RED4SEC

Smart Contracts Security Audit

Nested Finance

2021-11-15

1. Introduction

The Nested Finance platform allows users to put several ERC-20 and BEP-20 tokens into a unique non-fungible token (NFT) known as NestedNFT. At the end of the creation process, the user receives a NestedNFT that encrypts every detail of their portfolio.

Each NestedNFT issued on Nested Finance is backed by the underlying assets, and their real market value. Meanwhile, these underlying assets continue to be traded directly on decentralized exchanges, while managed through self-custodian smart contracts.

As requested by Nested Finance and as part of the vulnerability review and management process, Red4Sec has been asked to perform a security code audit in order to evaluate the security of the Nested Finance smart contracts.

2. 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.

3. Scope

The scope of this evaluation includes the smart contracts found in the following repository: https://github.com/NestedFinance/nested-core-lego, commit `864841f8f1c711d5d47b0734309764a0e7e8b753` and the subsequent reviews up to the following commit `aa7dc7c82953735401817532d2e1b19a4160982b`.

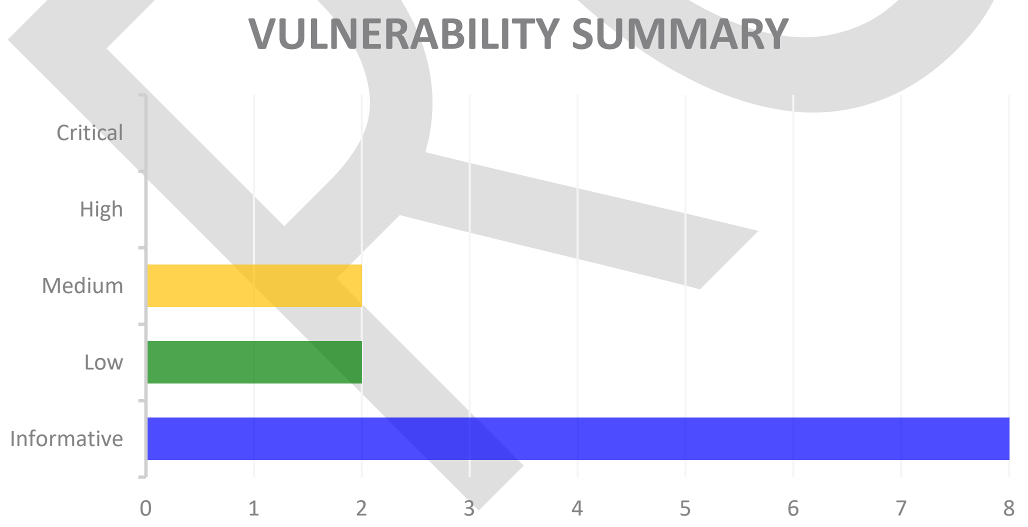
4. Conclusions

To this date, 15th of November 2021, the general conclusion resulting from the conducted audit is that **Nested Finance's smart contracts are secure** and do not present vulnerabilities that could compromise the security of the users and the project integrity.

The overall impression about code quality is very positive. Only a few issues have been detected in the contracts during the security audit that could have affected their proper operation. However, the Nested Finance team has successfully and immediately corrected them.

Low impact issues were detected and classified only as informative, but they will continue to help Nested Finance to improve and optimize the quality of the project.

Found vulnerabilities have been classified in the following levels of risk according to the impact level defined by CVSS v3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST).



