Smart Contracts Security Audit

LyreBird

2022-02-10

1. Introduction

Lyrebird is an algorithmic stablecoin protocol built on Neo N3. At its core, Lyrebird is a collection of smart contracts that enable the issuance of *L-assets* - NEP-17 tokens that are soft-pegged to desired currencies. At launch, Lyrebird will support *Lyrebird USD* (USDL), an L-asset pegged to the US Dollar. Other L-assets will be created based on community feedback once voting is enabled.

As requested by **Lyrebird** 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 *Lyrebird project*.

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, 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.

3. Scope

The scope of this evaluation includes the following projects enclosed in the repository:

- https://bitbucket.org/77696c6c/lyrebird-contract/src/master
  - branch: *master*
  - commit: *7edd6ebc32b2e1d34726a506d47167f0e24762da*
- **Remediations review** (10/02/2022)
  - branch*: master*
  - commit*: 6388dd414fc61e7ada425d011d24de67c07eb4a5*

4. Conclusions

To this date, 10th of February 2022, the general conclusion resulting from the conducted audit and after further review of the applied fixes, is that **LyreBird Smart Contracts are secure** and do not currently present known vulnerabilities that could compromise the security of the project.

The general conclusions of the performed audit are:

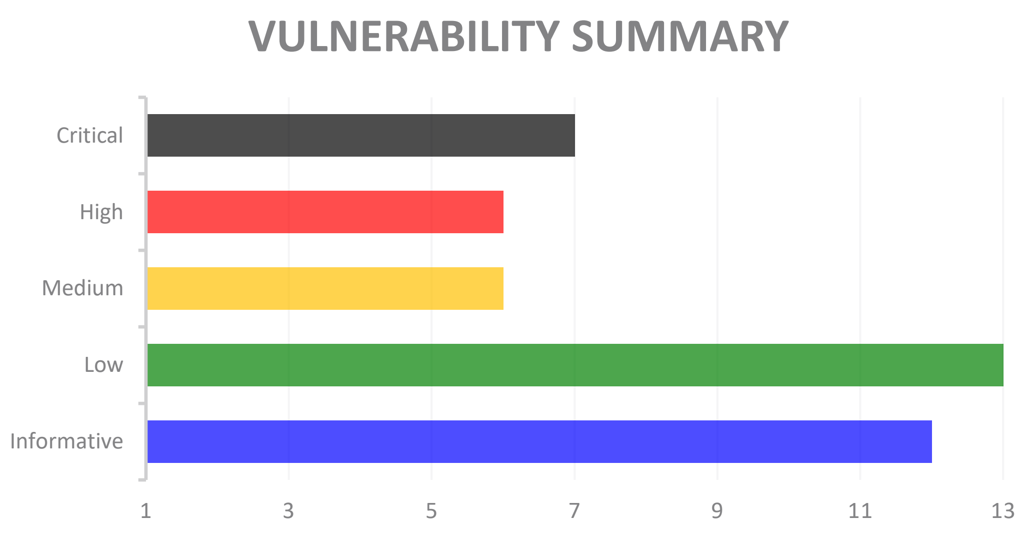
- Seven Critical vulnerabilities were detected during the security audit.
- Six High-risk vulnerabilities were detected during the security audit.

Nevertheless, LyreBird’s technical team has made an action plan to guarantee the resolution of the detected vulnerabilities.

- Certain detected vulnerabilities do not pose a risk by themselves and have been classified as informative or low risk vulnerabilities. However, it was advised to apply all the proposed recommendations in order to improve the project source code and ecosystem and most were properly applied.

- It is important to highlight that both, the new N3 blockchain and the smart contract compiler `neo3-boa` 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.

All these 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 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**
	General Project Issues		
LBD01	Wrong Testing Environment	High	Fixed
LBD02	Alpha/Beta Compiler	Medium	Assumed
LBD03	Lack of StorageMap	Low	Fixed
LBD04	Lack of Safe method attribute	Low	Open
LBD05	Decentralization Recommendation	Informative	Assumed
LBD06	Unnecessary Encoding	Informative	Assumed
LBD07	Optimize storage prefixes	Informative	Assumed
LBD08	Safe Contracts Update	Informative	Assumed
LBD09	Safe Storage Access	Informative	Fixed
	LyreBird USD Token		
LBD10	Arbitrary token burn	Critical	Fixed
LBD11	Lack of Inputs Validation	Low	Fixed
LBD12	Missing manifest SupportedStandards attribute	Low	Fixed
LBD13	Denial of Service in the query methods	Low	Fixed
LBD14	GAS Optimization	Informative	Partially Fixed
	LyreBird Token		
LBD15	Arbitrary token burn	Critical	Fixed
LBD16	Drain of Funds by Reentrancy	Critical	Fixed
LBD17	Unchecked Transfer Return	High	Fixed
LBD18	Lack of Inputs Validation	Medium	Fixed
LBD19	Unbounded Loop in method	Low	Assumed
LBD20	Missing manifest SupportedStandards attribute	Low	Fixed
LBD21	Denial of Service in the query methods	Low	Partially Fixed
LBD22	Lack of Token WhiteList	Low	Fixed
LBD23	GAS Optimization	Informative	Partially Fixed

LyreBird Hatchery			
LBD24	Limit Call Rights	Low	Fixed
LBD25	Lack of Token WhiteList	Informative	Fixed
LBD26	Unchecked Transfer Return	Informative	Fixed
LBD27	GAS Optimization	Informative	Fixed
LyreBird Cage			
LBD28	Drain of Funds by Reentrancy	Critical	Fixed
LBD29	Exposed Private Method	High	Fixed
LBD30	Unchecked Transfer Return	High	Fixed
LBD31	Lack of Inputs Validation	Medium	Fixed
LBD32	Lack of Abort on missing authorization	Low	Fixed
LBD33	GAS Optimization	Informative	Fixed
LyreBird Aviary			
LBD34	Broken delta update logic	Critical	Fixed
LBD35	Broken Last Height update logic	Critical	Fixed
LBD36	Broken distributeFees Logic	Critical	Fixed
LBD37	Unchecked Transfer Return	High	Fixed
LBD38	DistributeFees Reentrancy	High	Fixed
LBD39	Lack of Inputs Validation	Medium	Partially Fixed
LBD40	Lack of Abort on missing authorization	Medium	Fixed
LBD41	Unsafe token detection	Medium	Fixed
LBD42	Centralized Oracle Price	Low	Open
LBD43	Limit Call Rights	Low	Fixed
LBD44	GAS Optimization	Informative	Partially Fixed

5. Issues and Recommendations

General Project Issues

LBD01 – Wrong Testing Environment (High)

In the Lyrebird project, all the contracts have a **`TEST_MODE`** constant with a function of signature verification bypass, which intends to simplify the development of unit tests.

This practice is highly discouraged and can produce discrepancies between the tested code and the code that will finally be executed in the blockchain, as is the case with the following issues:

- Lack of Abort on authorization failure
- Broken delta update logic
- Broken LastHeight update logic

Besides producing an unnecessary gas cost for the executions that are in production, it invalidates unitary tests, which has the purpose of verifying the correct execution of the code and makes sure that if certain logics are forged, it cannot be guaranteed.

It is recommended to remove said constant and logic, in addition to including the signatures required during the execution of the unit tests.

Fixes Review

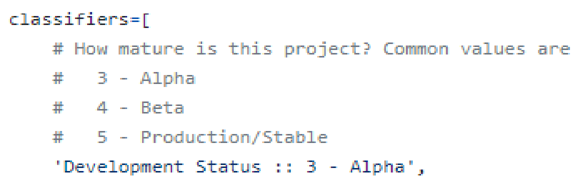
This issue has been fixed in the following commit:

- lyrebird-contract/commits/308d988ec3d13c9cc8caa92cb0431274fcd4c51b

LBD02 - Alpha/Beta Compiler (Medium)

The current smart contract has been developed using a programming language which is in "alpha" state for Neo's development. Our proven experience determines that the Python NEO compiler is still in a development state and that the presence of errors is more common than in other languages for smart contracts.

Languages such as C# is more stable and is the recommended option for smart contracts developed in NEO.



```
classifiers=[
    # How mature is this project? Common values are
    #   3 - Alpha
    #   4 - Beta
    #   5 - Production/Stable
    'Development Status :: 3 - Alpha',
```

The following issues in this report are examples of the lack of maturity of the neo3-boa compiler:

- Lack of Safe method attribute.
- Lack of constants.

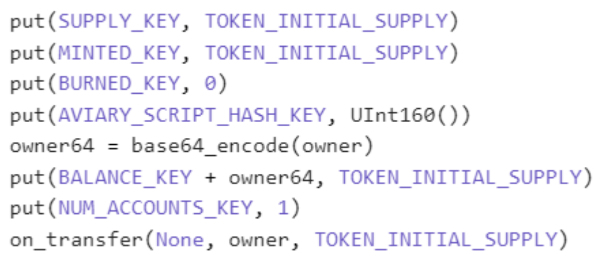
References

https://github.com/CityOfZion/neo3-boa/blob/7e3ba935228230a332bf96ca5e8fc3cdf83aba60/setup.py#L64

LBD03 – Lack of StorageMap (Low)

It has been detected that the storage of Smart Contract values is not performed in a secure manner. It is recommended to use *StorageMap*, instead of *Storage*, since this class adds the prefix and avoids possible errors.

In the following image you can see how the *StorageMap* class is not used:



```
put(SUPPLY_KEY, TOKEN_INITIAL_SUPPLY)
put(MINTED_KEY, TOKEN_INITIAL_SUPPLY)
put(BURNED_KEY, 0)
put(AVIARY_SCRIPT_HASH_KEY, UInt160())
owner64 = base64_encode(owner)
put(BALANCE_KEY + owner64, TOKEN_INITIAL_SUPPLY)
put(NUM_ACCOUNTS_KEY, 1)
on_transfer(None, owner, TOKEN_INITIAL_SUPPLY)
```

By adding prefixes with *StorageMap* to the storages, a large part of the vulnerabilities related to arbitrary writing in storage contracts would be avoided, mitigating in some way the appearance of new vulnerabilities.

