



Security Audit Report

09/02/2024

Flamingo OrderBook v2

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



Content

Introduction	4
Disclaimer	4
Scope	5
Executive Summary.....	6
Conclusions	7
Vulnerabilities	8
List of vulnerabilities	8
Vulnerability details	9
Storage collision	10
Abuse of negative values	11
Lack of inputs validation	12
Reentrancy pattern	13
Deposit on behalf of the contract	15
Unrestricted token receipt.....	16
Absence of unit test	17
Redundant arithmetic operations	18
Limit call rights.....	19
Unbounded loop in method	20
Lack of documentation	21
Possible transfer malfunction.....	22
Owner overprivileged access	23
Missing state change events.....	24
Unsecured ownership transfer	25
Safe contract update	26
Code quality.....	27
GAS optimization	30
Inefficient assertion usage	33
Optimize storage prefixes	34
Safe storage access	35
Bad coding practices	36
Optimize validation in storage operations.....	37
Incomplete manifest	38
Use list instead of an array	39
Outdated framework	40
Lack of safe method attribute.....	41
Information Disclosure in Repository	42

Annexes 43

Methodology 43

Manual Analysis..... 43

Automatic Analysis..... 43

Vulnerabilities Severity 44

Introduction

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



OrderBook V2 is designed to be used in conjunction with the AMM. The takers are forced to buy and sell both in the AMM and in the order book. Forcing the trading to happen in both places has the objective to ensure that arbitragers always have to trigger limit orders when trading.

As solicited by **Neo** and as part of the vulnerability review and management process, Red4Sec has been requested to perform a security code audit in order to evaluate the security of the **Flamingo OrderBook v2** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of **Flamingo OrderBook v2**. The performed analysis shows that the smart contract does contain critical and high-risk vulnerabilities.

Disclaimer

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

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

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

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

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

Scope

Red4Sec Cybersecurity has made a thorough audit of the **Flamingo OrderBook v2** 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/OrderBook>
 - commit: 87c63999d9af6fceb116cc81298cc2043580e7fe

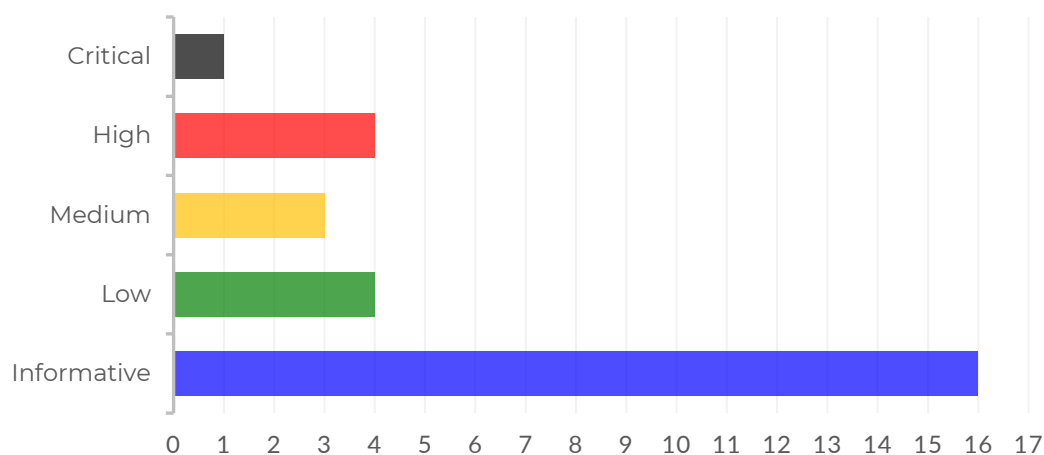
Executive Summary

The security audit against **Flamingo OrderBook v2** has been conducted between the following dates: **05/02/2024** and **09/02/2024**.

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

During the analysis, a total of **28 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.

VULNERABILITY SUMMARY



Conclusions

Based on the comprehensive list of vulnerabilities and their associated risks identified during the smart contract audit, it is evident that the **OrderBook** project is exposed to a significant level of security risk.

The absence of input verification mechanisms, which has been a recurring concern communicated in previous audits, represents a foundational flaw in the project's security architecture. Such a deficiency not only facilitates the exploitation of vulnerabilities like storage collision, abuse of negative value, lack of inputs validation, and reentrancy patterns—all classified with high to critical risk levels—but also undermines the integrity and reliability of the contract operations.

The overall quality and organization of the code are subpar, contributing to a less-than-optimal project structure that does not adhere to standard coding practices. The code contains exceedingly long methods and files, which hampers readability and complicates maintenance and auditing. This non-alignment with professional standards is unsuitable for a project designed to handle large volumes of financial assets.

The lack of a well-configured development environment and unit tests indicates that sufficient measures have not been taken to ensure the project's functionality and security. This deficiency, combined with the absence of economic models, raises concerns about the project's viability and sustainability.

To mitigate these risks and align the project with industry best practices, it is imperative to address the identified vulnerabilities promptly and comprehensively. This includes implementing input verification mechanisms, adhering to standard coding practices to enhance code quality and organization, and establishing a robust development methodology that includes unit testing and economic modeling. Additionally, revising the development cycle to ensure the functionality and accuracy of unit tests, as well as adjusting the contract's configuration to prevent owner overprivileged access, are critical steps toward securing the project.