Below we have a complete list of the issues 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**
NFSC01	Lack of Remove Logic	Medium	Fixed
NFSC02	Incorrect TokenID Clone	Medium	Fixed
NFSC03	Logic Inconsistency	Low	Fixed
NFSC04	Lack of Inputs Validation	Low	Fixed
NFSC05	Contracts Management Risks	Informative	Assumed
NFSC06	Missing Event	Informative	Assumed
NFSC07	Lack of Event Index	Informative	Assumed
NFSC08	Unsafe External Call	Informative	Discarded
NFSC09	Use of Context Class	Informative	Fixed
NFSC10	Outdated Compiler Version	Informative	Fixed
NFSC11	GAS Optimization	Informative	Fixed
NFSC12	Insecure Transfers	Informative	Fixed

5. Issues and Recommendations

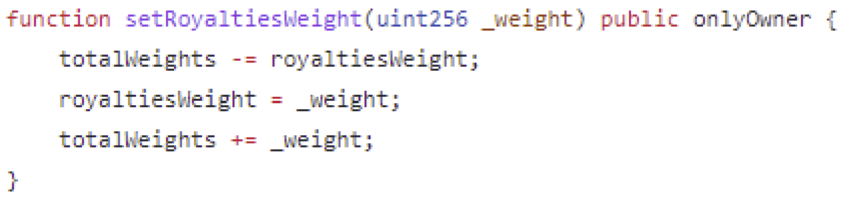
NFSC01 – Lack of Remove Logic (Medium)

Once a contract is deployed, it remains unchanged unless it is updated, a process that must always be avoided. For this reason, it is an important task to study all the required functionalities at the beginning of the development. In this case, a method to eliminate a *Factory* is missing, the *owner* has the ability to allow an address such as *Factory*, but it has no way of reversing this process if needed. So, we consider that there should be a method available that allows us to reverse this process, in case a factory becomes a malicious factory, or to reverse a human error when setting up a contract with errors.

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedAsset.sol
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedRecords.sol

Additionally, no event is emitted when calling to the *setRoyaltiesWeight* method of the **FeeSplitter** contract and this makes a series of changes that directly affect the econometrics of the project, so this change should be reported so that dApps or users can detect it and react accordingly.



```
function setRoyaltiesWeight(uint256 _weight) public onlyOwner {
    totalWeights -= royaltiesWeight;
    royaltiesWeight = _weight;
    totalWeights += _weight;
}
```

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/FeeSplitter.sol#L96

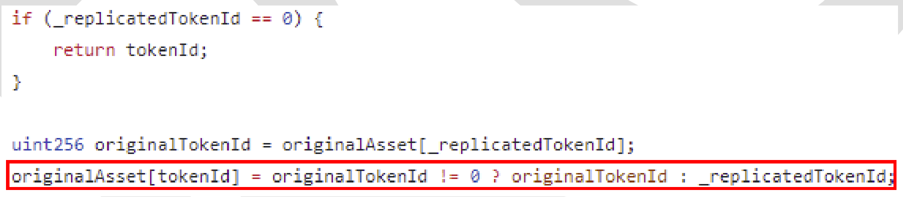
Remediation

The Nested Finance team has corrected the issue in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8 by adding the *removeFactory* function to the **NestedAsset** contract.

Nevertheless, in the https://github.com/NestedFinance/nested-core-lego/pull/15/files#diff-ca52eca62f0d120307e0f0c69576bc8028e50c1200957d4ff44597bc72d0dab3R79-R87 pull request, amongst other changes, a functionality has been added to remove operators, bringing a problem with the cache of the **MixinOperatorResolver** contract, since it requires to previously invoke the *importOperators* function of the **OperatorResolver** with the *address(0)* to ensure that the cache eliminates the previous registries.
This problem has been informed to the Nested Finance team and they consider that the modus operandi of deleting operators will always include this call, so it was discarded after further discussion.

NFSC02 – Incorrect TokenID Clone (Medium)

The *mint* method of the **NestedAsset** contract does not check that the *tokenId* that will be replicated actually exists, which allows creating an NFT by specifying a non-existent ID to replicate. If afterwards another NFT with that ID is minted, a relationship is created between them that has nothing to do with reality.



```
if (_replicatedTokenId == 0) {
    return tokenId;
}

uint256 originalTokenId = originalAsset[_replicatedTokenId];
originalAsset[tokenId] = originalTokenId != 0 ? originalTokenId : _replicatedTokenId;
```

This may affect the royalty distribution if a new Token has subsequently been created with the ID that was assigned when the first token was created.