Source reference

- All contracts are affected.

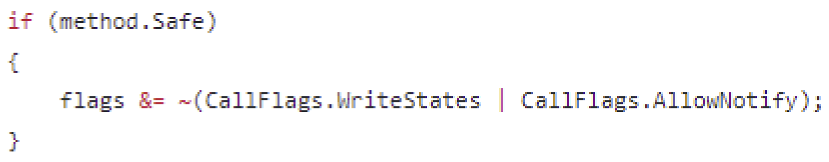
Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/120b89489d99e7ed883bed6178240d4de90f0f3a

LBD04 – Lack of Safe methods attribute (Low)

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



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

Additionally, it will provide the wallets and dApps with the necessary information to identify it as a query method and to make a reading invocation with no GAS costs. So, it is convenient to establish our query methods as *Safe* to keep the principle of least privilege[1].

Currently, the neo-boa compiler allows setting the *safe* methods using the development version that contains the pull request #793 with the `@public(safe=True)` decorator.

LBD05 – Decentralization Recommendation (Informative)

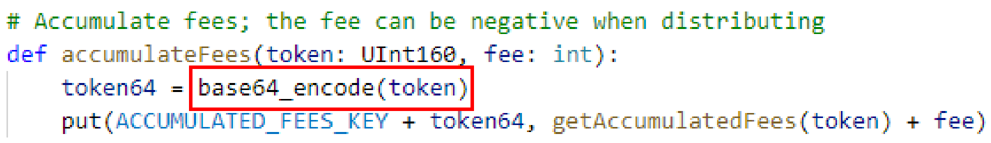
The LyreBird team maintains some centralized parts that imply trust in the project; a few of the administrative functionalities are under the control of the project. Therefore, according to the logic of smart contracts, the owner and author are able to claim any token and to modify certain values of the contract at will.

Additionally, the contracts do not have the ability to modify the owner, so in case of wanting to change to a more decentralized governance system or leak the private key, the admin is not to be modified and it will force the contract to be updated.

[1] https://en.wikipedia.org/wiki/Principle_of_least_privilege

LBD06 – Unnecessary Encoding (Informative)

During the logic of all the audited contracts, the user's addresses are systematically converted to base64, which implies that to make use of these values they must be continuously coded and decoded.



```
# Accumulate fees; the fee can be negative when distributing
def accumulateFees(token: UInt160, fee: int):
    token64 = base64_encode(token)
    put(ACCUMULATED_FEES_KEY + token64, getAccumulatedFees(token) + fee)
```

This extra logic adds computational cost, it unnecessarily increases the gas consumption and the storage required, since base64 implies an average of ~33% increment of the data original[2].

When working with public addresses, it is recommended to work with bytes, as long as it is necessary to present this information in a different way, the dApp should be making the conversion, so it is transparent for the smart contract.

LBD07 – Optimize storage prefixes (Informative)

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.

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 reference

- All contracts are affected.

[2] https://developer.mozilla.org/en-US/docs/Glossary/Base64#encoded_size_increase

LBD08 – Safe Contracts Update (Informative)

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.

There are a few good practices that help mitigate this problem, such as; add a *timelock* to start the *Update* operations, emit events when the *Update* operation is requested, temporarily disable contract functionalities and finally issue a last notification and execute the *Update* operation after the timelock is completed.

Source reference

- src/LyrebirdUSDToken.py#lines-380
- src/LyrebirdToken.py#lines-720
- src/LyrebirdHatchery.py#lines-169
- src/LyrebirdCage.py#lines-357
- src/LyrebirdAviary.py#lines-730

LBD09 – Safe Storage Access (Informative)

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)[3] in order to avoid future problems.

In the case of Python, the accesses to the storage for the query methods should follow the recommended format shown below:

```
get(KEY, get_read_only_context())
```

[3] https://en.wikipedia.org/wiki/Principle_of_least_privilege

Source reference

- All contracts are affected.

Fixes Review

This issue has been fixed in the following commit:

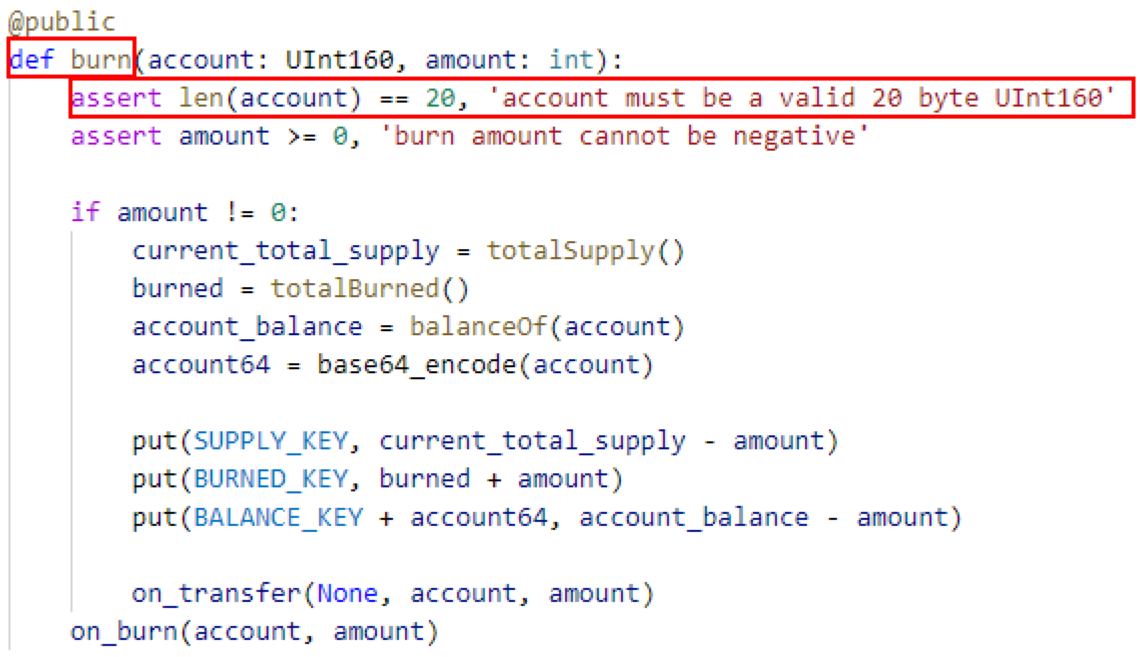
- lyrebird-contract/commits/120b89489d99e7ed883bed6178240d4de90f0f3a

LyreBird USD Token Issues

LBD10 – Safe Storage Access (Critical)

The *burn* method does not verify any type of authorization, which allows any user to burn tokens from arbitrary accounts, this action should be limited to the *sender* of the transaction, or in case of opting for a more centralized solution, it should be limited to the owner of the contract.

Following, it can be observed that only the length of the provided account is verified, however, it is not verified if it matches with the sender/signer of the transaction.



```
@public
def burn(account: UInt160, amount: int):
    assert len(account) == 20, 'account must be a valid 20 byte UInt160'
    assert amount >= 0, 'burn amount cannot be negative'

    if amount != 0:
        current_total_supply = totalSupply()
        burned = totalBurned()
        account_balance = balanceOf(account)
        account64 = base64_encode(account)

        put(SUPPLY_KEY, current_total_supply - amount)
        put(BURNED_KEY, burned + amount)
        put(BALANCE_KEY + account64, account_balance - amount)

        on_transfer(None, account, amount)
    on_burn(account, amount)
```

Source reference

- src/LyrebirdUSDToken.py#lines-291

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/d3cdeab52a4b4ddb84413ba627952571225a5a94

LBD11 – Lack of Inputs Validation (Low)

Some methods of the different contracts in the **Lyrebird USD Token** contract do not properly check the arguments, which can lead to major errors.

It is advisable to always check the format and type 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.

It is important to know that the type of the arguments from an invocation are not natively verified by neo VM and that neo3-boa does not add any type of verification during its compilation, therefore, all the arguments must be carefully verified.

The integrity of the `UInt160` types is only verified by its size (20 bytes) however, it is more convenient to use the *isinstance()* method since it makes verifications by type and size.

Additionally, in some methods it is convenient to check that the address value is not address `zero`, which could lead to unwanted behaviours in certain occasions.

It is convenient to add a method that unifies the format verification of the addresses, as suggested below:

```python
def validateAddress(address: UInt160) -> bool:
    if not isinstance(address, UInt160):
        return False
    if address == 0:
        return False
    return True
```

Source reference

- src/LyrebirdUSDToken.py#lines-91
- src/LyrebirdUSDToken.py#lines-137
- src/LyrebirdUSDToken.py#lines-173
- src/LyrebirdUSDToken.py#lines-257
- src/LyrebirdUSDToken.py#lines-291

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/36d9e35b5d308a0171172c156afddd1c11a2b3aa

LBD12 – Missing manifest SupportedStandards attribute (Low)

The Red4Sec team has detected that the Token contract does not properly specify the supported standards in its manifest. Presumably, LyrebirdUSDToken should specify NEP-17 in its manifest in order to be aligned with Neo's standards.

The *supportedstandards* field of the Manifest file describes which standard is supported, such as NEP or RFC. It must be an array. In order to make smart contracts or other clients understand correctly, all NEPs must be capitalized. NEP and number must be connected with '-'.

