Security Audit Report

06/02/2023

NeoCompounder

Content

Introduction

NeoCompounder introduces a new token cNEO (CompoundingNeo) that takes the GAS rewards from voting and compounds it into more underlying bNEO, with the intended result being that each cNEO token becomes worth more and more bNEO over time.

Users can mint new cNEO at any time by depositing NEO or bNEO in the cNEO contract and minting the equivalent value of cNEO tokens. They can later burn their cNEO tokens to redeem the underlying bNEO, which aims to be a greater quantity due to the GAS compounding.

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 **NeoCompounder** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of **NeoCompounder**. 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 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 **NeoCompounder** 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/neocompounder/cneo-contract
  - commit: `a90710104f38773ac6945ac13571696e45322bf0`

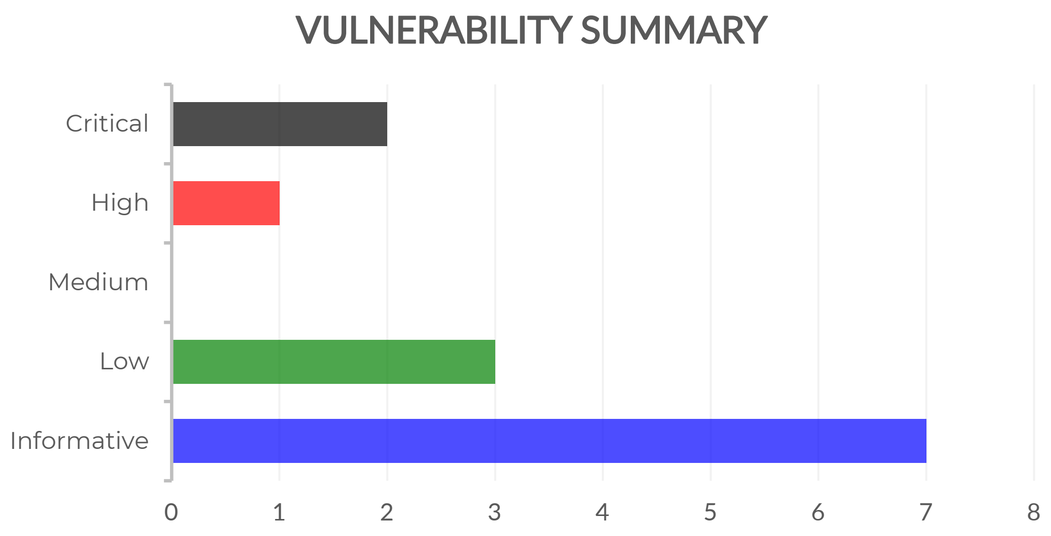
- Remediations review (06/02/2023)
  - commit: `2911e083cccaae7b6e4aceca08aad30f8b8b2dac`

Executive Summary

The security audit against **NeoCompounder** has been conducted between the following dates: **12/01/2023** and **06/02/2023**.

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 **13 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.



Conclusions

To this date, **06/02/2023**, the general conclusion resulting from the conducted audit and after further review of the applied fixes, is that the **NeoCompounder is secure** and do not currently present known vulnerabilities that could compromise the security of the project. Additionally a **few low impact issues** were detected and classified only as informative, but they will continue to help **NeoCompounder** improve the security and quality of its developments.

The general conclusions of the performed audit are:

- Critical and high-risk vulnerabilities were detected during the security audit. These vulnerabilities were of great risk for the **NeoCompounder** project and have been properly corrected by the team.
- The economics of the **NeoCompounder** project have not been audited, therefore there may be risks to the viability and sustainability of the project.
- The contracts have administrative functions that allow full control of the project and obtain an advantageous position for the owner. It is recommended to transfer these permissions to Time Lock contract, Multisignature wallet or even to renounce them, ensuring **the complete decentralization of the project.**
- It is important to mention that **NeoCompunder** uses third-party projects, such as flamingo or bNEO. Therefore, although it is the expected logic, this increases the surface of attack, by being susceptible to any failure in one of these projects and having a direct impact on **NeoCompounder**. Furthermore, this requires **NeoCompounder** users to indirectly trust multiple projects.
- Certain methods **did not make the necessary input checks** in order to guarantee the integrity and expected arguments format.
- The base code is clean; however, it could be improved in certain cases as it lacks organization. These minor defects do not represent a direct threat to the security of the application, but Red4Sec has given some additional recommendations on how to continue improving and how to apply good practices.
- It is important to highlight that both, the new N3 blockchain and the compiler of the smart contracts neow3j are new technologies, which are in constant development and have less than a year functioning. For this reason, they are prone to new bugs and breaking changes. Therefore, this audit is unable to detect any future security concerns with neo smart contracts and the compiler used.

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
NCP-01	Price Manipulation in bNEO purchases	**Critical**	**Fixed**
NCP-02	Missing Scope in Verify	**Critical**	**Fixed**
NCP-03	Wrong Interest Distribution	**High**	**Fixed**
NCP-04	Wrong NEP-17 Implementation	**Low**	**Fixed**
NCP-05	Lack of Inputs Validation	**Low**	**Partially Fixed**
NCP-06	Incomplete Burger Agent Removal	**Low**	**Fixed**
NCP-07	Emit Events on State Changes	**Informative**	**Fixed**
NCP-08	Safe Contract Update/Destroy	**Informative**	**Partially Fixed**
NCP-09	Use of Assert instead of Throw	**Informative**	**Fixed**
NCP-10	GAS Optimization	**Informative**	**Fixed**
NCP-11	Contracts Management Risks	**Informative**	**Partially Fixed**
NCP-12	Lack of Safe Method Attribute	**Informative**	**Fixed**
NCP-13	Bad Coding Practices	**Informative**	**Fixed**