Other low-risk vulnerabilities like limit call rights, unbounded loops, and several informational risks such as missing state change events, insecure ownership transfer, inefficient assertion usage, outdated framework, and others, pose minimal risks but constitute areas for improvement and should not be ignored.

It is imperative to contextualize that this audit was carried out as a partial audit, emphasizing its nature as a work in progress and requiring more tests and development to be completed. As such, while the audit provides valuable information, it may not cover all potential vulnerabilities.

In conclusion, the project is currently at a high risk of exploitation and failure due to the numerous critical and high-level vulnerabilities identified. Immediate action is required to address these issues and implement a more rigorous and standardized approach to development and security. Failure to do so could result in significant financial losses and damage to the project's reputation. The project needs to enhance a culture of continuous learning and improvement to evolve its security practices and mitigate risks effectively in the future.

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
FOB-01	Storage collision	Critical	Fixed
FOB-02	Abuse of negative values	High	Fixed
FOB-03	Lack of inputs validation	High	Open
FOB-04	Reentrancy pattern	High	Fixed
FOB-05	Deposit on behalf of the contract	High	Fixed
FOB-06	Unrestricted token receipt	Medium	Open
FOB-07	Absence of unit test	Medium	Open
FOB-08	Redundant arithmetic operations	Medium	Open
FOB-09	Limit call rights	Low	Fixed
FOB-10	Unbounded loop in method	Low	Open
FOB-11	Lack of documentation	Low	Open
FOB-12	Possible transfer malfunction	Low	Open
FOB-13	Owner overprivileged access	Informative	Open
FOB-14	Missing state change events	Informative	Open
FOB-15	Unsecured ownership transfer	Informative	Open
FOB-16	Safe contract update	Informative	Open
FOB-17	Code quality	Informative	Open
FOB-18	GAS optimization	Informative	Open
FOB-19	Inefficient assertion usage	Informative	Open
FOB-20	Optimize storage prefixes	Informative	Open
FOB-21	Safe storage access	Informative	Open
FOB-22	Bad coding practices	Informative	Open
FOB-23	Optimize validation in storage operations	Informative	Open
FOB-24	Incomplete manifest	Informative	Open

FOB-25	Use list instead of an array	Informative	Open
FOB-26	Outdated framework	Informative	Open
FOB-27	Lack of safe method attribute	Informative	Open
FOB-28	Information Disclosure in Repository	Informative	Open

Vulnerability details

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

- Category
- Active
- Risk
- Description
- Recommendations

Storage collision

Identifier	Category	Risk	State
FOB-01	Business Logic	Critical	Fixed

The TakerFeeStorage and UserOrderStorage storage classes use the same prefixes. This issue could allow an attacker to inject arbitrary input into certain parts of the contract, affecting user orders and taker fee.

Following, find the storage classes that reuse the same storage prefix.

```
public static class UserOrderStorage
{
    public class key
    {
        public UInt160 owner;
        public BigInteger id;
    }

    private static readonly byte[] userOrderPrefix = new byte[] { 0x01, 0x15 };
```

UserOrderStorage prefix

```
public static class TakerFeeStorage
{
    private static readonly byte[] takerFeePrefix = new byte[] { 0x01, 0x15 };
```

TakerFeeStorage Prefix

Recommendations

- Use different prefixes.
- Use inheritance to avoid duplicating code, which prevents human errors when copying and pasting from previous code.

Source Code References

- [OrderBook/OrderBook.Storage.cs#L386](#)

Fixes Review

The issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/OrderBook/pull/1>

Abuse of negative values

Identifier	Category	Risk	State
FOB-02	Business Logic	High	Fixed

The vulnerability in the **OrderBook** contract is due to a lack of input validation in critical methods such as `Sell`, `Buy`, `SellLimit`, and `BuyLimit`, allowing the use of negative values because the `Increase` and `Decrease` methods of `AccountStorage` do not reject negative numbers.

This exposes the system to exploits that could allow attackers to improperly manipulate balances, draining funds or altering the expected behavior of transactions. The absence of adequate controls compromises the security of user funds and the operational integrity of the contract, underscoring the critical need to implement positive input validations and adjustments to `AccountStorage` logic to prevent the introduction of negative values into transactions. These problems are due to a lack of verification as highlighted in the issue `Lack of inputs validation`.

As can be seen, the aforementioned methods do not produce an error when receiving negative inputs.

```
internal static void Increase(UInt160 user, UInt160 asset, BigInteger amount)
{
    if (amount <= 0)
        return;
    BigInteger pre = Get(user, asset);
    OnAccountUpdate(user, asset, pre, pre + amount);
    Put(user, asset, pre + amount);
}

internal static void Decrease(UInt160 user, UInt160 asset, BigInteger amount)
{
    if (amount <= 0)
        return;
```

Does not produce error with negative numbers

Recommendations

- Modify the verification with an `Assert` that prevents continuing with the execution of the contract.