An example of *supportedstandards* is as follows:

```
["NEP-11", "NEP-17", "RFC 1035"]
```

References

- https://github.com/neo-project/proposals/blob/master/nep-15.mediawiki
- https://dojo.coz.io/neo3/boa/boa3/builtin/boa3-builtins.html?highlight=neometadata#boa3.builtin.NeoMetadata
- https://dojo.coz.io/neo3/boa/boa3/builtin/interop/contract/boa3-builtin-interop-contract-contractmanifest.html

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/2b3a7f822ae22740222a08874f95268e78cbb2ec

LBD13 – Denial of Service in the query methods (Low)

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

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

Source reference

- src/LyrebirdUSDToken.py#lines-144

Fixes Review

This issue has been addressed in the following commit:

- lyrebird-contract/commits/56e460e99e4e977765d5f7ca0b6046c207ad85a7

LBD14 – 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.

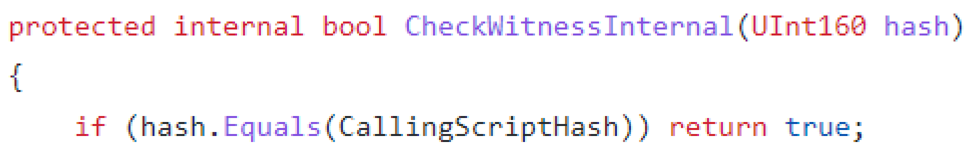
On the N3 blockchain, GAS is an execution fee which is used to compensate the network for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

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.

Unnecesary Logic

The *CheckWitness* syscall, internally verifies if the received argument is equal to *CallingScriptHash* and that it returns *true*. (ApplicationEngine.Runtime.cs#L198).



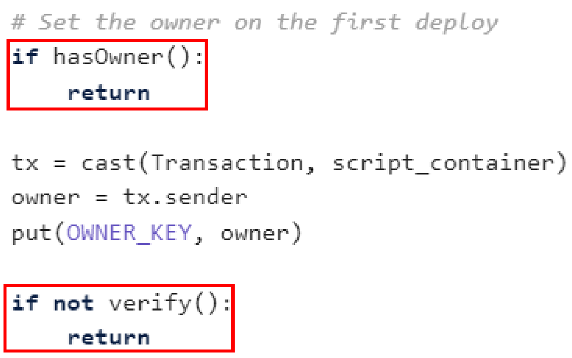
```
protected internal bool CheckWitnessInternal(UInt160 hash)
{
    if (hash.Equals(CallingScriptHash)) return true;
```

Therefore, it is redundant to make this verification and it increases the cost of gas for the execution.

Source reference

- src/LyrebirdUSDToken.py#lines-207

During the execution of the *_deploy* method, a check of the *hasOwner* is performed and it also checks that the owner signed the transaction, these verifications result unnecessary since the method can only be executed during the contract's deployment, so the owner will never be established, thus there is no need for an identity verification.



```
# Set the owner on the first deploy
if hasOwner():
    return

tx = cast(Transaction, script_container)
owner = tx.sender
put(OWNER_KEY, owner)

if not verify():
    return
```

Source reference

- src/LyrebirdUSDToken.py#lines-359:367

Unnecesary on-chain Metrics

The *NUM_ACCOUNTS_KEY* and *MINTED_KEY* storage keys are intended to store metrics of the contract usage, however this information is unnecessary for the proper functioning of the contract and it should not be stored on-chain, since it increases the GAS and storage consumption. These statistics can be easily resolved by querying the nodes of the blockchain.

Therefore, it is recommended to eliminate both the storage keys and the query methods associated with them, such as: *numAccounts* and *totalMinted*.

Source reference

- src/LyrebirdUSDToken.py#lines-43
- src/LyrebirdUSDToken.py#lines-45
- src/LyrebirdUSDToken.py#lines-121:123
- src/LyrebirdUSDToken.py#lines-132:133

Wrong Visibility

The ***callByAviary*** is a public method, nevertheless it does not have to be, making methods public increases the gas cost by increasing the size of the manifest and it also increases the surface of exposure to attacks.

Source reference

- src/LyrebirdUSDToken.py#lines-335

Fixes Review

This issue has been partially fixed in the following commit:

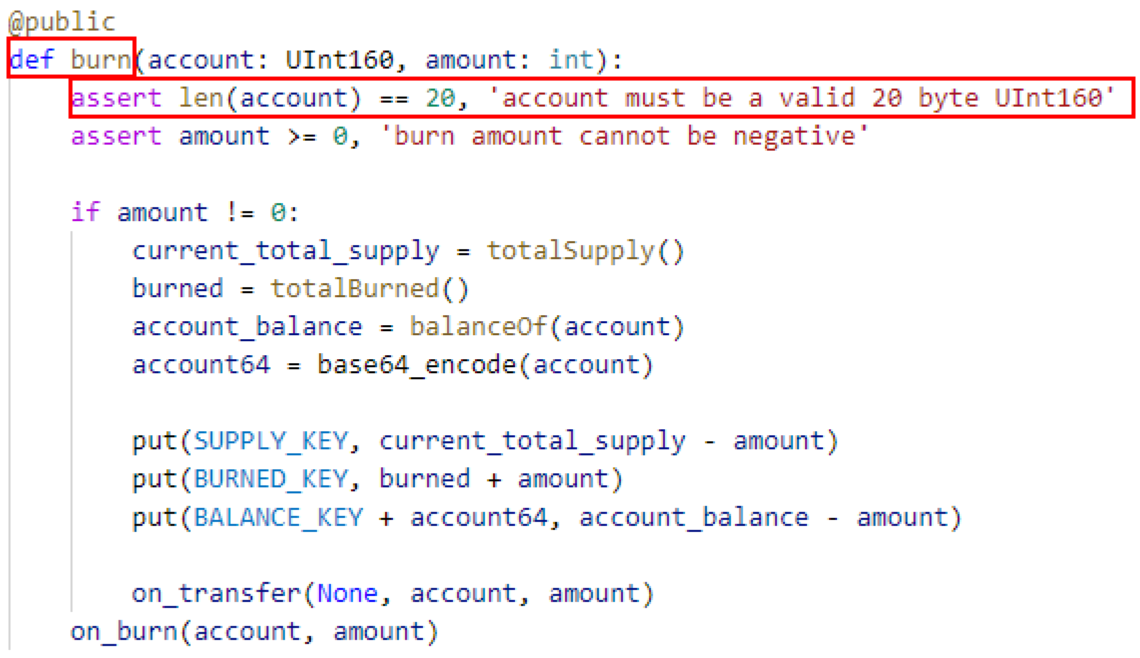
- lyrebird-contract/commits/64bd7db5b86b3384394fec4e6139166377722f89

LyreBird Token Issues

LBD15 – Arbitrary token burn (Critical)

The *burn* method does not verify any type of authorization, which allows any user to burn tokens from arbitrary accounts, this action should be limited to the *sender* of the transaction, or in case of opting for a more centralized solution, it should be limited to the owner of the contract.

Following, it can be observed that only the length of the provided account is verified, however, it is not verified if it matches with the sender/signer of the transaction.



```
@public
def burn(account: UInt160, amount: int):
    assert len(account) == 20, 'account must be a valid 20 byte UInt160'
    assert amount >= 0, 'burn amount cannot be negative'

    if amount != 0:
        current_total_supply = totalSupply()
        burned = totalBurned()
        account_balance = balanceOf(account)
        account64 = base64_encode(account)

        put(SUPPLY_KEY, current_total_supply - amount)
        put(BURNED_KEY, burned + amount)
        put(BALANCE_KEY + account64, account_balance - amount)

        on_transfer(None, account, amount)
    on_burn(account, amount)
```

Source reference

- src/LyrebirdToken.py#lines-535

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/4cf35a48e9483aa78dd0ec285bbb1aeba8ebde87

LBD16 – Drain of Funds by Reentrancy (Critical)

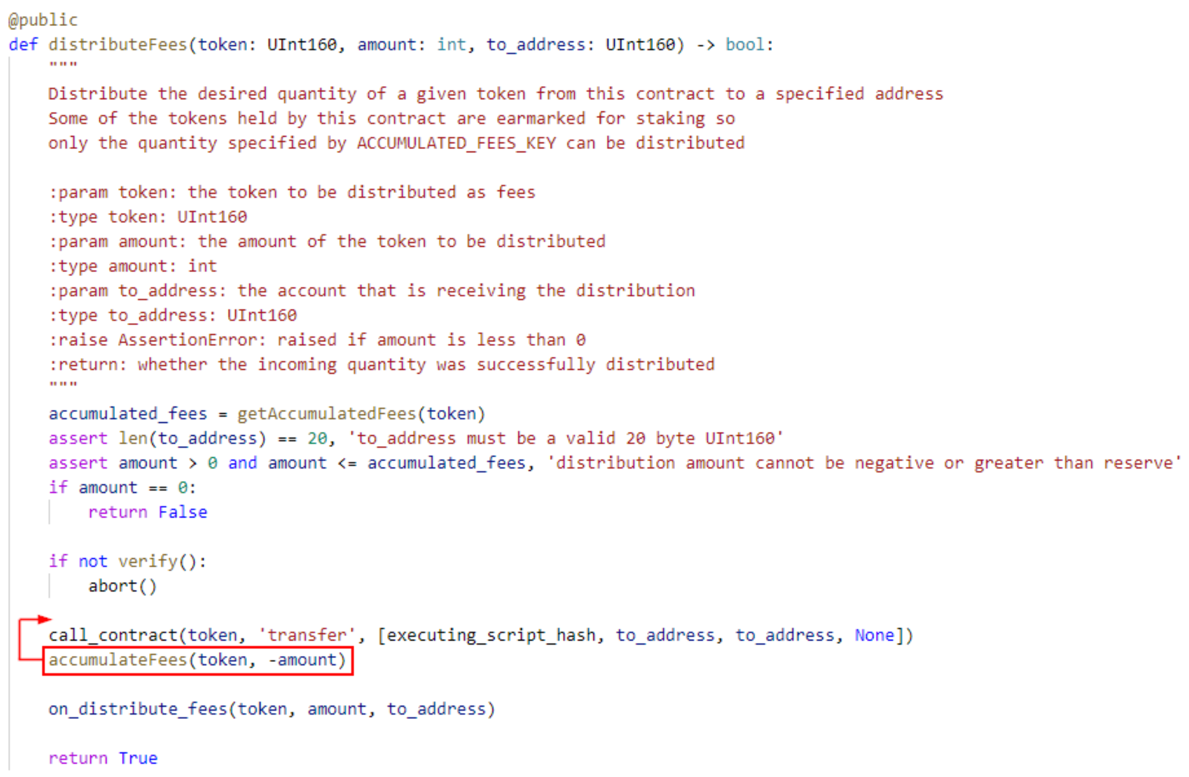
The Reentrancy attack is a vulnerability that occurs when external contract calls can make new calls to the calling contract, before the initial execution is completed. For a function, this means that the state of the contract could change in the middle of its execution as a result of a call to an untrusted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

This attack is possible in N3 because the NEP17[4] and NEP11[5] 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.