Source reference

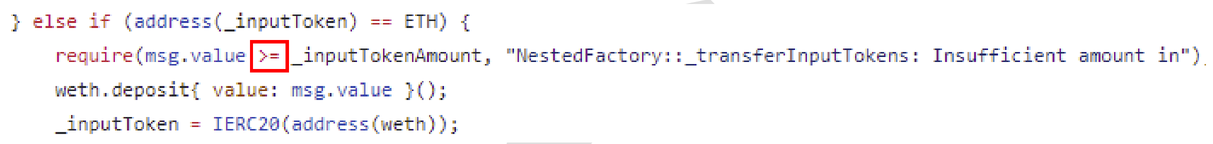
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedAsset.sol#L80
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedFactory.sol#L459

Remediation

The Nested Finance team has corrected the issue in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/5

NFSC03 – Logic Inconsistency (Low)

The *_transferInputTokens* function of the **NestedFactory** contract allows to send more ETH than required but it never allows to send more tokens than requested, this discrepancy in behaviours, in terms of different tokens, can cause confusion or allow a user to lose funds due to human error. So, it is convenient that the mentioned verification is of type ==, to ensure the same behaviour or on the counterpart to ensure that the execution fails.



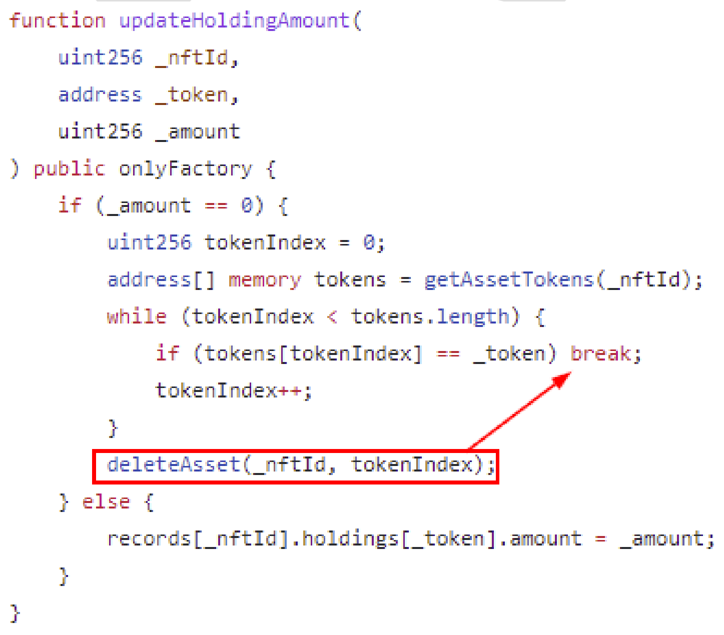
```
} else if (address(_inputToken) == ETH) {
    require(msg.value >= _inputTokenAmount, "NestedFactory::_transferInputTokens: Insufficient amount in");
    weth.deposit{ value: msg.value }();
    _inputToken = IERC20(address(weth));
```

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedFactory.sol#L446

There are also different behaviors in the *updateHoldingAmount* method of the **NestedRecords** contract, influenced by the path that the execution takes, for a non-existent *_nftId* if the *_amount* variable is different from 0, no failure will occur; however, if it is *0*, an error will occur during the execution of the *deleteAsset* method.