Source Code References

- [OrderBook/OrderBook.Storage.cs#L268](#)
- [OrderBook/OrderBook.Storage.cs#L277](#)
- [OrderBook/OrderBook.Storage.cs#L437](#)
- [OrderBook/OrderBook.Storage.cs#L445](#)
- [OrderBook/OrderBook.Order.cs#L140-L147](#)
- [OrderBook/OrderBook.Order.cs#L179-L186](#)
- [OrderBook/OrderBook.cs#L388-L400](#)

Fixes Review

The issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/OrderBook/pull/2>

Lack of inputs validation

Identifier	Category	Risk	State
FOB-03	Data Validation	High	Open

All the methods in the **OrderBook** project do not properly check any user arguments, which can lead to major errors. The general execution is not checking the inputs nor the integrity of most the `UInt160` types, nor does it verify whether the addresses that should be contracts are indeed so. Additionally, there are instances where integers with negative values are not properly validated.

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

For example, in the `Withdraw` method it delegates all security to a third-party contract, since it allows the entry of negative values.

```
public static void Withdraw(UInt160 user, UInt160 asset, BigInteger amount)
{
    //avoid recall
    Assert(EnteredStorage.Get() == 0, "OnNEP11Payment:Re-entered");
    EnteredStorage.Put(1);
    Assert(Runtime.CheckWitness(user), "Permission Invalid");
    BigInteger _amount = AccountStorage.Get(user, asset);
    if(amount > _amount)
    {
        amount = _amount;
    }

    SafeTransferNep17(asset, Runtime.ExecutingScriptHash, user, amount);
    AccountStorage.Decrease(user, asset, amount);
    EnteredStorage.Put(0);
}
```

Recommendations

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

Source Code References

- [OrderBook/OrderBook.Amm.cs](#)
- [OrderBook/OrderBook.Owner.cs](#)
- [OrderBook/OrderBook.Extend.cs](#)
- [OrderBook/OrderBook.Account.cs](#)
- [OrderBook/OrderBook.Order.cs](#)
- [OrderBook/OrderBook.cs](#)

Reentrancy pattern

Identifier	Category	Risk	State
FOB-04	Timing and State	High	Fixed

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

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

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

In the case of the **OrderBook** contract, there are many methods that update the values after making transfers, so calls can be redirected to any method before updating the values.

Recommendations

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

References

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

Source Code References

- [OrderBook/OrderBook.Account.cs#L35](#)
- [OrderBook/OrderBook.Account.cs#L50](#)
- [OrderBook/OrderBook.Account.cs#L58](#)
- [OrderBook/OrderBook.cs#L143-L157](#)
- [OrderBook/OrderBook.cs#L416](#)
- [OrderBook/OrderBook.cs#L493](#)
- [OrderBook/OrderBook.cs#L550](#)

Fixes Review

The issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/OrderBook/pull/3>

Deposit on behalf of the contract

Identifier	Category	Risk	State
FOB-05	Authentication	High	Fixed

It is possible to break the relationship between `AccountStorage` and the user's actual balance due to a lack of verification of the depositor's signature.

The `Deposit` method delegates the verification of the user's signature to the `asset` contract concerned. However, an attacker could establish the contract itself as a user, effectively bypassing signature verification when invoking the `asset` contract.

```
public static void Deposit(UInt160 user, UInt160 asset, BigInteger amount)
{
    //avoid recall
    Assert(EnteredStorage.Get() == 0, "OnNEP11Payment:Re-entered");
    EnteredStorage.Put(1);

    Assert(asset == BaseToken || asset == QuoteToken, "asset Wrong");

    SafeTransferNep17(asset, user, Runtime.ExecutingScriptHash, amount);

    AccountStorage.Increase(user, asset, amount);

    EnteredStorage.Put(0);
}
```

Absence of `CheckWitness`

This means that the balance of the contract itself can be increased arbitrarily, altering the relationship between the balance of the account and the balance of the contract. Moreover, it grants the administrator the ability to deny service altogether.

Recommendations

- Check the user signature by using `CheckWitness`.

Source Code References

- [OrderBook/OrderBook.Account.cs#L32-L45](#)

Fixes Review

The issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/OrderBook/pull/4>

Unrestricted token receipt

Identifier	Category	Risk	State
FOB-06	Business Logic	Medium	Open

The `OnNEP17Payment` implementations fail to limit the number of tokens that contracts can receive. This lack of validation can lead to users losing tokens due to human error. Specifically, if a user sends tokens like NEO or GAS directly to the contract, the funds will be locked.

From a technical perspective, the `OnNEP17Payment` function is automatically invoked when NEP-17 tokens are transferred to a contract that implements this interface, intended for managing incoming payments. However, in the absence of proper checks, there is no guarantee that received tokens will be managed correctly. This scenario may result in tokens becoming "trapped" within the contract, leaving users with no evident means of recovery.

Recommendations

- The `OnNEP17Payment` method should limit token receipt to only traded tokens, revoking the transaction for other tokens.