Therefore, 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.

In the case of the *distributeFees* method of the **LyrebirdToken** contract, the values are updated after the transfers are made so the call can be redirected to the *distributeFees* method before updating the values. In this case it can be used to drain the funds of the contract, as can be seen in the following image, it is necessary to call the *accumulateFees* method before calling external contracts.

[4] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki
[5] https://github.com/neo-project/proposals/blob/master/nep-11.mediawiki



```
@public
def distributeFees(token: UInt160, amount: int, to_address: UInt160) -> bool:
    """
    Distribute the desired quantity of a given token from this contract to a specified address
    Some of the tokens held by this contract are earmarked for staking so
    only the quantity specified by ACCUMULATED_FEES_KEY can be distributed

    :param token: the token to be distributed as fees
    :type token: UInt160
    :param amount: the amount of the token to be distributed
    :type amount: int
    :param to_address: the account that is receiving the distribution
    :type to_address: UInt160
    :raise AssertionError: raised if amount is less than 0
    :return: whether the incoming quantity was successfully distributed
    """
    accumulated_fees = getAccumulatedFees(token)
    assert len(to_address) == 20, 'to_address must be a valid 20 byte UInt160'
    assert amount > 0 and amount <= accumulated_fees, 'distribution amount cannot be negative or greater than reserve'
    if amount == 0:
        return False

    if not verify():
        abort()

    call_contract(token, 'transfer', [executing_script_hash, to_address, to_address, None])
    accumulateFees(token, -amount)

    on_distribute_fees(token, amount, to_address)

    return True
```

Source reference

- src/LyrebirdToken.py#lines-623:624

Fixes Review

This issue has been fixed in the following commit:

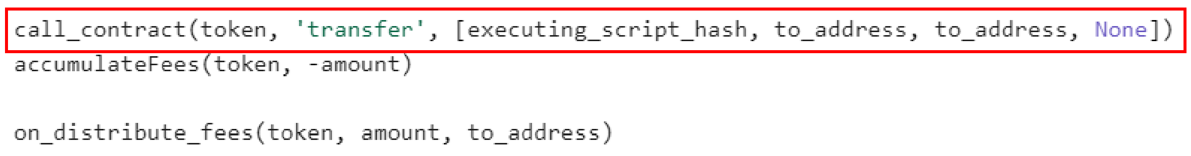
- lyrebird-contract/commits/8cdf98650a1ceabae4740edf4144297cfe3c93b3

LBD17 – Unchecked Transfer Return (High)

The NEP17[6] standard specifies that the transfer functions will return a boolean with the result of this operation.

The **LyrebirdToken** contract does not contemplate this result, although some NEP-17 token implementations make a revert if these methods fail, the standard dictates that cases such as; not enough balance or wrong signatures, should return a false. This is the case for NEO, GAS and Lyrebird tokens.

[6] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki#transfer



```
call_contract(token, 'transfer', [executing_script_hash, to_address, to_address, None])
accumulateFees(token, -amount)

on_distribute_fees(token, amount, to_address)
```

The ***distributeFees*** method does not guarantee that the contract has enough funds and if the transfer fails, before updating, the distributed fees will be registered.

The result of external contract calls should always be verified and It is mandatory to check that the returned value is *true* in all of the transfers.

Source reference

- src/LyrebirdToken.py#lines-623

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/8cdf98650a1ceabae4740edf4144297cfe3c93b3

LBD18 – Lack of Inputs Validation (Medium)

Some methods of the different contracts in the **Lyrebird Token** contract do not properly check the arguments, which can lead to major errors.

It is advisable to always check the format and type 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.

It is important to know that the `type` of the arguments from an invocation are not natively verified by NEO VM and that neo3-boa does not add any type of verification during its compilation, therefore, all the arguments must be carefully verified.

The integrity of the `UInt160` types is only verified by its size (20 bytes) however, it is more convenient to use the *isinstance()* method since it makes verifications by type and size.

Additionally, in some methods it is convenient to check that the address value is not address `zero`, which could lead to unwanted behaviours in certain occasions.

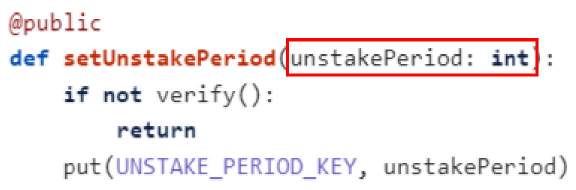
It is convenient to add a method that unifies the format verification of the addresses, as suggested below:

```
def validateAddress(address: UInt160) -> bool:
    if not isinstance(address, UInt160):
        return False
    if address == 0:
        return False
    return True
```

Source reference

- src/LyrebirdToken.py#lines-140
- src/LyrebirdToken.py#lines-202
- src/LyrebirdToken.py#lines-220
- src/LyrebirdToken.py#lines-257
- src/LyrebirdToken.py#lines-304
- src/LyrebirdToken.py#lines-345
- src/LyrebirdToken.py#lines-367
- src/LyrebirdToken.py#lines-414
- src/LyrebirdToken.py#lines-501
- src/LyrebirdToken.py#lines-535
- src/LyrebirdToken.py#lines-599
- src/LyrebirdToken.py#lines-632

Besides the verifications of types for *UInt160* arguments, it must be mentioned that there are two methods where the integer arguments are not checked to be greater than *0*, *onNEP17Payment* and *setUnstakePeriod*, in both methods the amount received should be checked to be positive.



```
@public
def setUnstakePeriod(unstakePeriod: int):
    if not verify():
        return
    put(UNSTAKE_PERIOD_KEY, unstakePeriod)
```

Source reference

- src/LyrebirdToken.py#lines-632
- src/LyrebirdToken.py#lines-213

Fixes Review

This issue has been fixed in the following commit:

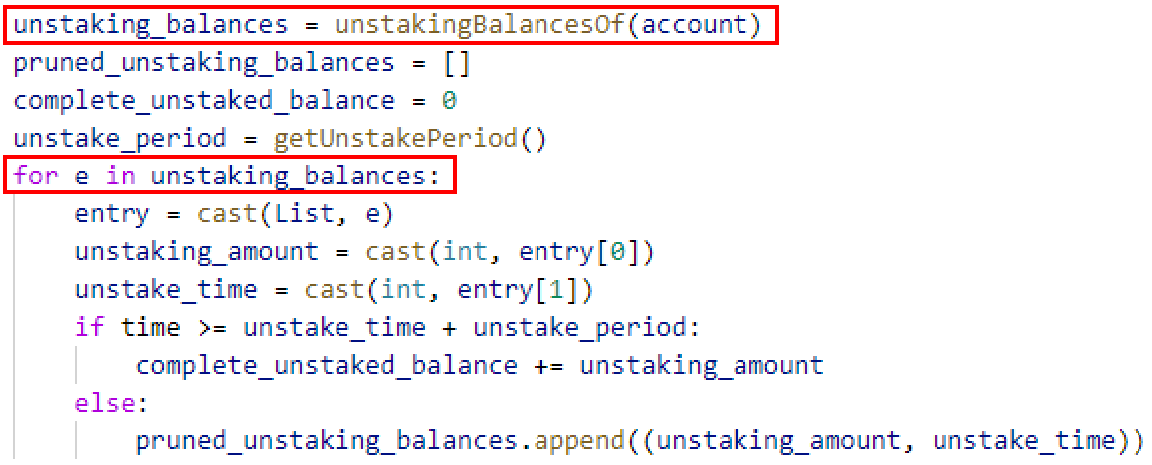
- lyrebird-contract/commits/e65d5e61cbbfb0bef0f64dee5bfa33ae7239423c

LBD19 – Unbounded Loop in method (Low)

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

The logic executed in *claimUnstakedBalances* method might trigger a denial of service (DoS) by GAS exhaustion because it iterates without limits over the user's unstakes, a value obtained by calling to the *unstakingBalancesOf* method.

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



```
unstaking_balances = unstakingBalancesOf(account)
pruned_unstaking_balances = []
complete_unstaked_balance = 0
unstake_period = getUnstakePeriod()
for e in unstaking_balances:
    entry = cast(List, e)
    unstaking_amount = cast(int, entry[0])
    unstake_time = cast(int, entry[1])
    if time >= unstake_time + unstake_period:
        complete_unstaked_balance += unstaking_amount
    else:
        pruned_unstaking_balances.append((unstaking_amount, unstake_time))
```

Source reference

- src/LyrebirdToken.py#lines-325

LBD20 – Missing manifest SupportedStandards attribute (Low)

The Red4Sec team has detected that the Token contract does not properly specify the supported standards in its manifest. Presumably, Lyrebird Token should specify NEP-17 in its manifest in order to be aligned with Neo's standards.

