-09	Governance	**Informative**	**Assumed**

The design of the logic of the contracts involves certain risks that must be considered and reviewed for possible improvement and decentralization purposes.

Collateral assets risk

The security of the contract depends on the Assets used as collateral, knowing that in N3 the assets can be updated without changing their *Hash* and a token could change its *Symbol* to imitate another, more expensive collateral. It is essential to thoroughly study the collaterals it accepts, as well as to develop possible protections against malicious updates.

System operation risks

The proper functioning of the system also depends on external elements, such as the price oracle or the whitelisted liquidators, as well as on the proper functioning of the liquidation and supply sale processes. Failures, outages, or malicious behavior of these elements will compromise the stability and security of the system.

Owner risk

The Owner maintains certain centralized parts that imply trust in the project; a few of the administrative functionalities are under the control of the project. Therefore, according to the logic of smart contracts, the owner can withdraw funds or take an advantage under specific circumstances.

Compiler risk

Currently, the smart contracts are developed using a programming language that is in an early stage of development for N3 development and bugs are more prevalent than in other smart contract languages like C#, a more stable and currently recommended option for smart contracts developed in NEO.

Safe Storage Access

Identifier	Category	Risk	State
FUSD02-10	Design Weaknesses	**Informative**	**Fixed**

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.

There are different query methods in the project, such as `getOwner`, `getVaultScriptHash` or `totalSupply`, etc. and it would be more convenient to use the following `StorageContext getReadOnlyContext()`. This is particularly important for the methods that are not marked as, `safe` but use the storage on read-only.

Recommendations

- Use `CurrentReadOnlyContext` as long as read-only permissions are needed.

References

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

Source Code References

- FToken.java#L114
- FToken.java#L126
- FToken.java#L148
- FTokenVault.java#L680
- FTokenVault.java#L692
- FTokenVault.java#L704
- FTokenVault.java#L716
- FTokenVault.java#L728
- FTokenVault.java#L740
- FTokenVault.java#L752
- FTokenVault.java#L764
- FTokenVault.java#L776
- FTokenVault.java#L788
- FlamingoPriceFeed.java#L94
- FlamingoPriceFeed.java#L106
- FlamingoPriceFeed.java#L118
- FlamingoPriceFeed.java#L130

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/d64fe7206361f1a80e5d4d768c965b784692d36e

Safe Contract Update/Destroy

Identifier	Category	Risk	State
FUSD02-11	Design Weaknesses	**Informative**	**Assumed**

It is important to mention that the owner of the contract has the possibility of updating/destroying 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 various good practices that help mitigate this problem, such as; add a `TimeLock` to start the `Update` or `Destroy` operations, emit events when the `Update` operation is requested, temporarily disable some contract functionalities and finally execute the `Update` or `Destroy` operation after the `TimeLock` is completed.

Source Code References

- FTokenVault.java#L250
- FTokenVault.java#L255
- FToken.java#L88
- FToken.java#L93
- FlamingoPriceFeed.java#L69
- FlamingoPriceFeed.java#L74

Unsecured Ownership Transfer

Identifier	Category	Risk	State
FUSD02-12	Data Validation	**Informative**	**Fixed**

The modification process of an owner is a delicate process, since the governance of our contract and therefore of the project may be at risk. For this reason, it is recommended to adjust the owner's modification logic, to a logic that allows to verify that the new owner is in fact valid and does exist.

In the case of N3, this process can be carried out in a single transaction without the need to lengthen the process. Transactions in N3 can be signed by multiple accounts and `CheckWitness` could be called with the proposed owner and the current owner simultaneously, provided that the scope of the transaction is properly configured.

Recommendations

- Make sure that the new owner signs accepting the transaction with the proper `Scope`.

Source Code References

- FTokenVault.java#L671
- FToken.java#L105
- FlamingoPriceFeed.java#L85

Fixes Review

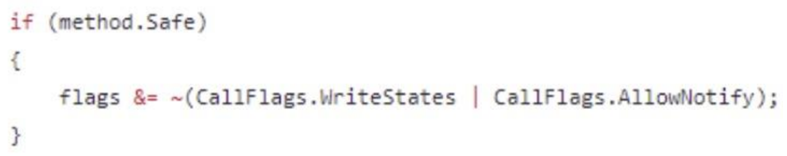
This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/a9490402f0f4c493835dd911a170e11d3f0b7a1f

Lack of Safe Method Attribute

Identifier	Category	Risk	State
FUSD02-13	Design Weaknesses	**Informative**	**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. So, 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

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

Source Code References

- FTokenVault.java#L266

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/c67c8bdbec3b874386f57839505daab3dd7e6b43

Lack of Inputs Validation

Identifier	Category	Risk	State
FUSD02-14	Data Validation	**Informative**	**Fixed**

Certain methods of the different contracts in the `FUSD` project do not properly check the arguments, which can lead to significant errors or unwanted behavior.