Source Code References

- [OrderBook/OrderBook.Account.cs#L27](#)

Absence of unit test

Identifier	Category	Risk	State
FOB-07	Testing and Documentation	Medium	Open

During the security review, we identified the absence of Unit Tests, a best practice that is not only highly recommended but has also become mandatory for projects managing large amounts of capital.

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

Note: *The criticality of this issue has been increased due to the inability to test the code in a controlled environment, which affects the audit, and therefore the security of the audited code.*

Unit tests play a crucial role for the following reasons:

- **Early error detection:** It assists in identifying and rectifying errors at an early stage of the software development process.
- **Enhanced code quality:** By confirming that each unit operates correctly in isolation, unit tests contribute to improved overall code quality.
- **Problem identification and resolution:** It facilitates pinpointing and resolve issues, both current and future.
- **Foundation for new features:** Unit tests establish a solid foundation for the development of new features, ensuring its proper integration.
- **Confidence in code:** it increases the confidence in the code, allowing for changes and updates with the assurance that existing functionalities remain working.
- **Simplified integration:** Unit tests facilitate seamless integration and enhance the collaboration process within development teams.

Redundant arithmetic operations

Identifier	Category	Risk	State
FOB-08	Business Logic	Medium	Open

The `AddOrder` method of the **OrderBook** contract contains a redundant arithmetic operation that does not alter the value of the `priceFixed` variable but introduces an unnecessary risk implying loss of precision.

The operation is to divide and then multiply `priceFixed` by `MinDecimal`, which should theoretically return the original value of `priceFixed`. However, due to the characteristics of floating-point arithmetic in computing. This sequence of operations can result in a loss of precision due to intermediate rounding, especially in systems that limit the precision of floating-point numbers. In the context of a smart contract, where the precision and correctness of financial operations are critical, such practices can lead to unforeseen results, affecting the logic of the contract and potentially compromising the integrity of financial transactions.

```
public static BigInteger AddOrder(UInt160 user, BigInteger priceFixed, BigInteger amount, bool isBuy)
{
    Assert(Runtime.CheckWitness(user), "Permission Invalid");
    OrderIdStorage.Increase();
    BigInteger curOrderId = OrderIdStorage.Current();
    priceFixed = priceFixed / MinDecimal * MinDecimal;
    BigInteger worth = amount * priceFixed / PriceDecimalFixed;
    BigInteger highestOrderRound = AddToTree(priceFixed, amount, worth, isBuy);
    if (isBuy)
    {
        //decrease baseToken
        AccountStorage.Decrease(user, BaseToken, amount * priceFixed / PriceDecimalFixed);
    }
}
```

Implementing arithmetic operations without proper consideration of the precision and order of the operations may compromise the logic of the contract, resulting in incorrect financial calculations. This is particularly critical in blockchain applications, where accuracy in transactions and fee calculations is critical to the integrity and trust in the system.

Recommendations

- Review and eliminate the sequence of redundant operations that do not contribute to changing the value of the variable, in this case, successive division and multiplication by `MinDecimal`. If the purpose of these operations is to ensure a certain precision or format, consider alternative approaches that do not introduce the risk of loss of precision.
- Implement validation and rounding mechanisms that ensure that the results of arithmetic operations conform to business expectations and rules.

Source Code References

- [OrderBook/OrderBook.Order.cs#L137](#)

Limit call rights

Identifier	Category	Risk	State
FOB-09	Design Weaknesses	Low	Fixed

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.

Hence, when invoking any contract, it is expected to be a read-only operation, such as retrieving the user's balance. In this context, such calls should consistently be made with the `ReadOnly` flag, rather than `CallFlags.All`.

```
public static UInt160 GetToken0FromPair(UInt160 pair)
{
    return (UInt160)Contract.Call(pair, "getToken0", CallFlags.All, new object[] { });
}
```

Recommendations

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

References

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

Source Code References

- [OrderBook/OrderBook.Extend.cs#L63](#)
- [OrderBook/OrderBook.Extend.cs#L92](#)
- [OrderBook/OrderBook.Extend.cs#L97](#)
- [OrderBook/OrderBook.Extend.cs#L102](#)
- [OrderBook/OrderBook.Extend.cs#L107](#)
- [OrderBook/OrderBook.Extend.cs#L112](#)
- [OrderBook/OrderBook.Extend.cs#L117](#)
- [OrderBook/OrderBook.Extend.cs#L122](#)
- [OrderBook/OrderBook.Extend.cs#L129](#)

Fixes Review

The issue has been addressed in the following pull request:

- <https://github.com/flamingo-finance/OrderBook/pull/5>

Unbounded loop in method

Identifier	Category	Risk	State
FOB-10	Denial of Service	Low	Open

Certain logic 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 `GetOrdersOf` method might trigger a denial of service (DoS) by GAS exhaustion because it iterates without limits over the user's orders.