Vulnerability details

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

- Category
- Active
- Risk
- Description
- Recommendations

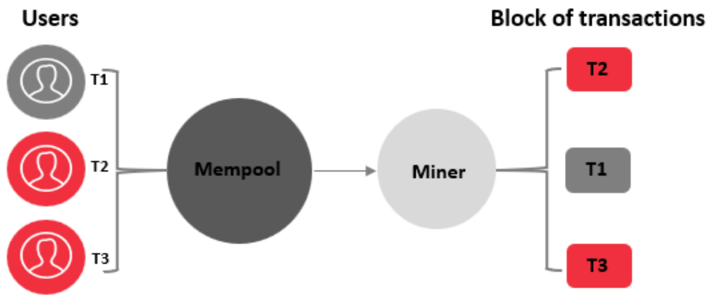
Price Manipulation in bNEO purchases

Identifier	Category	Risk	State
NCP-01	Business Logic	**Critical**	**Fixed**

The `swapGasForBneo` method does not contemplate the slippage in the swaps or the price in the **Flamingo** pools, which allows third parties of ways to obtain economic performance as the public `compund` is invoked.

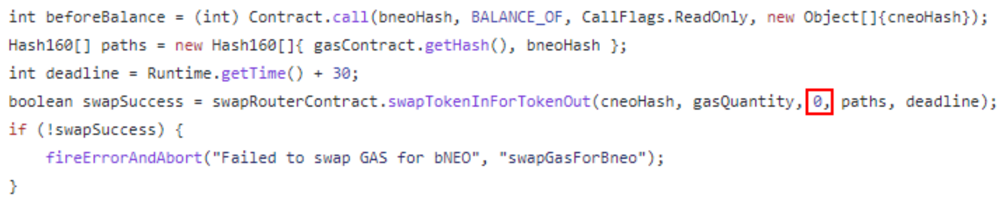
An attacker can use this method to alter the GAS price in the **Flamingo** pool before buying GAS, or he can perform a Sandwich Attack to the `compound` method of the **NeoCompounder**, producing an economic loss on the expected performance of the project.

A Sandwich Attack is conducted with the objective of front-running a cryptocurrency in DeFi projects. A user makes a transaction to trade a cryptocurrency, X asset for a Y asset with the intent of making a large purchase. Immediately a front-running bot detects it and buys the asset before the large trade is executed, front-running the victim. This action sandwiches the transaction of the victim and raises the price of the Y asset for the user, consequently increasing the slippage.



In order to gain profit from this exploit, an attacker would detect a buy transaction (the `compound` invocation) previous to its execution, or invoke it himself, then buy an amount of the same token before the victim's transaction is carried out and wait for the user's transaction to be executed; finally, the attacker would sell all his tokens right after this action and take the profit. Since it is a public method, the attacker will be able to carry out all the steps in the same transaction from a smart contract.

In the `swapGasForBneo` method, the `amountOutMin` variable determines the slippage, therefore, determining the minimum that can be obtained for the swap; however, it is defined to 0.



```
int beforeBalance = (int) Contract.call(bneoHash, BALANCE_OF, CallFlags.ReadOnly, new Object[]{cneoHash});
Hash160[] paths = new Hash160[]{ gasContract.getHash(), bneoHash };
int deadline = Runtime.getTime() + 30;
boolean swapSuccess = swapRouterContract.swapTokenInForTokenOut(cneoHash, gasQuantity, 0, paths, deadline);
if (!swapSuccess) {
    fireErrorAndAbort("Failed to swap GAS for bNEO", "swapGasForBneo");
}
```

This allows, through a sandwich attack, to manipulate a pool in order to unbalance the exchange of gas/bNeo, allowing to enter the `swapGasForBneo` transaction to later withdraw the profit.

Recommendations

- It is necessary to establish a minimum of slippage or limit the use of compound.

References

- FlamingoSwapRouterContract.cs#L330

Source Code References

- CompoundingNeo.java#L840

Fixes Review

This issue has been addressed in the following commits:

- https://github.com/neocompounder/cneo-contract/commit/9d04166f3495bcb7ab4e1999294be557c66bd28b
- https://github.com/neocompounder/cneo-contract/commit/2911e083cccaae7b6e4aceca08aad30f8b8b2dac

Missing Scope in Verify

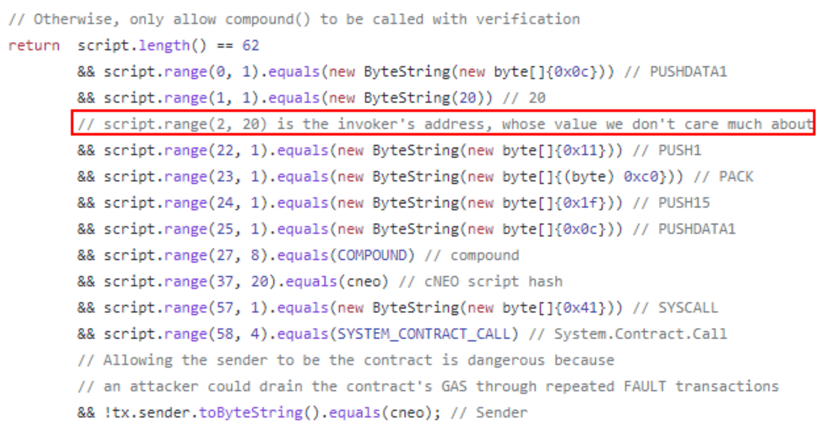
Identifier	Category	Risk	State
NCP-02	Timing and State	**Critical**	**Fixed**

The implementation of the `verify` method in the **CompoundingNeo** does not cover all possible scenarios and it allows an attacker to create an invocation to drain the funds from the smart contract.