It is convenient to call *deleteAsset* within the *break* conditional, and to perform an existing verification of the NFT at the beginning of the method.



```
function updateHoldingAmount(
    uint256 _nftId,
    address _token,
    uint256 _amount
) public onlyFactory {
    if (_amount == 0) {
        uint256 tokenIndex = 0;
        address[] memory tokens = getAssetTokens(_nftId);
        while (tokenIndex < tokens.length) {
            if (tokens[tokenIndex] == _token) break;
            tokenIndex++;
        }
        deleteAsset(_nftId, tokenIndex);
    } else {
        records[_nftId].holdings[_token].amount = _amount;
    }
}
```

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedRecords.sol#L98

Remediation

The Nested Finance team has corrected the issue in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8

NFSC04 – Lack of Inputs Validation (Low)

Some methods of the different contracts in the **NestedFinance** project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

- In the **FeeSplitter** smart contract the *updateShareholder* method does not verify that *_accountIndex* value received by the arguments previously exists.

  Source References

  - https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/FeeSplitter.sol#L168

  Remediation

  This issue has been corrected the issue in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8

- In the **NestedReserve** contract, the *updateFactory* method does not verify that the address sent through the *_newFactory* argument is valid, so it allows the address to be established to *address(0)*.

  Additionally, since the address provided will be the new *factory* it is recommended to adjust the factory's modification logic, to a logic that allows to verify that the new factory is in fact valid and does exist, for example where a new factory is proposed first, the factory accepts the proposal, and we make sure that there are no errors when writing the address of the new factory.

Source References

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedReserve.sol#L64

Remediation

This issue has been corrected the issue in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8; even though the verification is not made in the constructor of the **NestedReserve** contract, the Red4Sec team does not consider this to be a problem.

- In the **NestedRecords** smart contract the *updateLockTimestamp*, *freeToken* and *setReserve* methods do not verify that the *_nftId* received by the arguments previously exists.
  This is also present in the **NestedFactory** contract, in the *isUnlocked* modifier*.*

  Source References

  - https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedRecords.sol#L157
  - https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedRecords.sol#L198
  - https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedRecords.sol#L205
  - https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedFactory.sol#L74-L76

  Remediation

  The Nested Finance team considers that applying measures to solve this issue is unnecessary due to the flow of execution of the contracts.

- In some contracts of the **NestedFinance** project, the address sent through the arguments is not checked to be valid, so it allows the address to be established to *address(0).*

Source References

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/operators/Synthetix/SynthetixStorage.sol#L17
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/operators/ZeroEx/ZeroExStorage.sol#L16

Remediation

The Nested Finance team considers that applying measures to solve this issue is unnecessary due to the flow of execution of the contracts.

- During the logic of the constructor in the contracts, none of the received arguments are verified; for example, in the case of the **NestedRecords** contract, it is important to check that _*maxHoldingsCount* is greater than 0, since it is this same verification which is performed when modifying said value through the *setMaxHoldingsCount* method.

Source References

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/FeeSplitter.sol#L61-L71
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedBuybacker.sol#L45-L56
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedFactory.sol#L47-L62
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/MixinOperatorResolver.sol#L20-L22
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedRecords.sol#L52-L54
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedReserve.sol#L19-L21

Remediation

The Nested Finance team considers that applying measures to solve this issue is unnecessary due to the flow of execution of the contracts.

NFSC05 – Contracts Managment Risks (Informative)

The logic design of the **NestedFinance** contracts imply a few minor risks that should be reviewed and considered for their improvement.

Decentralization Recommendation

In order to promote decentralization, it would be advisable to improve the logic of the **NestedFinance** contracts.

The **NestedFinance** team maintains some centralized parts that imply trust in the project, which are indeed necessary. A few of the administrative functionalities are under the control of the project.

There is the possibility of altering the value of the fee that will be burned or it will be allocated in the reserve by modifying its value in the current block through the *setBurnPart* method of the **NestedBuybacker** contract, being able to reach 100% of the amount, which would allow the owner to perform a front running attack, so it is recommended to use the **TimeLock** contract to avoid this hypothetical scenario.

Operators Risk

The functioning of the Nested Finance project includes the use of operators with the ability to be added in the future at any time.