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.

```
[Safe]
public static Order[] GetOrdersOf(UInt160 maker, uint pos, uint n)
{
    var results = new Order[n];

    var iterator = UserOrderStorage.Find(maker);
    for (int i = 0; i < pos + n; i++)
    {
        if (iterator.Next() && i >= pos)
        {
            results[i - pos] = (Order)iterator.Value;
        }
    }
    return results;
}
```

Uncontrolled loop

Source Code References

- [OrderBook/OrderBook.Order.cs#L32](#)

Lack of documentation

Identifier	Category	Risk	State
FOB-11	Testing and Documentation	Low	Open

The project does not contain sufficient technical documentation, execution flowcharts, class diagrams or properly commented code. This is a bad practice that complicates the understanding of the project and makes it difficult for the audit team to analyze the functionalities, since there is not enough technical documentation to verify whether 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.

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

Recommendations

- Comment public methods in the code in a way that makes the code more understandable and auditable.

References

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

Source Code References

- [OrderBook/OrderBook.cs#L185](#)
- [OrderBook/OrderBook.Order.cs#L44](#)

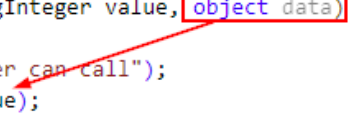
Possible transfer malfunction

Identifier	Category	Risk	State
FOB-12	Design Weaknesses	Low	Open

A deficiency has been identified in the `ApprovedTransfer` method with the `data` argument that may cause unexpected behavior in the implementation of future tokens.

Failure to use the `data` argument in transfer operations implies a risk that the contract will not meet certain requirements or expected logic in the context of NEP17 tokens. Since the NEP17 standard does not specify a specific purpose for the `data` argument, but includes it in its definition, it is presumed that it may have important applications in certain cases.

```
public static bool ApprovedTransfer(UInt160 asset, UInt160 to, BigInteger value, object data)
{
    Assert(Runtime.CallingScriptHash == SwapRouter, "only ammRouter can call");
    SafeTransferNep17(asset, Runtime.ExecutingScriptHash, to, value);
    return true;
}
```



Unused data argument

Failure to implement this argument limits the flexibility and adaptability of the contract to different types of tokens and their possible unique requirements.

Recommendations

- Adjust the `ApprovedTransfer` implementation to properly incorporate and handle the `data` argument. This would ensure that the contract can adapt to different token requirements that might arise in the future.
- Ensure that all modifications and their purpose are clearly documented. Communicate these changes to developers and users to maintain transparency and facilitate understanding of the contract.

Source Code References

- [OrderBook/OrderBook.cs#L159](#)

Owner overprivileged access

Identifier	Category	Risk	State
FOB-13	Governance	Informative	Open

The logical design of the OrderBook contracts introduces certain minor risks that should be reviewed and considered for their improvement.

The current implementation grants the Owner full control over the smart contract. On one instance with methods such as `Withdraw` that allows draining the contract. However, it is more noteworthy the condition in the `verify` method that allows the Owner to do any action, even if it is not directly coded in the contract. This includes transferring tokens like NEO or GAS, updating, destroying, or even participating in consensus voting without using any contract method, and building the transaction directly.

Recommendations

- Limit access to the `verify` method and restrict its use exclusively to strictly necessary actions.
- Review and restrict the capabilities granted to the contract owner. This may include removing the ability to perform critical actions without the consent of other stakeholders or additional validation.

Source Code References

- [OrderBook/OrderBook.Owner.cs#L16](#)
- [OrderBook/OrderBook.Account.cs#L47](#)

Missing state change events

Identifier	Category	Risk	State
FOB-14	Auditing and Logging	Informative	Open

The **OrderBook** contract implementation fails to adhere to the best practice of emitting events upon significant changes in the states of a smart contract.

This practice is vital for ensuring transparency and facilitating effective monitoring of contractual conditions that may influence execution and user decisions. However, the absence of events during modifications to contract operating values, such as changes in rates or fees, presents a vulnerability in terms of auditing and transparency.

This omission restricts the ability of DApps, automated processes, and users to receive notifications about significant changes in contract status, essential for real-time adaptation and response to new operating conditions.

Recommendations

- The contract must be modified to include the emission of specific events each time a change is made to the operating values of the contract.

Source Code References

- [OrderBook/OrderBook.Owner.cs#L26](#)
- [OrderBook/OrderBook.Owner.cs#L34](#)

Unsecured ownership transfer

Identifier	Category	Risk	State
FOB-15	Design Weaknesses	Informative	Open

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

- [OrderBook/OrderBook.Owner.cs#L46](#)

Safe contract update

Identifier	Category	Risk	State
FOB-16	Design Weaknesses	Informative	Open

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

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

Recommendations

- Several effective practices can mitigate this issue. Such as; add a `TimeLock` to start the `Update` operations, emit events when the `Update` operation is requested, temporarily disable contract functionalities, and finally issue a last notification and execute the `Update` operation after the `TimeLock` is completed.

Source Code References

- [OrderBook/OrderBook.Owner.cs#L60](#)

Code quality

Identifier	Category	Risk	State
FOB-17	Codebase Quality	Informative	Open

It has been possible to verify that the code lacks proper organization and structure, which significantly complicates reading and analyzing the code.

This absence of clear structure is a common issue, especially in dynamic and evolving projects like this one. While it is not a vulnerability in itself, addressing this concern is important for improving the code quality and mitigating the potential emergence of new vulnerabilities. Below, we outline specific areas for potential style, quality, and code readability improvements in the audited contracts.