The intention of the logic in the `verify` method is to allow the invocation of the `compound` method by third parties on behalf of **CompoundingNeo**, however, since the `compound` method contains an invocation to the **GasToken** contract during its execution, this can be used by an attacker to reuse the witnesses scope in a transfer of assets in the same transaction.

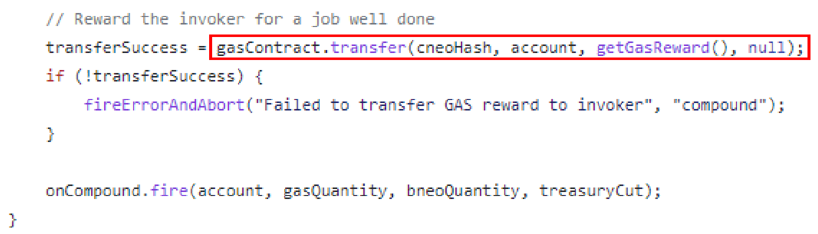
This is possible in N3 because the `NEP17` standard establishes that after sending tokens to a contract, the `onNEP17Payment()` method must be invoked. Therefore, transfers to contracts will invoke the execution of the payment method of the recipient and the recipient may redirect the execution to himself or to another contract.

To exploit this vulnerability, an attacker can construct a transaction where `sender` is **CompoundingNeo** and it calls the `compound` method by setting the `account` argument to the address of a smart contract under the control of the attacker.
This is possible because the argument of the `compound` method is not considered during the verification of the script.



```
// Otherwise, only allow compound() to be called with verification
return  script.length() == 62
        && script.range(0, 1).equals(new ByteString(new byte[]{0x0c})) // PUSHDATA1
        && script.range(1, 1).equals(new ByteString(20)) // 20
        // script.range(2, 20) is the invoker's address, whose value we don't care much about
        && script.range(22, 1).equals(new ByteString(new byte[]{0x11})) // PUSH1
        && script.range(23, 1).equals(new ByteString(new byte[]{(byte) 0xc0})) // PACK
        && script.range(24, 1).equals(new ByteString(new byte[]{0x1f})) // PUSH15
        && script.range(25, 1).equals(new ByteString(new byte[]{0x0c})) // PUSHDATA1
        && script.range(27, 8).equals(COMPOUND) // compound
        && script.range(37, 20).equals(cneo) // cNEO script hash
        && script.range(57, 1).equals(new ByteString(new byte[]{0x41})) // SYSCALL
        && script.range(58, 4).equals(SYSTEM_CONTRACT_CALL) // System.Contract.Call
        // Allowing the sender to be the contract is dangerous because
        // an attacker could drain the contract's GAS through repeated FAULT transactions
        && !tx.sender.toByteString().equals(cneo); // Sender
```

During the execution of this transaction, the `compound` method calls `transfer` of GAS, which will automatically invoke the `onNEP17Payment` method of the malicious contract and it will only have to implement a transfer of assets from **CompoundingNeo** to the attacker.



```
    // Reward the invoker for a job well done
    transferSuccess = gasContract.transfer(cneoHash, account, getGasReward(), null);
    if (!transferSuccess) {
        fireErrorAndAbort("Failed to transfer GAS reward to invoker", "compound");
    }

    onCompound.fire(account, gasQuantity, bneoQuantity, treasuryCut);
}
```

When the contract of the asset requested by the attacker performs the verifications to check the authorization to conduct a `transfer`, it invokes `checkWitness` which, since the `sender` is a smart contract, depends on the result of the `verify` method of the **CompoundingNeo** that returns `true`, so the transfer is successful and it allows the attacker to arbitrarily drain funds from **CompoundingNeo**.

Verifying the scope of the signature is a good security practice to confirm that the signature is valid and that it has been generated for a specific scope, eliminating risks of signature reuse.

Additionally, it is important to check during the `verify` method that the argument used in the `compound` invocations is the **CompoundingNeo** contract itself.

Recommendations

- The scope of the signature must be verified, or the sender should be limited when the contract itself pays for the gas.

References

- https://docs.neo.org/docs/en-us/basic/concept/transaction.html#signature-scope
- https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki

Source Code References

- CompoundingNeo.java#L152-L178
- CompoundingNeo.java#L450

Fixes Review

This issue has been addressed in the following commit:

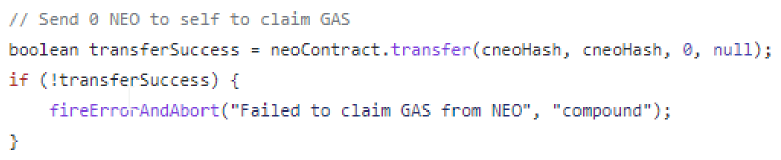
- https://github.com/neocompounder/cneo-contract/commit/d1b3ae34464dbc8fc26510b1edadafaaf543dfd9

Wrong Interest Distribution

Identifier	Category	Risk	State
NCP-03	Business Logic	**High**	**Fixed**

The `compound` method of the `CompoundingNeo` contract aims to convert the gas that it will receive from NEO staking and bNEO into bNEO. However, this logic can be manipulated by external agents in a way that the amount converted is much less than expected, affecting the metrics and economics of the project.