These operators run in the context of **NestedFactory** so we must highlight that the security of these operators is crucial since they will have the ability to manage the reserve and withdraw the funds. Additionally, the operator of the smart contract and the owner of the contract have the ability to alter *storageAddress* and redirect calls between contracts maliciously to drain the funds.

Composite assets risks

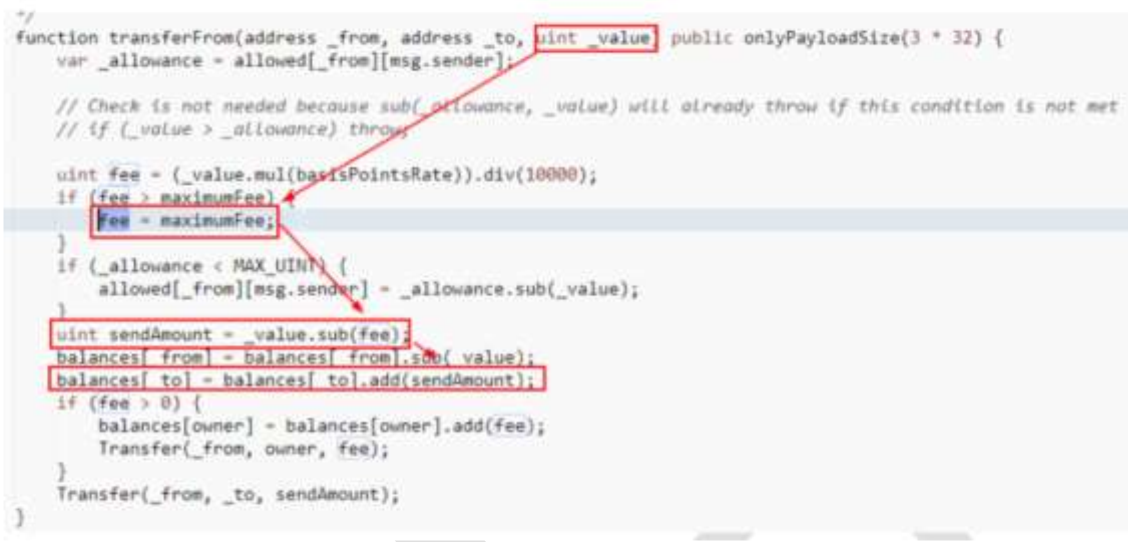
The project makes use of external platforms through the Operators which could lead to manipulations among the operations of portfolio creation, adding liquidity, withdrawal of liquidity, portfolio destruction, ... in Nested Finance. Although it is not currently at risk, it is advisable to add some protections so these operations cannot be carried out in a chained manner in the same transaction, reducing possible exploitation vectors with flash loans and/or altering the price on the platforms of the assets that make up the different NestedNFT portfolios.

Deflationary tokens

As the **NestedFinance** team itself explains in the section "*Known issues:*"

> *Deflationary tokens: Tokens with elastic supply and tokens which burn or transfer a part of each transaction are not supported by the protocol, as they can't be tracked over time by the NestedReserve and NestedRecords contract. -> Those tokens are blacklisted in the frontend platform to protect users.*

Some tokens may implement a fee during transfers, this is the case of USDT even though the project has currently set it to 0. Therefore, the *transferFrom* function would return 'true' despite receiving less than expected.



```
*/
function transferFrom(address _from, address _to, uint _value) public onlyPayloadSize(3 * 32) {
    var _allowance = allowed[_from][msg.sender];

    // Check is not needed because sub(_allowance, _value) will already throw if this condition is not met
    // if (_value > _allowance) throw;

    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    if (_allowance < MAX_UINT) {
        allowed[_from][msg.sender] = _allowance.sub(_value);
    }
    uint sendAmount = _value.sub(fee);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(_from, owner, fee);
    }
    Transfer(_from, _to, sendAmount);
}
```

Therefore, it is considered convenient to use a whitelist instead of a blacklist, as it is a more conservative measure, it is considered safer and more resilient.