The *supportedstandards* field of the Manifest file describes which standard is supported, such as NEP or RFC. It must be an array. In order to make smart contracts or other clients understand correctly, all NEPs must be capitalized. NEP and number must be connected with '-'.

An example of *supportedstandards* is as follows:

```
["NEP-11", "NEP-17", "RFC 1035"]
```

References

- https://github.com/neo-project/proposals/blob/master/nep-15.mediawiki
- https://dojo.coz.io/neo3/boa/boa3/builtin/boa3-builtins.html?highlight=neometadata#boa3.builtin.NeoMetadata
- https://dojo.coz.io/neo3/boa/boa3/builtin/interop/contract/boa3-builtin-interop-contract-contractmanifest.html

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/43206af95ef43b52b85711718345c93d88b567d7

LBD21 – Denial of Service in the query methods (Low)

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

Taking into consideration a scenario where these query methods fail at some point by exceeding the limits of the virtual machine and returning *FAULT* in its execution, a denial of service would be produced, this limits the *getBalances, stakedQuantities* and *unstakingBalancesOf* methods.

Source reference

- src/LyrebirdToken.py#lines-227 (getBalances)
- src/LyrebirdToken.py#lines-276 (stakedQuantities)
- src/LyrebirdToken.py#lines-345 (unstakingBalancesOf)

Fixes Review

This issue has been addressed in the following commit:

- lyrebird-contract/commits/5e0c62bdbc5f274f1ad8ed684b01ba85b86f30e3

LBD22 – Lack of Token WhiteList (Low)

The *OnNEP17Payment* method of the **LyrebirdToken** contract does not limit the tokens received, in the case of the *ACTION_ACCUMULATE_FEES* type, this allows to send any NEP-17 token to the contract, which can facilitate phishing attacks through a malicious token.

If the *owner* decides to send the tokens from the malicious contract to another account, and it uses a *Scope* of signatures from *Global* (or similar), the malicious contract could steal funds from the account by using the signature.

Even though the *Scope* prevents these situations, it is always advised to reduce the exposure surface. Adding a whitelist of allowed tokens is a good practice and secures our contracts and the users.

Source reference

- src/LyrebirdToken.py#lines-653

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/6c43004fc533b1f981549cab4b7b19da08fc178e

LBD23 – 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 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.

Unused Code

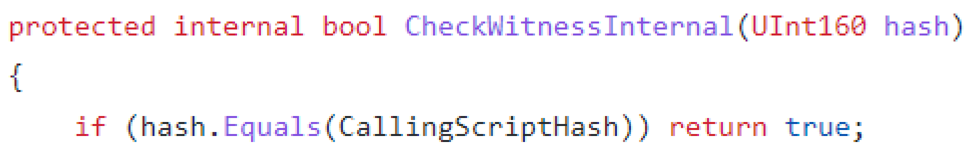
The *DISTRIBUTED_KEY* constant is not used throughout the **LyrebirdToken** contract. NeoBoa compiles all of the constants as static variables, so the cost of execution increases since they are initialized during the call to the *_initialize* method, everytime the contract is invoked.

Source reference

- src/LyrebirdToken.py#lines-46

Unnecesary Logic

The *CheckWitness* syscall, internally verifies if the received argument is equal to *CallingScriptHash* and that it returns *true*. (ApplicationEngine.Runtime.cs#L198).



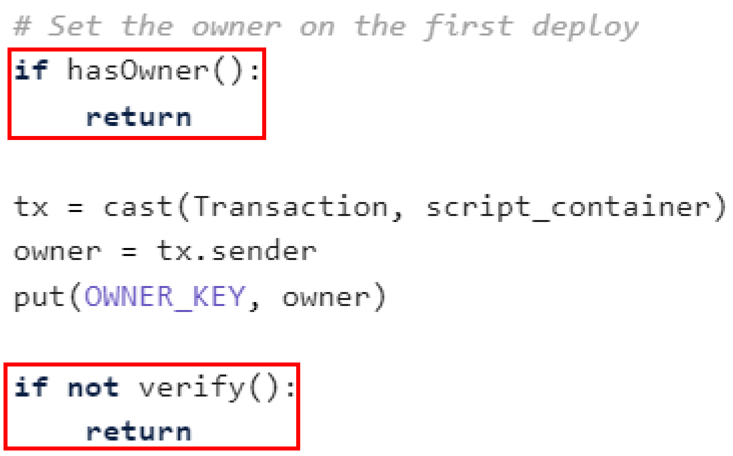
```
protected internal bool CheckWitnessInternal(UInt160 hash)
{
    if (hash.Equals(CallingScriptHash)) return true;
```

Therefore, it is redundant to make this verification and it increases the cost of gas for the execution.

Source reference

- src/LyrebirdToken.py#lines-317
- src/LyrebirdToken.py#lines-386
- src/LyrebirdToken.py#lines-448
- src/LyrebirdToken.py#lines-582

During the execution of the *_deploy* method, a check of the *hasOwner* is performed and it also checks that the owner signed the transaction, these verifications result unnecessary since the method can only be executed during the contract's deployment, so the owner will never be established, thus there is no need for an identity verification.



```
# Set the owner on the first deploy
if hasOwner():
    return

tx = cast(Transaction, script_container)
owner = tx.sender
put(OWNER_KEY, owner)

if not verify():
    return
```

Source reference

- src/LyrebirdToken.py#lines-696:704

Unnecessary on-chain Metrics

The *NUM_ACCOUNTS_KEY* and *MINTED_KEY* storage keys are intended to store metrics of the contract usage, however this information is unnecessary for the proper functioning of the contract and it should not be stored on-chain, since it increases the GAS and storage consumption. These statistics can be easily resolved by querying the nodes of the blockchain.

Therefore, it is recommended to eliminate both the storage keys and the query methods associated with them, such as: *numAccounts*, *numStaking* and *totalMinted*.

Source reference

- src/LyrebirdToken.py#lines-47
- src/LyrebirdToken.py#lines-50:51
- src/LyrebirdToken.py#lines-170:173
- src/LyrebirdToken.py#lines-185:192

Fixes Review

This issue has been partially fixed in the following commit:

- lyrebird-contract/commits/116efc2c2970b21a150a9ee042e8aa5a00c436e8

LyreBird Hatchery Issues

LBD24 – Limit Call Rights (Low)

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, a reuse or a signature scope; so, it is advisable to use the Principle of Least Privilege (PoLP)[7] during all the external processes or during the calls to contracts.

Therefore, when making the call to any contract it is expected to be read-only; as it is the case of obtaining the user's balance, this call should always be made with the *READ_ONLY* flag instead of *CallFlags.ALL*.

```
call_contract(token, 'balanceOf', [ executing_script_hash ], CallFlags.READ_ONLY)
```

Source reference

- src/LyrebirdHatchery.py#lines-97

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/b149dd200d8638e60a1759a12fc3f3d0e6b51f02

LBD25 – Lack of Token WhiteList (Low)

The *OnNEP17Payment* method of the **LyrebirdHatchery** contract does not limit the tokens received, this allows to send any NEP-17 token to the contract, which can facilitate phishing attacks through a malicious token.

If the *owner* decides to send the tokens from the malicious contract to another account, and it uses a *Scope* of signatures from *Global* (or similar), the malicious contract could steal funds from the account by reusing the signature.

[7] https://en.wikipedia.org/wiki/Principle_of_least_privilege

Even though the *Scope* prevents these situations, it is always advised to reduce the exposure surface. Adding a whitelist of allowed tokens is a good practice and secures our contracts and the users.

Source reference

- src/LyrebirdHatchery.py#lines-122

Fixes Review

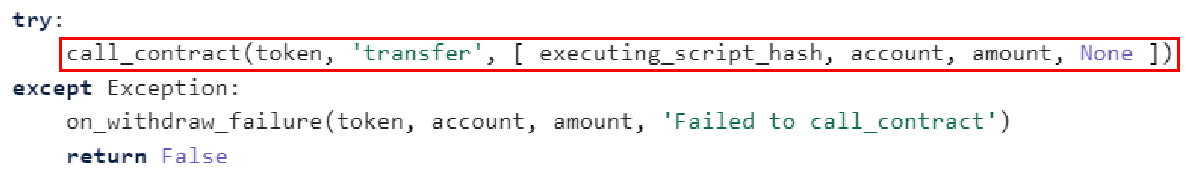
This issue has been fixed in the following commit:

- lyrebird-contract/commits/826e75ce79985bd4e39ba0353837df1f9f5d593a

LBD26 – Unchecked Transfer Return (Low)

The NEP17[8] standard specifies that the transfer functions will return a boolean with the result of this operation.

The ***withdraw*** method from the **LyrebirdHatchery** contract does not contemplate this result, although some NEP-17 token implementations make a revert if these methods fail, the standard dictates that cases such as; not enough balance or wrong signatures, should return a false. This is the case for NEO, GAS and Lyrebird tokens.



```
try:
    call_contract(token, 'transfer', [ executing_script_hash, account, amount, None ])
except Exception:
    on_withdraw_failure(token, account, amount, 'Failed to call_contract')
    return False
```

The result of external contract calls should always be verified and it is mandatory to check that the returned value is *true* in all of the transfers.

Source reference

- src/LyrebirdHatchery.py#lines-103

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/c5910d923c2de2c7b513a6351c6f5e132f586dde

[8] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki#transfer

LBD27 – 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 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.

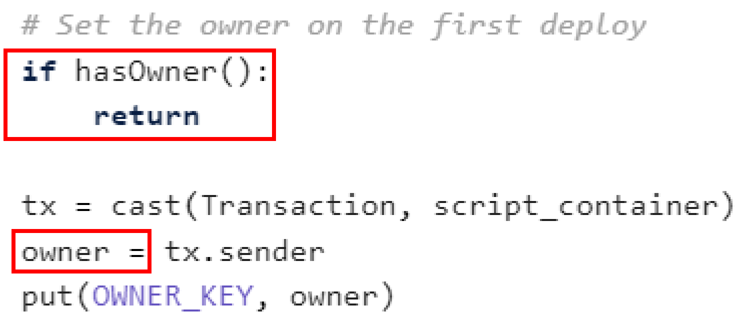
These are some of the requirements that must be met to reduce GAS consumption:

- 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.