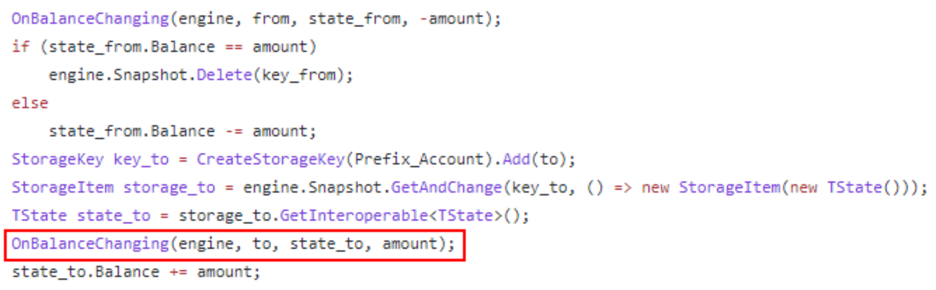
In order to claim the gas from NEO staking, the method makes a transfer of *0* NEO to itself.



```
// Send 0 NEO to self to claim GAS
boolean transferSuccess = neoContract.transfer(cneoHash, cneoHash, 0, null);
if (!transferSuccess) {
    fireErrorAndAbort("Failed to claim GAS from NEO", "compound");
}
```

However, there are other ways to claim the accrued GAS without using this method, which can result in the `compound` method not being able to claim the GAS and convert it to NEO to generate the compound interests.

This is possible because the N3 blockchain not only claims gas for an address when it sends neo, but also when it receives it.



```
OnBalanceChanging(engine, from, state_from, -amount);
if (state_from.Balance == amount)
    engine.Snapshot.Delete(key_from);
else
    state_from.Balance -= amount;
StorageKey key_to = CreateStorageKey(Prefix_Account).Add(to);
StorageItem storage_to = engine.Snapshot.GetAndChange(key_to, () => new StorageItem(new TState()));
TState state_to = storage_to.GetInteroperable<TState>();
OnBalanceChanging(engine, to, state_to, amount);
state_to.Balance += amount;
```

If a user sends *1* NEO before the call to the `compound` method, they will claim the GAS of the contract, which will remain in possession of the contract itself, but without the ability to increase the interests of CNEO. In addition, the attacker will be able to decompose his CNEO in the same transaction without incurring any economic loss for him, beyond the cost of the transaction.

It should be noted that this scenario can occur without the need of an intentional attack, because simply with the action of a user converting their NEO to CNEO before the call to the `compound` method this can take place.

Recommendations

- The `compound` method must consider the non-computed gas, not the difference of balances.

References

- FungibleToken.cs#L189
- NeoToken.cs#L122

Source Code References

- CompoundingNeo.java#L419-L456

Fixes Review

This issue has been partially addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/610a0638a13229e39bc068f6931da900964bf7b1

Wrong NEP-17 Implementation

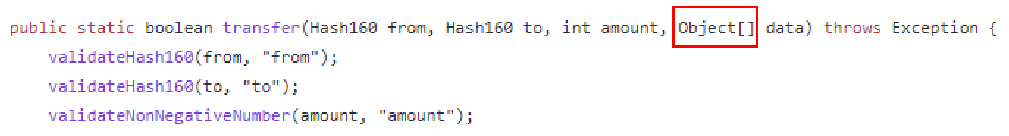
Identifier	Category	Risk	State
NCP-04	Codebase Quality	**Low**	**Fixed**

The `NEP17` standard establishes the methods to be implemented by the project, however the following discrepancies have been found in the audited token.

The standard defines the `transfer` method of `NEP17` tokens must be as displayed in the following image.

```
{
  "name": "transfer",
  "safe": false,
  "parameters": [
    {
      "name": "from",
      "type": "Hash160"
    },
    {
      "name": "to",
      "type": "Hash160"
    },
    {
      "name": "amount",
      "type": "Integer"
    },
    {
      "name": "data",
      "type": "Any"
    }
  ],
  "returntype": "Boolean"
}
```

However, an `Object[]` type has been used for the `data` argument, infringing the standard.



```
public static boolean transfer(Hash160 from, Hash160 to, int amount, Object[] data) throws Exception {
    validateHash160(from, "from");
    validateHash160(to, "to");
    validateNonNegativeNumber(amount, "amount");
```

Recommendations

- Follow the `NEP-17` standard and modify the type of the argument `data` to be only `Object`.

References

- https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki#user-content-transfer

Source Code References

- CompoundingNeo.java#L357

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/ce94a70de6583697b9a1daa13c9c2fb20952f6e2

Lack of Inputs Validation

Identifier	Category	Risk	State
NCP-05	Data Validation	**Low**	**Partially Fixed**

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

In various areas of the code, the inputs and type integrity of some arguments and/or their range are not checked. This bad practice greatly increases the risk of storage injections.

Contract Interface

Certain methods such as `setBneoScriptHash`, `setSwapPairScriptHash`, `setSwapRouterScriptHash` or `setBurgerAgentScriptHash` receive the address of a contract, but only verify that the address is a contract; it is not verified whether or not it has the necessary methods or whether it implements the expected interface, an analogy with the ethereum network could be the use of an interface style EIP-165.

Source Code References

- CompoundingNeo.java#L197
- CompoundingNeo.java#L210
- CompoundingNeo.java#L223
- CompoundingNeo.java#L302

Value Range

The `setMaxSupply` method allows setting the supply to less than the current supply found in the contract, which could produce a discrepancy between the configured values and the existing ones. It would be convenient to check the balance and prevent this from happening.

Source Code References

- CompoundingNeo.java#L275

The `setGasReward` method lacks any valid range to be verified, so the owner can abuse this value to drain the contract of all the generated GAS.