Typo in name

During the security audit, it has been detected that the `CheckAddrVaild` method is not properly written, so it is recommended to modify it to `CheckAddrValid`.

The same scenario can be seen in the `CheckAddrVaild` method with the `vaild` variable, which should be modified by `valid`.

```
private static bool CheckAddrVaild(params UInt160[] addrs)
{
    bool vaild = true;

    foreach (UInt160 addr in addrs)
    {
        vaild = vaild && addr is not null && addr.IsValid;
        if (!vaild)
            break;
    }

    return vaild;
}
```

Typo error

Source References

- *vaild* rather than *valid*:
 - [OrderBook/OrderBook.Extend.cs#L35](#)
- *pirce* rather than *price*:
 - [OrderBook/OrderBook.cs#L115](#)

Wrong comment

Throughout the project, several inaccurate comments have been detected that provide contradictory or completely irrelevant information. An illustrative case is the following, where "Decrease" is mentioned when, in fact, the opposite should be indicated.

```
//reback token
if (order.IsBuy)
{
    //decrease quoteToken
    AccountStorage.Increase(order.Owner, BaseToken, amount * order.Price / PriceDecimalFixed);
}
```

Source References

- [OrderBook/OrderBook.Order.cs#L180](#)
- [OrderBook/OrderBook.Order.cs#L184](#)
- [OrderBook/OrderBook.Order.cs#L340](#)

Open ToDo

Certain to-do comments have been detected that reflect that the code is unfinished. Thus, many changes that have not been audited could introduce new vulnerabilities in the future.

Source References

An example of source references are the following:

- [OrderBook/OrderBook.Order.cs#L258](#)
- [OrderBook/OrderBook.Order.cs#L292](#)

Wrong use of events

Throughout the project, test events inappropriate for a production environment have been identified, which should be excluded from the final code. These events are currently being used as a `console.log`.

```
public static event TestEvent OnTest;
public delegate void TestEvent(BigInteger b0, BigInteger b1);

public static event TestEvent2 OnTest2;
public delegate void TestEvent2(BigInteger b0, BigInteger b1, BigInteger b2, BigInteger b3);

if (column == PriceBitLength - 1)
{
    OnTest(10089, priceAtSecondNode);
    amountUseAtSecondNode = Max(amount - amountAtPriceFirstNode, 0);
    amountUseAtSecondNode = Min(amountUseAtSecondNode, amountAtPriceSecondNode);
    worth1 = amountUseAtSecondNode * secondNode.BuyWorth / amountAtPriceSecondNode;
}
```

Source References

- [OrderBook/OrderBook.cs#L1114](#)

Wrong reuse of code

It has been identified that throughout the contract parts of code have been reused without being correctly modified. An example of this is the reuse of the key storage between `MakerFeeStorage` and `TakerFeeStorage`.

```
public static class MakerFeeStorage
{
    private static readonly byte[] makerFeePrefix = new byte[] { 0x01, 0x14 };

    internal static void Put(BigInteger feeRate)
    {
        StorageMap map = new(Storage.CurrentContext, makerFeePrefix);
        map.Put("maker", feeRate);
    }

    internal static BigInteger Get()
    {
        StorageMap map = new(Storage.CurrentReadOnlyContext, makerFeePrefix);
        return (BigInteger)map.Get("maker");
    }
}

public static class TakerFeeStorage
{
    private static readonly byte[] takerFeePrefix = new byte[] { 0x01, 0x15 };

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

Source References

- [OrderBook/OrderBook.Storage.cs#L368](#)
- [OrderBook/OrderBook.Storage.cs#L374](#)

Another example of this issue is the failure to modify messages. In this case, it incorrectly refers to `baseToken` when it should be `QuoteToken`.

- [OrderBook/OrderBook.Storage.cs#L147](#)
- [OrderBook/OrderBook.Storage.cs#L155](#)

GAS optimization

Identifier	Category	Risk	State
FOB-18	Codebase Quality	Informative	Open

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

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

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

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

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, something that is not necessary.

Source References

- [OrderBook/OrderBook.Storage.cs#L439](#)
- [OrderBook/OrderBook.cs#L1133-L1135](#)
- [OrderBook/OrderBook.Extend.cs#L69](#)
- [OrderBook/OrderBook.Extend.cs#L135](#)
- [OrderBook/OrderBook.Extend.cs#L149](#)

Logic optimization

It has been detected that it is necessary to optimize the `CheckAddrValid` method. Therefore, by optimizing this function, the cost of GAS in each transaction will be lower, saving the users costs in GAS.

It is important to note that `IsValid` method of the `UInt160` already checks if the value is null. The logic of the `CheckAddrValid` function is not optimal and can be optimized as indicated below:

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

Source References

- [OrderBook/OrderBook.Extend.cs#L35](#)

Furthermore, redundant methods with duplicated logic have been identified. For instance, the `GetBaseToken` and `GetQuoteToken` functions exhibit identical logic to the `BaseToken` and `QuoteToken` methods, making them entirely unnecessary.