Zero addresses

`validateHash160` method in `FToken` and `FlamingoPriceFeed` needs to be updated in order to add the `hash.isZero()` condition.

- FToken.java#L251
- FlamingoPriceFeed.java#L277

Valid address format

The FlamingoPriceFeed's `deploy` function does not check if the argument received meets the format of a valid address.

- FlamingoPriceFeed.java#L64

Recommendations

- It is recommended to always check the format of arguments before using their value, otherwise incorrect values may be set, either intentionally or unintentionally, which can lead to unexpected behavior.

Fixes Review

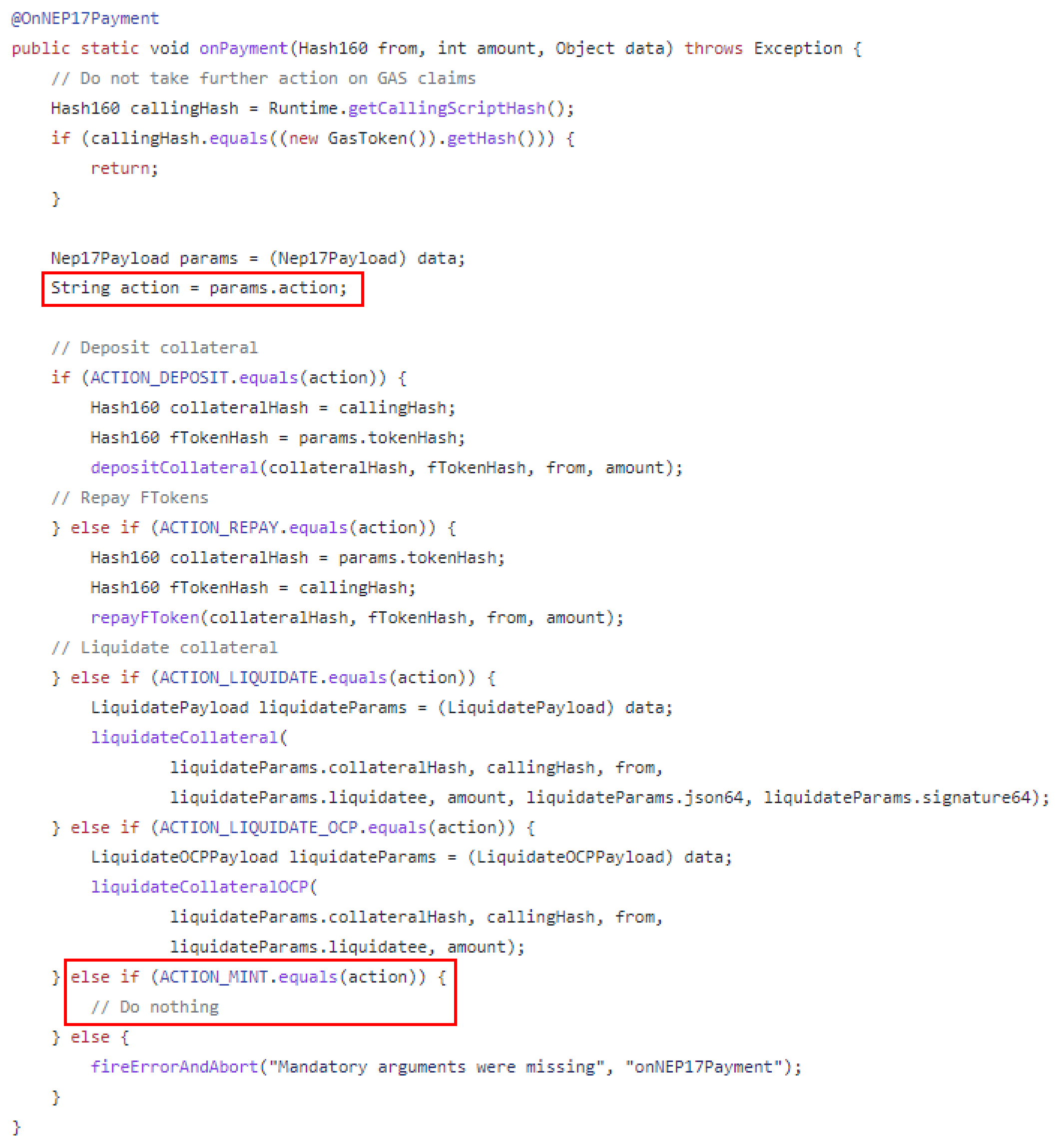
These issues have been addressed in the following commits:

- https://github.com/flamingo-finance/flamingo-fusd/commit/668f15d539c8117c8e581f84088ac14bd239f3fc
- https://github.com/flamingo-finance/flamingo-fusd/commit/9dcdcf831d749bd22b843e8a91933a29313d6e0c