Source Code References

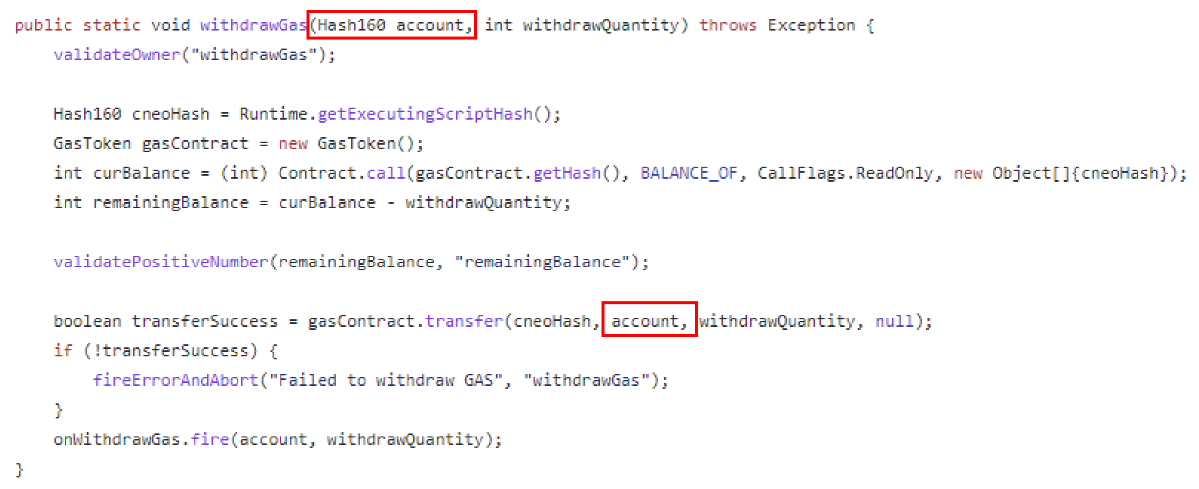
- CompoundingNeo.java#L288

The `setFeePercent` method allows a very wide range that gives the owner the possibility to set a fee of 100% that will allow the owner to capture all the GAS generated.

Source Code References

- CompoundingNeo.java#L258

The `withdrawGas` method does not validate the received `account` argument and delegates this validation to the external contract `GasToken`.



```java
public static void withdrawGas(Hash160 account, int withdrawQuantity) throws Exception {
    validateOwner("withdrawGas");

    Hash160 cneoHash = Runtime.getExecutingScriptHash();
    GasToken gasContract = new GasToken();
    int curBalance = (int) Contract.call(gasContract.getHash(), BALANCE_OF, CallFlags.ReadOnly, new Object[]{cneoHash});
    int remainingBalance = curBalance - withdrawQuantity;

    validatePositiveNumber(remainingBalance, "remainingBalance");

    boolean transferSuccess = gasContract.transfer(cneoHash, account, withdrawQuantity, null);
    if (!transferSuccess) {
        fireErrorAndAbort("Failed to withdraw GAS", "withdrawGas");
    }
    onWithdrawGas.fire(account, withdrawQuantity);
}
```

Source Code References

- CompoundingNeo.java#L396

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.

References

- https://eips.ethereum.org/EIPS/eip-165

Fixes Review

This issue has been partially addressed in the following commit:

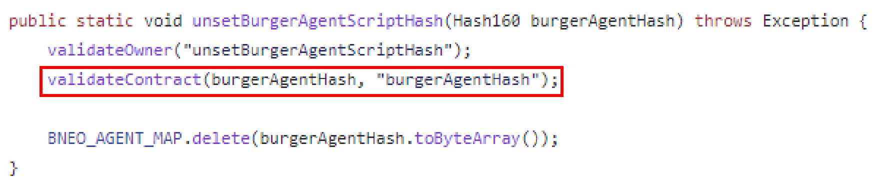
- https://github.com/neocompounder/cneo-contract/commit/30466c6642cd1ccc970519473ef911d1e26d6c0e

Incomplete Burger Agent Removal

Identifier	Category	Risk	State
NCP-06	Data Validation	**Low**	**Fixed**

It is possible to deny the administrative task of removing a Burger Agent, if the owner of the Agent destroys the contract before the `unsetBurgerAgentScriptHash` method is invoked.

During the erase of an agent, the `burgerAgentHash` argument is unnecessarily verified to be a valid contract.



```
public static void unsetBurgerAgentScriptHash(Hash160 burgerAgentHash) throws Exception {
    validateOwner("unsetBurgerAgentScriptHash");
    validateContract(burgerAgentHash, "burgerAgentHash");

    BNEO_AGENT_MAP.delete(burgerAgentHash.toByteArray());
}
```

Therefore, if the Burger Agent contract is destroyed, it will cause the transaction to `unsetBurgerAgentScriptHash` to fail this validation and it cannot be removed from the list of agents. Subsequently, the owner could deploy again the exact same contract, with the same address if he uses the same values, as long as the neo committee votes to unlock said address.

The criticality of this issue has been lowered because at the time of `Destroy`*, the address is locked, and it requires the consensus committee vote to unlock the account.*

Recommendations

- It is convenient to modify the `validateContract` call to `validateHash160` in the `unsetBurgerAgentScriptHash` method.

Source Code References

- CompoundingNeo.java#L309

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/e9d334037da75cfc6bb8367eeab47d77bea3f8e4

Emit Events on State Changes

Identifier	Category	Risk	State
NCP-07	Auditing and Logging	**Informative**	**Fixed**

It is a good practice to emit events when there are significant changes in the states of the contract that can affect the result of its execution by the users.

The changes in the Fees, in the periods, or any important change should emit events so that the potential actors monitoring the blockchain; such as DApps, automated processes and users, can be notified of these significant state changes.

Following find a list of the main methods that make significant changes and should be properly notified.