```
[Safe]
public static UInt160 GetBaseToken()
{
    return BaseTokenStorage.Get();
}

[Safe]
public static UInt160 GetQuoteToken()
{
    return QuoteTokenStorage.Get();
}

public static UInt160 BaseToken => BaseTokenStorage.Get();
public static UInt160 QuoteToken => QuoteTokenStorage.Get();
```

Source References

- [OrderBook/OrderBook.cs#L69-L82](#)
- [OrderBook/OrderBook.Storage.cs#L84](#)

Different methods have been identified that could be optimized to avoid unnecessary variable declaration, as they consistently return `true`. Therefore, it is advisable to refactor the logic to directly return `true`.

```
[Safe]
public static bool CanClaim(BigInteger orderId)
{
    Order order = OrderStorage.Get(orderId);
    bool canClaim = false;
    for (var column = 0; column < PriceBitLength; column++)
    {
        BigInteger columnPriceIndex = GetColumnPriceIndex(order.Price, column, PriceBitLength);
        bool iszero = NodeRoundStorage.Get(new NodeRoundKey() { ColumnPriceIndex = columnPriceIndex, IsBuy = order.IsBuy, Round = order.Round });
        if (iszero)
        {
            canClaim = true;
            break;
        }
    }
    return canClaim;
}
```

Diagram illustrating the logic flow: The `canClaim = true;` and `break;` statement is highlighted in a red box, with an arrow pointing to the `return true;` statement. Another red box highlights the `return canClaim;` statement at the end of the function.

Unnecessary logic

Source References

- [OrderBook/OrderBook.Order.cs#L57](#)
- [OrderBook/OrderBook.Order.cs#L83](#)
- [OrderBook/OrderBook.Order.cs#L108-L109](#)

Likewise, the `ChangeOwner` and `ApprovedTransfer` methods always return *true*, so this logic is unnecessary and can be removed, consequently saving GAS.

- [OrderBook/OrderBook.Owner.cs#L51](#)
- [OrderBook/OrderBook.cs#L163](#)

The `CanClaim` and `IsZero` methods can be improved in several aspects. First of all, it is advisable to initialize the class outside the loop, avoiding initializing it for each iteration. In addition, it is advisable to use a different prefix for buy and sell. Applying these recommendations, these methods will consume less GAS and will be less prone to errors.

However, it is advisable to review the rest of the methods, as they may also be affected by this logic and therefore could be optimized.

Source References

- [OrderBook/OrderBook.Order.cs#L54](#)
- [OrderBook/OrderBook.Order.cs#L80](#)

Inefficient assertion usage

Identifier	Category	Risk	State
FOB-19	Codebase Quality	Informative	Open

An inefficient programming practice related to the handling of Asserts was identified. Currently, the contract implements its own Assert method to verify conditions and raise exceptions if they are not met. However, this practice is not optimal in terms of efficiency and code clarity. Instead of using custom methods, we recommend using the Neo platform's `ExecutionEngine.Assert` native method, which is already optimized for these operations.

Using the native `ExecutionEngine.Assert` method, which also allows you to include descriptive messages, is a more appropriate practice and aligned with development conventions in Neo. This native method is optimized for assertions and provides greater clarity in error handling.

Recommendations

- Replace the custom Assert methods with `ExecutionEngine.Assert` native method on all instances within the contract. This will simplify the code and take advantage of the optimizations and capabilities of the Neo platform.
- Use the ability to include descriptive messages in the `ExecutionEngine.Assert` method to provide clear contexts and facilitate debugging in case of assertion failures.

Source Code References

- [OrderBook/OrderBook.Extend.cs#L26](#)
- [OrderBook/OrderBook.Extend.cs#L134](#)
- [OrderBook/OrderBook.Extend.cs#L170](#)

Optimize storage prefixes

Identifier	Category	Risk	State
FOB-20	Codebase Quality	Informative	Open

Different `Storage.CurrentContext` prefixes are used throughout the contracts of the project, the only objective of the prefixes is to separate the different sections of the storage. Therefore, it is convenient to use the same length between prefixes and they should be as optimal and as small as possible, since when the prefixes of our smart contract are only of 1 byte it becomes more efficient, and the parsing becomes easier as well.

Recommendations

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.

Source Code References

- [OrderBook/OrderBook.Storage.cs#L14](#)
- [OrderBook/OrderBook.Storage.cs#L27](#)

Safe storage access

Identifier	Category	Risk	State
FOB-21	Design Weaknesses	Informative	Open

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

References

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

Source Code References

- [OrderBook/OrderBook.Storage.cs#L20](#)

Bad coding practices

Identifier	Category	Risk	State
FOB-22	Codebase Quality	Informative	Open

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

Visibility improvement

There are classes exposed which are intended solely for the internal use of the contract. These classes do not need to be publicly exposed. Making them `internal` reduces the complexity of interpreting the contract, in addition to reducing the cost of deploying.

Source References

- [OrderBook/OrderBook.Storage.cs](#)

Storage classes improvement

The classes created in `OrderBook.Storage` are intended to control the different storages of the contract through different methods. However, the `GetObject` and `PutObject` statements are not being used, which are recommended for dealing with object serialization.

Source References

- [OrderBook/OrderBook.Storage.cs#L102-L08](#)
- [OrderBook/OrderBook.Storage.cs#L198-L204](#)
- [OrderBook/OrderBook.Storage.cs#L293-L301](#)
- [OrderBook/OrderBook.Storage.cs#L391-L397](#)

Optimize validation in storage operations

Identifier	Category	Risk	State
FOB-23	Design Weaknesses	Informative	Open

The examination of the code provided for the **OrderBook** contract uncovers a suboptimal practice in managing warehouse operations, particularly concerning data validation. Presently, consistency and validation checks on stored data are conducted during the read (Get) operation rather than during the write (Put).

This approach can result in inefficient use of blockchain resources, escalating gas costs for users during data retrieval, and introducing unnecessary additional load on the system.

This implementation leads to a greater computational load and incurs higher costs each time information is retrieved, thereby impacting the effectiveness and economy of contract operations. The recommended best practice is to conduct all necessary validation during write time (Put), thereby guaranteeing that only valid data is stored. This simplifies and reduces the cost of read (Get) operations, as they can assume that all retrieved data is consistent and valid.

Recommendations

- Modify the Put method to include data validations before storage.
- Redesign the Get method to remove validations. This reduces the complexity and costs associated with read operations.

Source Code References

- [OrderBook/OrderBook.Storage.cs#L43](#)

Incomplete manifest

Identifier	Category	Risk	State
FOB-24	Codebase Quality	Informative	Open

It has been detected that the **OrderBook** contract does not properly specify the information related to the Smart Contract in its manifest. All the contract information identified in the contract's manifest (*author, email, description, source*) belongs to the developer of the project. Therefore, it is recommended to customize said content by adding your own information, which will also provide the users with relevant information about the project, increase the trust and contributes to the improved SEO of the contract.

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

In the case of the audited smart contracts, it has been identified that the `source` attribute is not being included. This value can be defined using the `ContractSourceCodeAttribute` attribute.

Recommendations

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

References

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

Source Code References

- [OrderBook/OrderBook.cs](#)

Use list instead of an array

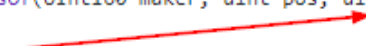
Identifier	Category	Risk	State
FOB-25	Testing and Documentation	Informative	Open

The `GetOrdersOf` method returns *null* elements if there are not enough entries in the storage to reach the `results` array.

When creating an array using user input (*n*), if there are not enough storage entries, the rest of the entries will be returned as *null*.