Unnecessary logic

During the execution of the *_deploy* method, a check of the *hasOwner* is performed and it also checks that the owner signed the transaction, these verifications result unnecessary since the method can only be executed during the contract's deployment, so the owner will never be established, thus there is no need for an identity verification.

Likewise, the *owner* variable can be eliminated and directly use tx.sender to establish the value in *OWNER_KEY.*



```
# Set the owner on the first deploy
if hasOwner():
    return

tx = cast(Transaction, script_container)
owner = tx.sender
put(OWNER_KEY, owner)
```

Source reference

- src/LyrebirdHatchery.py#lines-160

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/83ae8f39bbd21a652833f3bbda7d3b7c9c0d390b

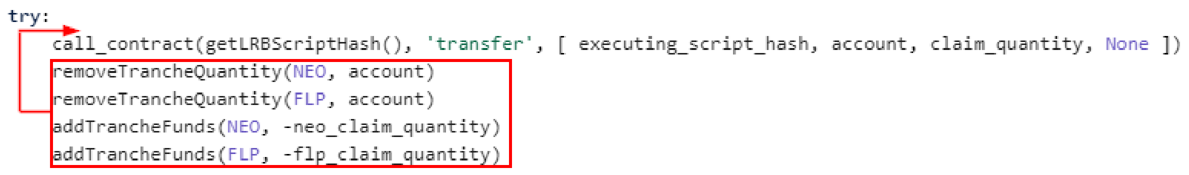
LyreBird Cage Issues

LBD28 – Drain of Funds by Reentrancy (Critical)

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[9] and NEP11[10] 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.

Therefore, 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.



```
try:
    call_contract(getLRBScriptHash(), 'transfer', [ executing_script_hash, account, claim_quantity, None ])
    removeTrancheQuantity(NEO, account)
    removeTrancheQuantity(FLP, account)
    addTrancheFunds(NEO, -neo_claim_quantity)
    addTrancheFunds(FLP, -flp_claim_quantity)
```

In the case of the *claim* method of the **LyrebirdCage** contract, the values are updated after the transfers are made so the call can be redirected to the *claim* method before updating the values. In this case it can be used to drain the funds of the contract, as can be seen in the following image, it is necessary to call the *remove* and *add tranches* methods before calling external contracts.

Source reference

- src/LyrebirdCage.py#lines-247:284

[9] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki
[10] https://github.com/neo-project/proposals/blob/master/nep-11.mediawiki

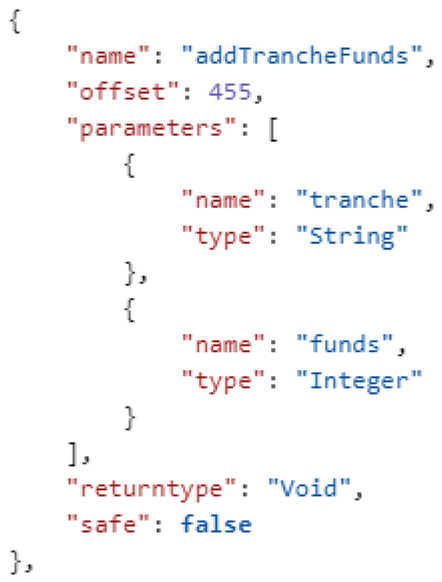
Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/a6698f7806138cacb2018b9617de47232fd55e76

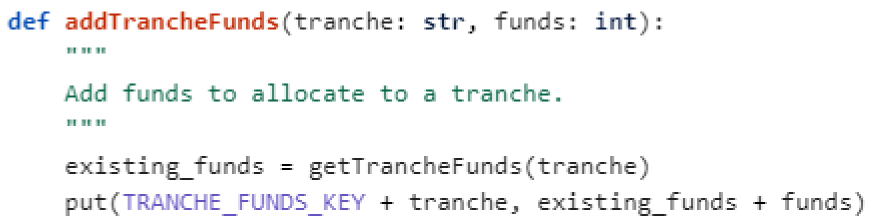
LBD29 – Exposed Private Method (High)

It has been detected that the *addTrancheFunds* method is exposed in the *manifest* file of the **LyrebirdCage** contract, as can be seen in the following image.



```
{
    "name": "addTrancheFunds",
    "offset": 455,
    "parameters": [
        {
            "name": "tranche",
            "type": "String"
        },
        {
            "name": "funds",
            "type": "Integer"
        }
    ],
    "returntype": "Void",
    "safe": false
},
```

This means that although its visibility in the contract is not public, the method will be published once the deployment is carried out, as stated in the manifest. This feature allows any user to call the *addTrancheFunds* method, adding funds to a tranche without actually making the deposit, breaking the proposed business logic.



```
def addTrancheFunds(tranche: str, funds: int):
    """
    Add funds to allocate to a tranche.
    """
    existing_funds = getTrancheFunds(tranche)
    put(TRANCHE_FUNDS_KEY + tranche, existing_funds + funds)
```

Additionally, it is recommended not to store the own binaries of the compiled contracts together with the source code to avoid this type of errors and it should be essential to keep them synchronized with the current version of the code.

Source reference

- src/LyrebirdCage.manifest.json#lines-144:157
- src/LyrebirdCage.py#lines-200:205

Fixes Review

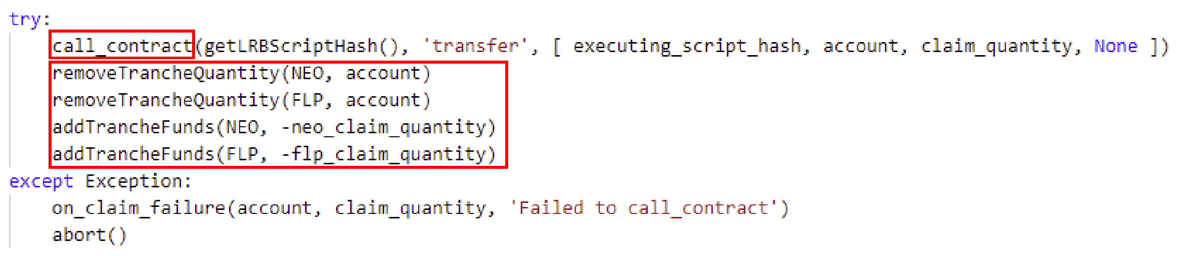
This issue has been fixed in the following commit:

- lyrebird-contract/commits/7edd6ebc32b2e1d34726a506d47167f0e24762da

LBD30 – Unchecked Transfer Return (High)

The NEP17[11] standard specifies that the transfer functions will return a boolean with the result of this operation.

The **LyrebirdCatch** contract does not contemplate this result, although some NEP-17 token implementations make a revert if these methods fail, the standard dictates that cases such as; not enough balance or wrong signatures, should return a false. This is the case for NEO, GAS and Lyrebird tokens.



```
try:
    call_contract(getLRBScriptHash(), 'transfer', [ executing_script_hash, account, claim_quantity, None ])
    removeTrancheQuantity(NEO, account)
    removeTrancheQuantity(FLP, account)
    addTrancheFunds(NEO, -neo_claim_quantity)
    addTrancheFunds(FLP, -flp_claim_quantity)
except Exception:
    on_claim_failure(account, claim_quantity, 'Failed to call_contract')
    abort()
```

The ***claim*** method does not guarantee that the contract has previous funds. So, even if the user makes a legitimate claim, the smart contract may not have any funds, therefore it is important to control the result of the *return* from the *transfer* method to proceed with an abort in case it fails.

[11] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki#transfer

The result of external contract calls should always be verified and It is mandatory to check that the returned value is *true* in all of the transfers.

Source reference

- src/LyrebirdCage.py#lines-247:284

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/a6698f7806138cacb2018b9617de47232fd55e76

LBD31 – Lack of Inputs Validation (Medium)

Some methods of the different contracts in the **LyrebirdCage** contract do not properly check the arguments, which can lead to major errors.

It is advisable to always check the format and type 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.

It is important to know that the `type` of the arguments from an invocation are not natively verified by NEO VM and that neo3-boa does not add any type of verification during its compilation, therefore, all the arguments must be carefully verified.

The integrity of the `UInt160` types is only verified by its size (20 bytes) however, it is more convenient to use the *isinstance()* method since it makes verifications by type and size.

Additionally, in some methods it is convenient to check that the address value is not address `zero`, which could lead to unwanted behaviours in certain occasions.

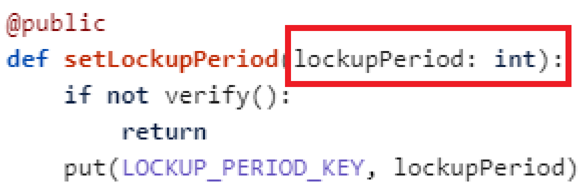
It is convenient to add a method that unifies the format verification of the addresses, as suggested below:

```python
def validateAddress(address: UInt160) -> bool:
    if not isinstance(address, UInt160):
        return False
    if address == 0:
        return False
    return True
```

Source reference

- src/LyrebirdCage.py#lines-99:102
- src/LyrebirdCage.py#lines-128:134
- src/LyrebirdCage.py#lines-208:228
- src/LyrebirdCage.py#lines-230:244

Besides the verifications of types for *UInt160* arguments, it must be mentioned that there are three methods where the integer arguments are not checked to be greater than 0, *setLockupPeriod, addTrancheFunds* and *setTrancheQuantities*, in all the methods the amount received should be checked to be positive.