NFSC06 – Missing Event (Informative)

No event is emitted when calling to the *setRoyaltiesWeight* method of the **FeeSplitter** contract and this makes a series of changes that directly affect the econometrics of the project, so this change should be reported so that dApps or users can detect it and react accordingly.

```
function setRoyaltiesWeight(uint256 _weight) public onlyOwner {
    totalWeights -= royaltiesWeight;
    royaltiesWeight = _weight;
    totalWeights += _weight;
}
```

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/FeeSplitter.sol#L96

NFSC07 – Lack of Event Index (Informative)

Event indexing of smart contracts can be used to filter during the querying of events. This can be very useful when making dApps or in the off-chain processing of the events in our contract, as it allows filtering by specific addresses, making it much easier for developers to query the results of the invocations.

It could be convenient to review the **FeeSplitter** contract to ensure that all the events have the necessary indexes for the correct functioning of the possible DApps. Addresses are usually the best argument to filter an event.

```
/// @dev Emitted when a payment is released
/// @param to The address receiving the payment
/// @param token The token transfered
/// @param amount The amount paid
event PaymentReleased(address to, address token, uint256 amount);

/// @dev Emitted when a payment is released
/// @param from The address sending the tokens
/// @param token The token received
/// @param amount The amount received
event PaymentReceived(address from, address token, uint256 amount);
```

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/FeeSplitter.sol#L24
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/FeeSplitter.sol#L30

NFSC08 – Unsafe External Call (Informative)

The **OperatorResolver** contract has an external function named *rebuildCaches*, that makes calls to destination contracts without verifying the sender or the destination of said contracts, this in certain circumstances could result insecure and it allows to call to contracts that implement a *rebuildCache* or *fallback* method, both external contracts and from the platform itself (inherited from *MixinOperatorResolver*).
It is recommended to limit the interaction with external contracts only to authorized *operators* that are defined in the contract itself.

Source reference

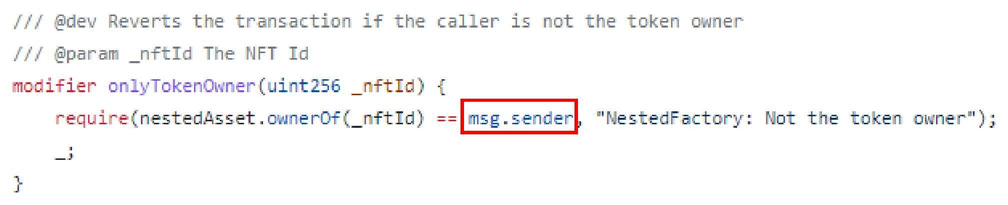
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/OperatorResolver.sol#L57
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/OperatorResolver.sol#L12

Remediation

The Nested Finance team understands that this issue is a risk worth evaluating but is currently not exploitable and the current operators are managed and approved by the project. Therefore, no measure is considered to be applied in the near future.

NFSC09 – Use of Context Class (Informative)

The **NestedFinance** project inherits a functionality of the *Context* contract of OpenZeppelin, which is designed to be used with Ethereum Gas Station Network (GSN[1]), but it also contains references to msg.sender.



```
/// @dev Reverts the transaction if the caller is not the token owner
/// @param _nftId The NFT Id
modifier onlyTokenOwner(uint256 _nftId) {
    require(nestedAsset.ownerOf(_nftId) == msg.sender, "NestedFactory: Not the token owner");
    _;
}
```

[1] https://docs.opengsn.org/

It is advisable to review that these functionalities are being used and to replace *msg.sender* with *_msgSender()* in every occurrence. If this is not the case, remove the *Context* functionality.

Remediation

This issue has been corrected in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8

NFSC10 – Outdated Compiler Version (Informative)

Solc frequently launches new versions of the compiler. 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.