- `setFeePercent`
- `setGasReward`
- `setMaxSupply`
- `setCompoundPeriod`
- `setOwner`
- `setSwapRouterScriptHash`
- `setSwapPairScriptHash`
- `setBneoScriptHash`
- `setBurgerAgentScriptHash`

Recommendations

- Consider issuing events to notify of changes in the contract values that may affect relationship of the users with the contract.

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/49f4b397b7d66a791d6824f4a2454035b20a798a

Safe Contract Update/Destroy

Identifier	Category	Risk	State
NCP-08	Design Weaknesses	**Informative**	**Partially Fixed**

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

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

Recommendations

- There are various good practices that help mitigate this problem, such as; add a `TimeLock` to start the `Update` or `Destroy` operations, emit events when the `Update` operation is requested, temporarily disable some contract functionalities, and finally execute the `Update` or `Destroy` operation after the `TimeLock` is completed.
- In the case of `destroy`, the contract must verify that there are no funds stored in the contract that could be lost in the destruction of the contract.

Source Code References

- CompoundingNeo.java#L142-L150

Fixes Review

This issue has been partially addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/617cfcebcdbc8257b479a21694f952b5e7acc85c

Use of Assert instead of Throw

Identifier	Category	Risk	State
NCP-09	Design Weaknesses	**Informative**	**Fixed**

The native `ExecutionEngine.Assert` must be used instead of `throw`. The `ASSERT` and `ABORT` opcodes are approximately 9 times less expensive than the `THROW` opcode, and in the case of the `ExecutionEngine.Assert` it also allows to specify an error message.

`Throw` may be captured by the `try/catch` instructions; for this reason, its use must be limited only to some initial verifications in the methods when none of the values have been updated and even in these cases it is better to use `ExecutionEngine.Assert`.

In fact, `throw` it is only recommended to be used within a `try` sentence in the same contract, otherwise it can end up being captured by an external contract, and it will not necessarily end the execution of the transaction.

Recommendations

- `throw` it is only recommended to be used within a `try` sentence in the same contract.

Source Code References

- CompoundingNeo.java#L263
- CompoundingNeo.java#L502
- CompoundingNeo.java#L545
- CompoundingNeo.java#L552
- CompoundingNeo.java#L792
- CompoundingNeo.java#L873
- CompoundingNeo.java#L876
- CompoundingNeo.java#L946
- CompoundingNeo.java#L953
- CompoundingNeo.java#L960
- CompoundingNeo.java#L966
- CompoundingNeo.java#L973
- CompoundingNeo.java#L980

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/ca0c48083d58b0c1852d91fb49f4ae5870676645

GAS Optimization

Identifier	Category	Risk	State
NCP-10	Codebase Quality	**Informative**	**Fixed**

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

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

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

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

Static Variables

The static variables in N3 are processed at the beginning of the execution of the smart contract, so they must be carefully used. Although these static variables are not used for the call that will be made, their initialization will be processed anyway, and they will occupy elements in the stack with the corresponding cost that this entails.
It is a good practice to reduce the static variables, either through constants or through methods that return the desired value.

Source Code References

- CompoundingNeo.java#L47-L87

Logic Optimization

The logic belonging to the `CompoundingNeo` contract can be improved in order to get executions of lower cost.

It is possible to remove the call to the `StringLiteralHelper.stringToInt` method in order to convert the supplied value, since it is within a range that does not conflict with the compiler.

```
private static final int INITIAL_COMPOUND_PERIOD = StringLiteralHelper.stringToInt
("604800000");
```

Another example is the case of the `COMPOUND` constant, where only by casting the *"compound"* value to a bytes string, it is possible to obtain the same result without going through hexadecimal conversions.

```
private static final ByteString COMPOUND = StringLiteralHelper.hexToBytes
("636f6d706f756e64");
```

Source Code References

- CompoundingNeo.java#L60
- CompoundingNeo.java#L86

Useless Event

It is convenient to review the project to ensure that all the events are necessary; for example, the `onError` event from `CompoundingNeo` can be eliminated since the core emits an event when an error occurs. However, according to the code of the core all notifications are removed if an uncaught error occurs in the context, so it is not possible to display said event.

Source Code References

- CompoundingNeo.java#L122

Fixes Review

This issue has been addressed in the following commits:

- https://github.com/neocompounder/cneo-contract/commit/875ecd6b2e68e40eadf0b6ac014f17347479874a
- https://github.com/neocompounder/cneo-contract/commit/71c234b2d93ce9656b084fc83c058c63fd625c0e

Contracts Management Risks

Identifier	Category	Risk	State
NCP-11	Governance	**Informative**	**Partially Fixed**

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

Owner Rights

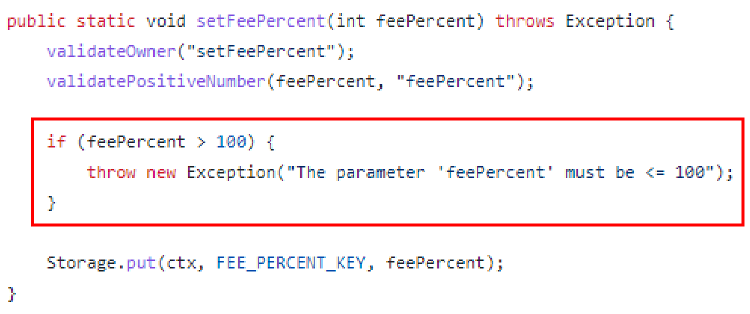
The current implementation gives the Owner full control over the smart contract, on one instance with methods such as `withdrawGas` that allows draining the contract of GAS. However, it is more noteworthy the condition in the `verify` method that allows the Owner to do any action even, if it is not directly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