Incomplete OnNEP17Payment logic

Identifier	Category	Risk	State
FUSD02-15	Data Validation	**Informative**	**Fixed**

The `OnNEP17Payment` implementations do not restrict all the types of tokens that contracts can receive, this can lead to users losing tokens due to human errors.



```
@OnNEP17Payment
public static void onPayment(Hash160 from, int amount, Object data) throws Exception {
    // Do not take further action on GAS claims
    Hash160 callingHash = Runtime.getCallingScriptHash();
    if (callingHash.equals((new GasToken()).getHash())) {
        return;
    }

    Nep17Payload params = (Nep17Payload) data;
    String action = params.action;

    // Deposit collateral
    if (ACTION_DEPOSIT.equals(action)) {
        Hash160 collateralHash = callingHash;
        Hash160 fTokenHash = params.tokenHash;
        depositCollateral(collateralHash, fTokenHash, from, amount);
    // Repay FTokens
    } else if (ACTION_REPAY.equals(action)) {
        Hash160 collateralHash = params.tokenHash;
        Hash160 fTokenHash = callingHash;
        repayFToken(collateralHash, fTokenHash, from, amount);
    // Liquidate collateral
    } else if (ACTION_LIQUIDATE.equals(action)) {
        LiquidatePayload liquidateParams = (LiquidatePayload) data;
        liquidateCollateral(
                liquidateParams.collateralHash, callingHash, from,
                liquidateParams.liquidatee, amount, liquidateParams.json64, liquidateParams.signature64);
    } else if (ACTION_LIQUIDATE_OCP.equals(action)) {
        LiquidateOCPPayload liquidateParams = (LiquidateOCPPayload) data;
        liquidateCollateralOCP(
                liquidateParams.collateralHash, callingHash, from,
                liquidateParams.liquidatee, amount);
    } else if (ACTION_MINT.equals(action)) {
        // Do nothing
    } else {
        fireErrorAndAbort("Mandatory arguments were missing", "onNEP17Payment");
    }
}
```

The `onPayment` method contemplates multiple cases such as `DEPOSIT`, `REPAY`, `LIQUIDATE_OCP` and `LIQUIDATE`, and it also considers if the collaterals are accepted. However, it does not contemplate the sending of Tokens with and `ACTION_MINT` that are not valid `FTokens`.

Therefore, if a user sends any tokens with `ACTION_MINT` action argument to the contract, the funds will be inevitably locked. The verification that `callingHash` is an accepted `FToken` with `validateFToken(callingHash)` when the action is `ACTION_MINT` should be performed.

Recommendations

- Presumably the `OnNEP17Payment` method should limit the receipt of tokens with `ACTION_MINT` action, with the exception of `GAS` or `FTokens` during their minting to the `Vault`, revoking the transaction in the case of other tokens or actions.

Source Code References

- FTokenVault.java#L319

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/17b4d12975ad3c31c7d20eb3b4c348dda1bbeb84

GAS Optimization

Identifier	Category	Risk	State
FUSD02-16	Codebase Quality	**Informative**	**Fixed**

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.

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

- FToken.java#L36-L51
- FTokenVault.java#L70-L215
- FlamingoPriceFeed.java#L33-L47

Unnecessary logic

Redundant verification processes have been identified that by simply eliminating them it will produce GAS savings.

In Neo2 it was necessary to verify that an account was the same as the `CallingScriptHash` in order to be able to make transfers between contracts, in N3 this is no longer necessary, since it is done automatically in the native `CheckWitness` syscall.

Source Code References

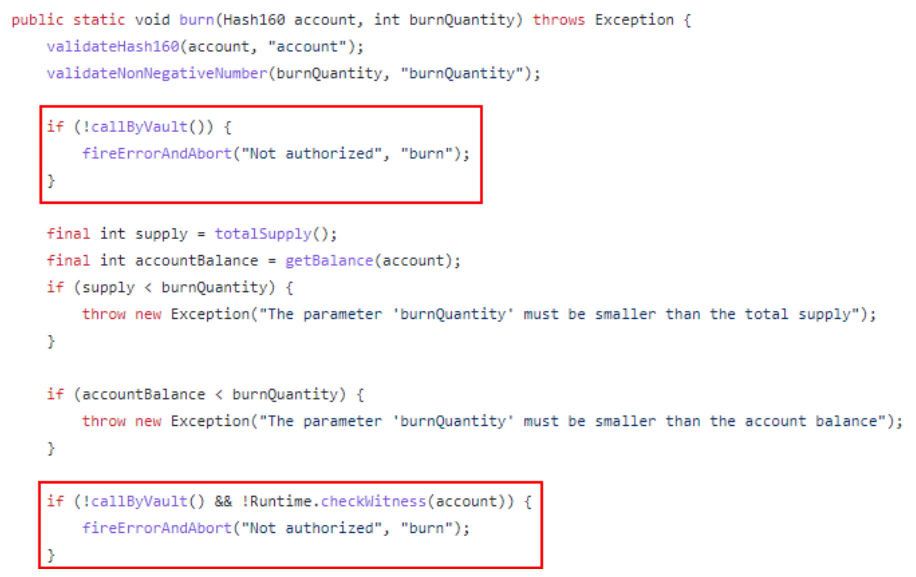
- FToken.java#L300

It is also possible to remove the verification that ensures that the `totalSupply` will not overflow, since it would mean that the accounting was wrong in another point.