We have detected that the audited contract uses the following version of Solidity pragma 0.8.4:

It is always a good policy to use the most up to date version of the pragma.

Solidity branch *0.8.9* has important bug fixes in the immutable state, so it is recommended to use the most up to date version of the pragma.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md

Remediation

This issue has been corrected in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8

NFSC11 – GAS Optimization (Informative)

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 Ethereum blockchain, GAS is an execution fee which is used to compensate miners 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.
- Explicit function visibility.
- 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.

Unused 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 unnecessary executions.

The *combineArrays* internal method of the **MixinOperatorResolver** contract is not used throughout the code, as is the case of the *_getRevertMsg* method located in the **ExchangeHelpers** contract.

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/MixinOperatorResolver.sol#L56
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/libraries/ExchangeHelpers.sol#L43

Additionally, it has been possible to verify that the *_token* argument of the *withdraw* method in the **NestedFactory** contract is not necessary, since its value must be the same as the one stored in *nestedRecords*, so it becomes an unnecessary argument that generates an extra cost of GAS.

Source reference

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedFactory.sol#L258

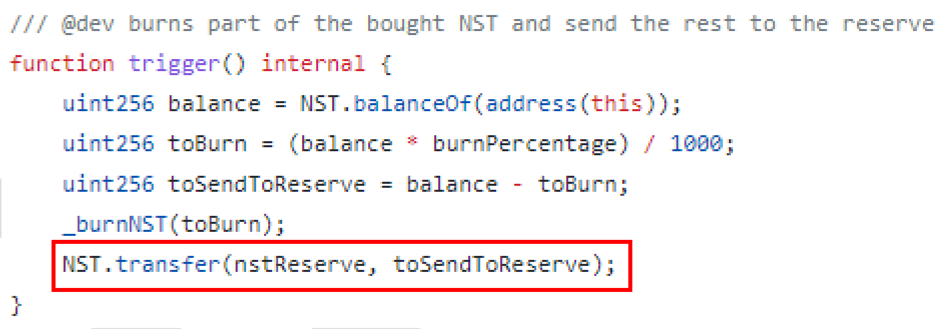
Remediation

This issue has been corrected in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8

NFSC12 – Insecure Transfers (Informative)

The ERC-20[2] standard specifies that the *transfer* and *transferFrom* functions will return a boolean with the result of this operation.

The **NestedBuybacker** contract does not contemplate this result, although it is true that most of the tokens ERC-20 implementations make a revert if these methods fail, the result of external contract calls should always be verified.



```
/// @dev burns part of the bought NST and send the rest to the reserve
function trigger() internal {
    uint256 balance = NST.balanceOf(address(this));
    uint256 toBurn = (balance * burnPercentage) / 1000;
    uint256 toSendToReserve = balance - toBurn;
    _burnNST(toBurn);
    NST.transfer(nstReserve, toSendToReserve);
}
```

It is pivotal to check that the returned value is true in all of the transfers, it is also possible to use SafeERC20[3] from OpenZeppelin contracts, which already makes the verifications after the execution of the transfers.

Furthermore, this case is also found in the *triggerForToken* method as it does not check the results of the *approve* and *fillQuote* calls.

Source references

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedBuybacker.sol#L112
- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedBuybacker.sol#L100

[2] https://eips.ethereum.org/EIPS/eip-20
[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/8b58fc71919efda463e53b3ffa083edac19c85b8/contracts/token/ERC20/SafeERC20.sol#L72

- https://github.com/NestedFinance/nested-core-lego/blob/864841f8f1c711d5d47b0734309764a0e7e8b753/contracts/NestedBuybacker.sol#L101

Remediation

This issue has been fixed by modifying the call of the transfer method to the *safeTransfer* method in the following pull request https://github.com/NestedFinance/nested-core-lego/pull/8

RED4SEC

Invest in Security, invest in your future