Source Code References

- CompoundingNeo.java#L156
- CompoundingNeo.java#L397

Uncontrolled fee

The Owner can arbitrarily set the percentage fee, without considering fees of 100%. The current logic allows abusive fees for the users, so it is advisable to verify that the fee can never exceed a realistic maximum.



```
public static void setFeePercent(int feePercent) throws Exception {
    validateOwner("setFeePercent");
    validatePositiveNumber(feePercent, "feePercent");

    if (feePercent > 100) {
        throw new Exception("The parameter 'feePercent' must be <= 100");
    }

    Storage.put(ctx, FEE_PERCENT_KEY, feePercent);
}
```

Source Code References

- CompoundingNeo.java#L262

Dependency on third-party projects

Although we trust the **CompoundingNeo** project, intrinsically we must also trust projects such as Flamingo or bNEO, which the project uses throughout its logic, this increases the necessary surface of trust, and the surface of attacks, since a failure in these two projects could have a direct impact on **CompoundingNeo**.

References

- https://academy.binance.com/en/glossary/rug-pull

Fixes Review

This issue has been partially addressed in the following commit:

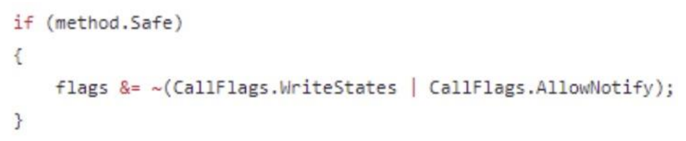
- https://github.com/neocompounder/cneo-contract/commit/26a1491d757afd1dee24c1c48ec5e6c3c61d79c6

The `Verify` method adds new checks to limit the actions allowed by the Owner but the checks do not adequately verify the integrity of the Script, limited to checking specific parts of it. The Owner could build a script that performs other actions and fill in the script so that the checked positions pass the validation.

Lack of Safe Method Attribute

Identifier	Category	Risk	State
NCP-12	Design Weaknesses	**Informative**	**Fixed**

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



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

Additionally, it will provide the wallets and dApps with the necessary information to identify it as a query method and to make a reading invocation with no GAS costs. 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

- CompoundingNeo.java#L153

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/ba64f951eeb76d95e9592dbd5835c887225e7996

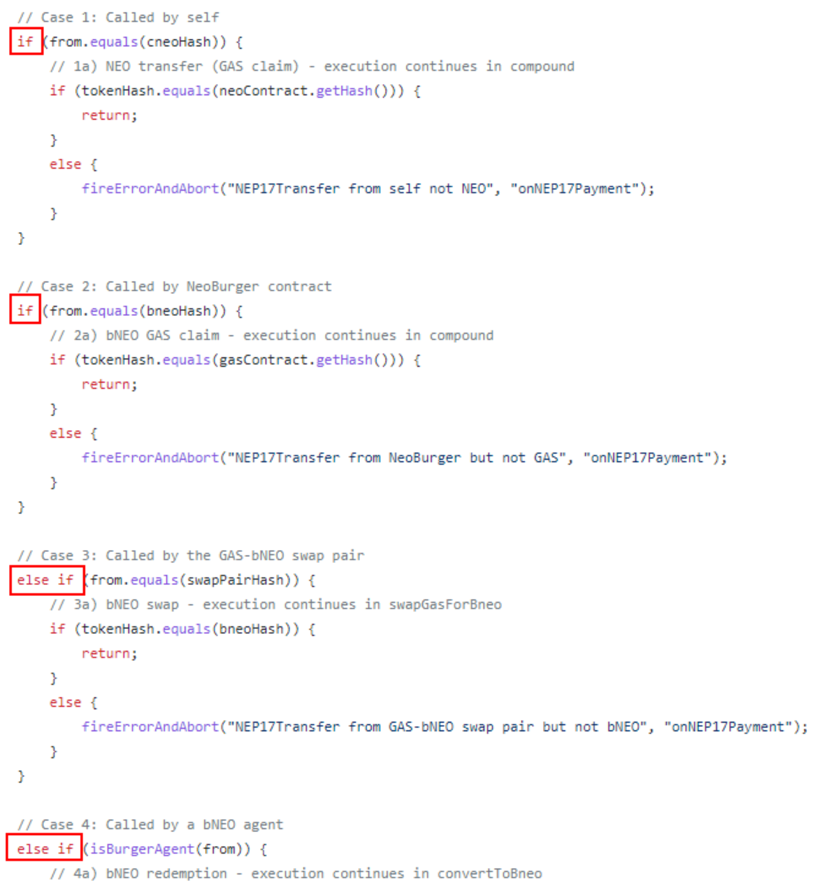
Bad Coding Practices

Identifier	Category	Risk	State
NCP-13	Codebase Quality	**Informative**	**Fixed**

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.

It has been possible to verify that, there is a lack of order and structure that makes reading and analyzing the code difficult. Following, we detail a few points that could be improved in terms of style, quality, and readability of the code throughout the audited contracts.

The `onPayment` method controls the possible cases in which it can be invoked, however, the same structure is not being followed for each one. As can be seen in the following image, the first cases are checked through `if` conditionals. But, from case 3 onwards, it is verified with `else-if` statements.