```
public static Order[] GetOrdersOf(UInt160 maker, uint pos, uint n)
{
    var results = new Order[n];

    var iterator = UserOrderStorage.Find(maker);
    for (int i = 0; i < pos + n; i++)
    {
        if (iterator.Next() && i >= pos)
        {
            results[i - pos] = (Order)iterator.Value;
        }
    }
    return results;
}
```



Size set by user input

Recommendations

- Use `Neo.SmartContract.FrameworkList` instead of array.

References

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

Source Code References

- [OrderBook/OrderBook.Order.cs#L22](#)

Outdated framework

Identifier	Category	Risk	State
FOB-26	Outdated Software	Informative	Open

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

It has been verified that the project uses Neo's framework 3.6.0. This version can be updated containing major improvements. The compiler version can be updated to 3.6.2, which contains major improvements.

Recommendations

- Use the latest version of the compiler and the SDK.

References

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

Source Code References

- [OrderBook/OrderBook.csproj#L10](#)

Lack of safe method attribute

Identifier	Category	Risk	State
FOB-27	Design Weaknesses	Informative	Open

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

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

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

Recommendations

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

References

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

Source Code References

- [OrderBook/OrderBook.cs#L81-L85](#)
- [OrderBook/OrderBook.cs#L711](#)
- [OrderBook/OrderBook.cs#L727](#)
- [OrderBook/OrderBook.Extend.cs#L80](#)
- [OrderBook/OrderBook.Extend.cs#L90](#)
- [OrderBook/OrderBook.Extend.cs#L95](#)
- [OrderBook/OrderBook.Extend.cs#L100](#)
- [OrderBook/OrderBook.Extend.cs#L105](#)
- [OrderBook/OrderBook.Extend.cs#L110](#)
- [OrderBook/OrderBook.Extend.cs#L115](#)
- [OrderBook/OrderBook.Extend.cs#L120](#)

Information Disclosure in Repository

Identifier	Category	Risk	State
FOB-28	Data Exposure	Informative	Open

On the macOS operating system, `.DS_Store` is a file that stores custom attributes of its containing folder, such as icon position, window size, or the choice of a background image, but also lists all the files that have been created in the directory.

Following, it can be observed how through a `DS_Store` file, it is possible to obtain information related to the directory. For example, in this case, the folders it contains.

```
3\ -\ Contracts\OrderBook\.DS_Store
Count:  2
OrderBook
OrderBookSupport
```

This information could provide an attacker sensitive project information including access to hidden files or paths, among others.

Recommendations

- It is recommended to erase any `.DS_Store` file stored in the project.
- Additionally, it is recommended to improve the `.gitignore` file in order to prevent uploading unwanted files, keeping the repositories and the work environment as safe as possible.

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



Invest in Security, invest in your future