Source Code References

- FToken.java#L217-L219

Additionally, the following conditional `if` statement is totally redundant since this verification has been previously carried out, as can be seen in the following image, and it can be eliminated, optimizing the function and saving GAS accordingly.



```
public static void burn(Hash160 account, int burnQuantity) throws Exception {
    validateHash160(account, "account");
    validateNonNegativeNumber(burnQuantity, "burnQuantity");

    if (!callByVault()) {
        fireErrorAndAbort("Not authorized", "burn");
    }

    final int supply = totalSupply();
    final int accountBalance = getBalance(account);
    if (supply < burnQuantity) {
        throw new Exception("The parameter 'burnQuantity' must be smaller than the total supply");
    }

    if (accountBalance < burnQuantity) {
        throw new Exception("The parameter 'burnQuantity' must be smaller than the account balance");
    }

    if (!callByVault() && !Runtime.checkWitness(account)) {
        fireErrorAndAbort("Not authorized", "burn");
    }
```

Source Code References

- FToken.java#L225-L227

The `getDirectPrice` private method of the `FlamingoPriceFeed` contract verifies that the obtained parameters are valid through the `validatePositiveNumber` and `validateHash160` methods, however, this logic is unnecessary and redundant since these checks have been previously carried out in the `getPrice` public method.

Source Code References

- FlamingoPriceFeed.java#L179
- FlamingoPriceFeed.java#L201

Serialization improvements

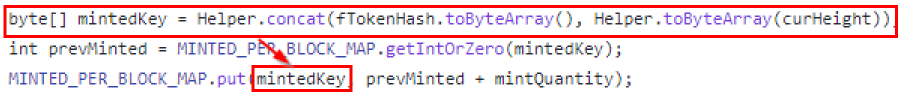
In the `FlamingoPriceFeed` contract, the saved pairs are made by concatenating the hashes, which forces to split them each time they have to be queried. This process can be made more low-cost by using the `StdLib.Serialize` and `StdLib.Deserialize` of objects. This will make the code more optimal and readable.

Source Code References

- FlamingoPriceFeed.java#L140-L144
- FlamingoPriceFeed.java#L152-L157

Inefficient storage

When saving the amount minted in the current block, the height of the block is used as a storage key, this implies that a registry is created per stored block, unnecessarily wasting resources and gas, it is convenient to use a structure that stores the last saved height, so that the key is reused in the next block, in this way gas can be saved thanks to the compensation system of the N3 storage and also favor the growth of the chain to be less exponential and more scalable.



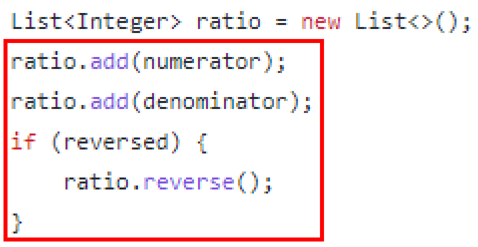
```
byte[] mintedKey = Helper.concat(fTokenHash.toByteArray(), Helper.toByteArray(curHeight));
int prevMinted = MINTED_PER_BLOCK_MAP.getIntOrZero(mintedKey);
MINTED_PER_BLOCK_MAP.put(mintedKey, prevMinted + mintQuantity);
```

Source Code References

- FTokenVault.java#L1256-L1258

Logic optimization

The `getRatio` method of the `FlamingoPriceFeed` contract allows to obtain the existing ratio between two tokens. As can be seen in the following image, in certain cases, said ratio can be returned in reverse, so a `reverse` of the `List` is performed.



```
List<Integer> ratio = new List<>();
ratio.add(numerator);
ratio.add(denominator);
if (reversed) {
    ratio.reverse();
}
```

However, it is considered more optimal to add the reverse ratio directly in the cases that it is necessary, instead of inverting it later as it is done in the current implementation.

Source Code References

- FlamingoPriceFeed.java#L247-L251

Fixes Review

These issues have been addressed in the following commit:

- https://github.com/flamingo-finance/flamingo-fusd/commit/79eff3b9cc282d2ed3adcab995df6b7f0ae65159

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 possible 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 behaviours that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

RED4SEC

Invest in Security, invest in your future