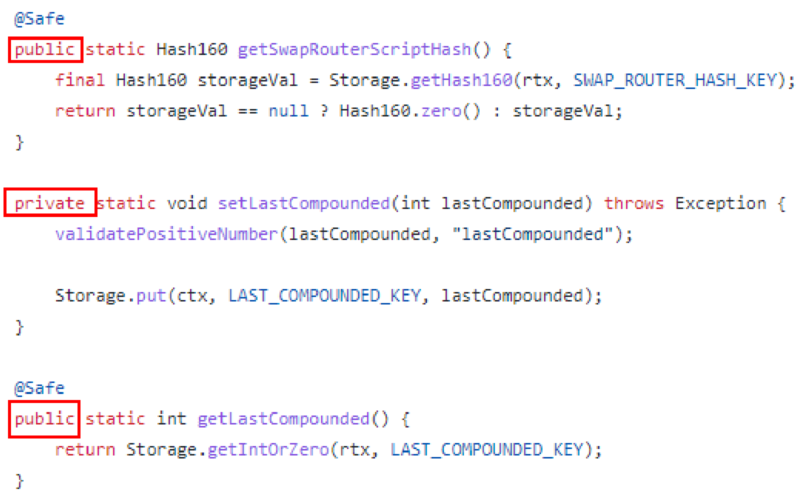
```
// Case 1: Called by self
if (from.equals(cneoHash)) {
    // 1a) NEO transfer (GAS claim) - execution continues in compound
    if (tokenHash.equals(neoContract.getHash())) {
        return;
    }
    else {
        fireErrorAndAbort("NEP17Transfer from self not NEO", "onNEP17Payment");
    }
}

// Case 2: Called by NeoBurger contract
if (from.equals(bneoHash)) {
    // 2a) bNEO GAS claim - execution continues in compound
    if (tokenHash.equals(gasContract.getHash())) {
        return;
    }
    else {
        fireErrorAndAbort("NEP17Transfer from NeoBurger but not GAS", "onNEP17Payment");
    }
}

// Case 3: Called by the GAS-bNEO swap pair
else if (from.equals(swapPairHash)) {
    // 3a) bNEO swap - execution continues in swapGasForBneo
    if (tokenHash.equals(bneoHash)) {
        return;
    }
    else {
        fireErrorAndAbort("NEP17Transfer from GAS-bNEO swap pair but not bNEO", "onNEP17Payment");
    }
}

// Case 4: Called by a bNEO agent
else if (isBurgerAgent(from)) {
    // 4a) bNEO redemption - execution continues in convertToBneo
```

It is recommended to always use the same conditional structure in order to improve the readability, auditability and maintainability fo the code.

Source Code References

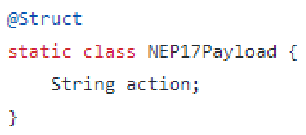
- CompoundingNeo.java#L598
- CompoundingNeo.java#L609

Furthermore, it has been identified that certain parts of the code are not correctly organized, an example of this is the `setLastCompounded` method, which has its visibility set to `private`, but it is located between public methods.



```
@Safe
public static Hash160 getSwapRouterScriptHash() {
    final Hash160 storageVal = Storage.getHash160(rtx, SWAP_ROUTER_HASH_KEY);
    return storageVal == null ? Hash160.zero() : storageVal;
}

private static void setLastCompounded(int lastCompounded) throws Exception {
    validatePositiveNumber(lastCompounded, "lastCompounded");

    Storage.put(ctx, LAST_COMPOUNDED_KEY, lastCompounded);
}

@Safe
public static int getLastCompounded() {
    return Storage.getIntOrZero(rtx, LAST_COMPOUNDED_KEY);
}
```

Another example of this is the `NEP17Payload` struct, that should be located with the rest of the variables/structs declarations and not between different methods as it currently is.



```
@Struct
static class NEP17Payload {
    String action;
}
```

Source Code References

- CompoundingNeo.java#L234
- CompoundingNeo.java#L386-L390

Fixes Review

This issue has been addressed in the following commit:

- https://github.com/neocompounder/cneo-contract/commit/69b8d1e901385eb6ab302060dc81db8a5e7db902

Annexes

Methodology

A code audit is a thorough examination of the source code of a project with the objective of identifying errors, discovering security breaches, or contraventions of programming standards. It is an essential component to the defense in programming, which seeks to minimize errors prior to the deployment of the product.

Red4Sec adopts a set of cybersecurity tools and best security practices to audit the source code of the smart contract by conducting a search for vulnerabilities and flaws.

The audit team performs an analysis on the functionality of the code, a manual audit, and automated verifications, considering the following crucial features of the code:

- The implementation conforms to protocol standards and adheres to best coding practices.
- The code is secure against common and uncommon vectors of attack.
- The logic of the contract complies with the specifications and intentions of the client.
- The business logic and the interactions with similar industry protocols do not contain errors or lead to dangerous situations to the integrity of the system.

In order to standardize the evaluation, the audit is executed by industry experts, in accordance with the following procedures:

Manual Analysis

- Manual review of the code, line-by-line, to discover errors or unexpected conditions.
- Assess the overall structure, complexity, and quality of the project.
- Search for issues based on the SWC Registry and known attacks.
- Review known vulnerabilities in the third-party libraries used.
- Analysis of the business logic and algorithms of the protocol to identify potential risk exposures.
- Manual testing to verify the operation, optimization, and stability of the code.

Automatic Analysis

- Scan the source code with static and dynamic security tools to search for known vulnerabilities.
- Manual verification of all the issues found by the tools and analyzes their possible impact.
- Perform unit tests and verify the coverage.

Vulnerabilities Severity

Red4Sec determines the severity of vulnerabilities found in risk levels according to the impact level defined by CVSSv3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST), classifying the risk of vulnerabilities on the following scale:

Severity	Description
Critical	Vulnerabilities that possess the highest impact over the systems, services and/or sensitive information. The existence of these vulnerabilities is dangerous and should be fixed as soon as possible.
High	Vulnerabilities that could compromise severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.
Medium	Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.
Low	These vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.
Informative	It covers various characteristics, information or behaviours that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

RED4SEC

Invest in Security, invest in your future