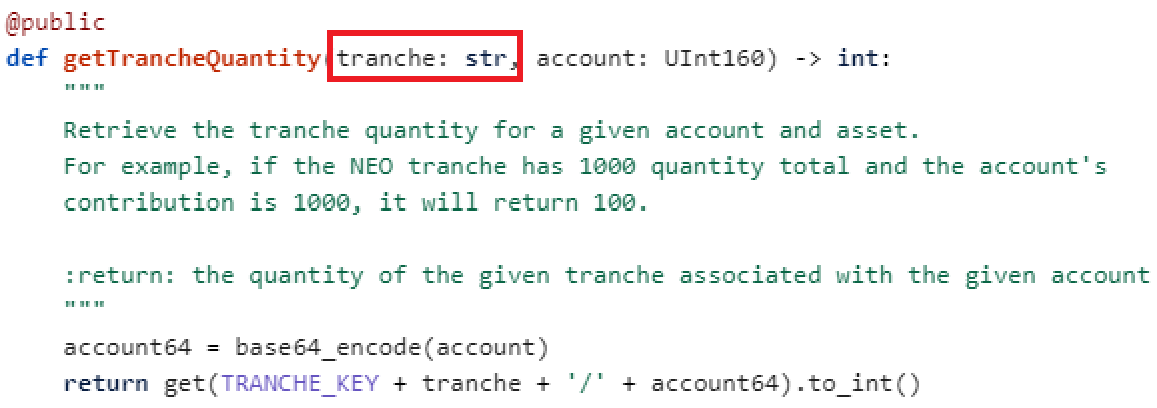


```
@public
def setLockupPeriod(lockupPeriod: int):
    if not verify():
        return
    put(LOCKUP_PERIOD_KEY, lockupPeriod)
```

Source reference

- src/LyrebirdCage.py#lines-115:119
- src/LyrebirdCage.py#lines-200:205
- src/LyrebirdCage.py#lines-208:227

Finally, since the value of *tranche* can only be NEO or FLP, it is important to verify that the value entered, in the methods that require it, is as expected in order to avoid unnecessary executions and possible logic failures.



```
@public
def getTrancheQuantity(tranche: str, account: UInt160) -> int:
    """
    Retrieve the tranche quantity for a given account and asset.
    For example, if the NEO tranche has 1000 quantity total and the account's
    contribution is 1000, it will return 100.

    :return: the quantity of the given tranche associated with the given account
    """
    account64 = base64_encode(account)
    return get(TRANCHE_KEY + tranche + '/' + account64).to_int()
```

Source reference

- src/LyrebirdCage.py#lines-137:147
- src/LyrebirdCage.py#lines-200:205
- src/LyrebirdCage.py#lines-208:227
- src/LyrebirdCage.py#lines-230:244

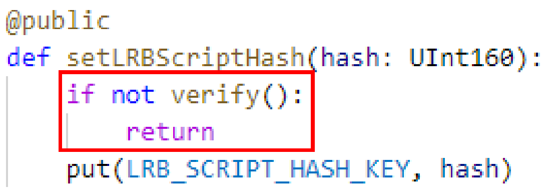
Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/bbdabca06315c804de5d73a21e2d507a716c0aaa

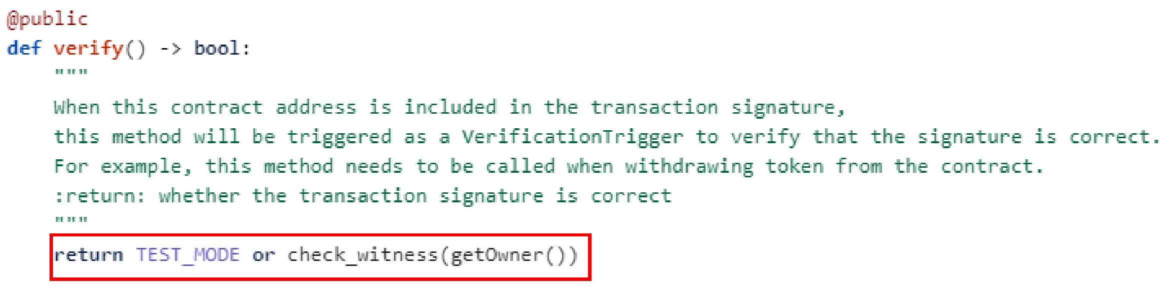
LBD32 – Lack of Abort on missing authorization (Low)

Throughout the **LyrebirdCage** contract, several calls are made to the *verify* method to authenticate the identity of the owner.



```
@public
def setLRBScriptHash(hash: UInt160):
    if not verify():
        return
    put(LRB_SCRIPT_HASH_KEY, hash)
```

This method makes a *check_witness*[12] in order to verify if the sender is the *owner*, returning a boolean with the result of whether the script hash has been verified or not.

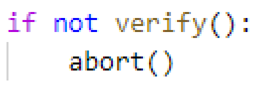


```
@public
def verify() -> bool:
    """
    When this contract address is included in the transaction signature,
    this method will be triggered as a VerificationTrigger to verify that the signature is correct.
    For example, this method needs to be called when withdrawing token from the contract.
    :return: whether the transaction signature is correct
    """
    return TEST_MODE or check_witness(getOwner())
```

The problem is that when the *verify* method is unable to verify the owner's identity it returns a *false* but fails to abort the execution. So, it is recommended

[12] https://dojo.coz.io/neo3/boa/boa3/builtin/interop/runtime/boa3-builtin-interop-runtime.html?highlight=check_witness#boa3.builtin.interop.runtime.check_witness

to replace it with an *Abort*, since it would stop the execution of the Smart Contract and properly inform the user.



```
if not verify():
    abort()
```

This may not seem like a relevant issue, but this bad practice can trigger other important problems in the Smart Contract.

Source reference

- src/LyrebirdCage.py#lines-98:102
- src/LyrebirdCage.py#lines-117:118
- src/LyrebirdCage.py#lines-130:131
- src/LyrebirdCage.py#lines-217:218
- src/LyrebirdCage.py#lines-238:239
- src/LyrebirdCage.py#lines-238:239
- src/LyrebirdCage.py#lines-258:260

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/5b2d2e98cf2964395c84ee38c2e34bbdd86940e8

LBD33 – 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 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.

Unused code

The *on_whitelist* event and the *CLAIMABLE_KEY* constant are not used throughout the **LyrebirdCage** contract. NeoBoa compiles all of the constants as static variables, so the cost of execution increases since they are initialized during the call to the *_initialize* method.

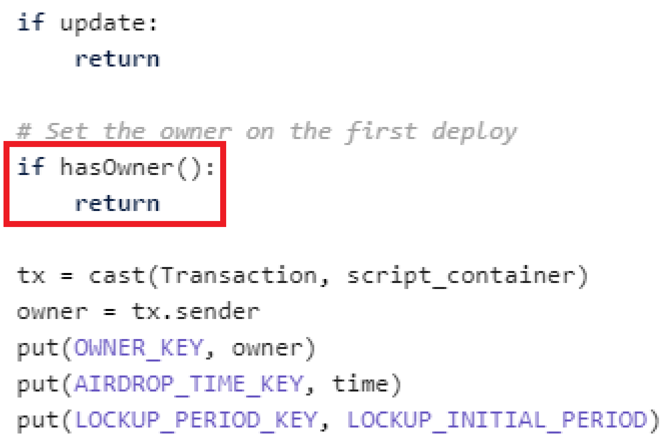
Source reference

- src/LyrebirdCage.py#lines-40
- src/LyrebirdCage.py#lines-65:70

Unnecessary logic

During the execution of the *_deploy* method, a check of the *hasOwner* is performed and it also checks that the owner signed the transaction, these verifications result unnecessary since the method can only be executed during the contract's deployment, so the owner will never be established, thus there is no need for an identity verification.

Likewise, the *owner* variable can be eliminated and directly use tx.sender to establish the value in *OWNER_KEY.*



```
if update:
    return

# Set the owner on the first deploy
if hasOwner():
    return

tx = cast(Transaction, script_container)
owner = tx.sender
put(OWNER_KEY, owner)
put(AIRDROP_TIME_KEY, time)
put(LOCKUP_PERIOD_KEY, LOCKUP_INITIAL_PERIOD)
```

Source reference

- src/LyrebirdCage.py#lines-336:354

Fixes Review

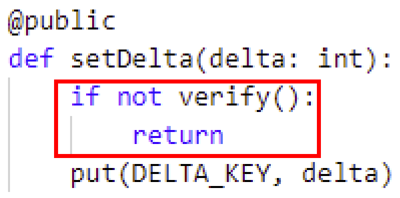
This issue has been fixed in the following commit:

- lyrebird-contract/commits/20f519a9f229bfc0d6323b38807bad397d7c40c9

LyreBird Aviary Issues

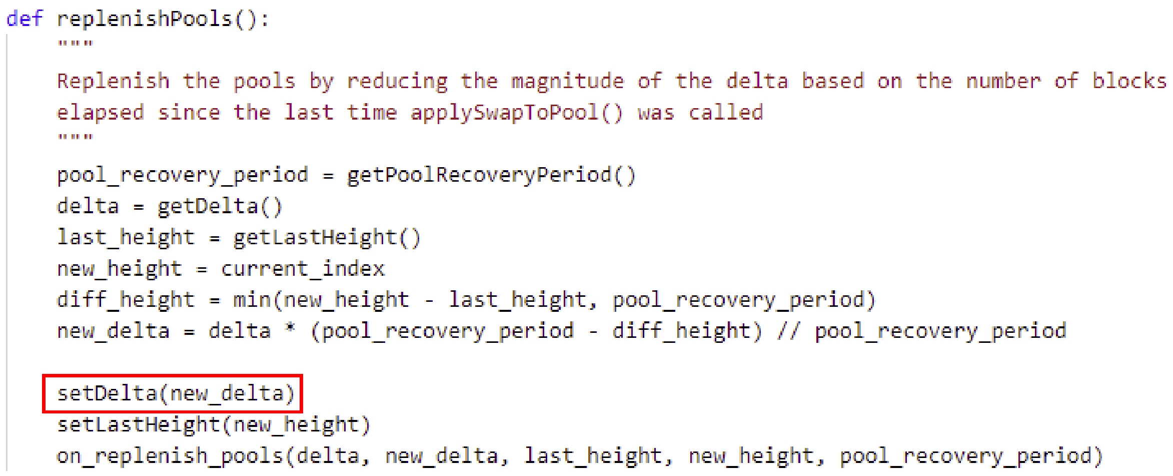
LBD34 – Broken delta update logic (Critical)

The *setDelta* method is responsible for updating the Delta registry in the storage. It contains an invocation to *verify()*, so in order to be effective it should be invoked by the owner.



```
@public
def setDelta(delta: int):
    if not verify():
        return
    put(DELTA_KEY, delta)
```

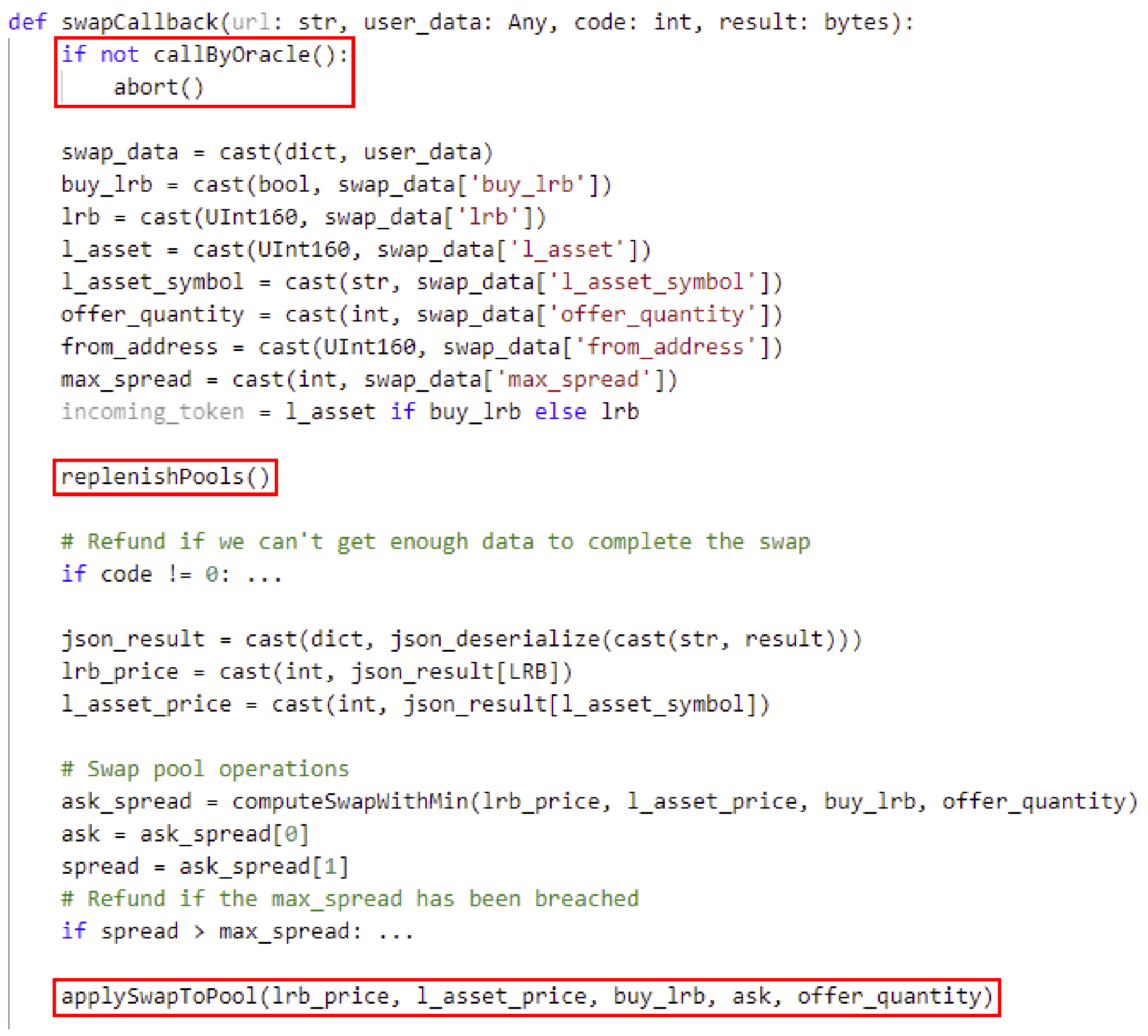
This requirement keeps the logic of *replenishPools*, *applySwapToPool* and *swapCallback* methods from properly completing, since it is unable to update the delta when used by the platform users.



```
def replenishPools():
    """
    Replenish the pools by reducing the magnitude of the delta based on the number of blocks
    elapsed since the last time applySwapToPool() was called
    """
    pool_recovery_period = getPoolRecoveryPeriod()
    delta = getDelta()
    last_height = getLastHeight()
    new_height = current_index
    diff_height = min(new_height - last_height, pool_recovery_period)
    new_delta = delta * (pool_recovery_period - diff_height) // pool_recovery_period

    setDelta(new_delta)
    setLastHeight(new_height)
    on_replenish_pools(delta, new_delta, last_height, new_height, pool_recovery_period)
```

According to the current behaviour of the **LyrebirdAviary** contract, these invocations should proceed from the oracle's callback. Therefore, it neither contains the user's signature nor the *owner*.



```
def swapCallback(url: str, user_data: Any, code: int, result: bytes):
    if not callByOracle():
        abort()

    swap_data = cast(dict, user_data)
    buy_lrb = cast(bool, swap_data['buy_lrb'])
    lrb = cast(UInt160, swap_data['lrb'])
    l_asset = cast(UInt160, swap_data['l_asset'])
    l_asset_symbol = cast(str, swap_data['l_asset_symbol'])
    offer_quantity = cast(int, swap_data['offer_quantity'])
    from_address = cast(UInt160, swap_data['from_address'])
    max_spread = cast(int, swap_data['max_spread'])
    incoming_token = l_asset if buy_lrb else lrb

    replenishPools()

    # Refund if we can't get enough data to complete the swap
    if code != 0: ...

    json_result = cast(dict, json_deserialize(cast(str, result)))
    lrb_price = cast(int, json_result[LRB])
    l_asset_price = cast(int, json_result[l_asset_symbol])

    # Swap pool operations
    ask_spread = computeSwapWithMin(lrb_price, l_asset_price, buy_lrb, offer_quantity)
    ask = ask_spread[0]
    spread = ask_spread[1]
    # Refund if the max_spread has been breached
    if spread > max_spread: ...

    applySwapToPool(lrb_price, l_asset_price, buy_lrb, ask, offer_quantity)
```

By being unable to update the values among the different swaps, the behaviour becomes erratic with values that do not reflect the real value of the pools.

Source reference

- src/LyrebirdAviary.py#lines-255
- src/LyrebirdAviary.py#lines-447
- src/LyrebirdAviary.py#lines-548
- src/LyrebirdAviary.py#lines-630
- src/LyrebirdAviary.py#lines-652

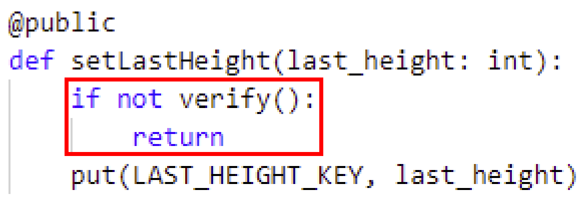
Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/ed23d70b7fc96c3e9c6741035d9530b44c4cc30b

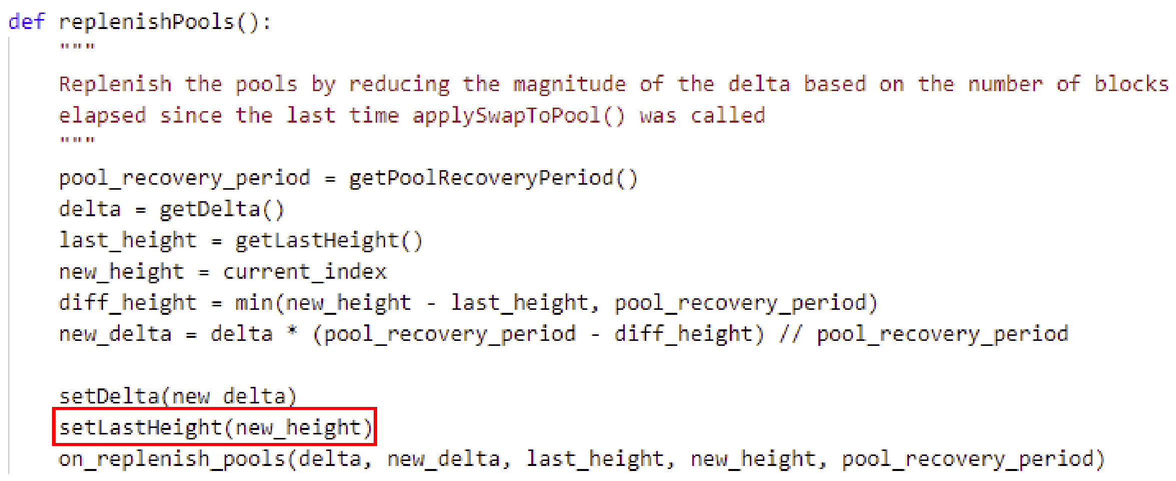
LBD35 – Broken Last Height update logic (Critical)

The *setLastHeight* method, responsible for saving the number of the block from the last *replenish pools*, contains an invocation to *verify(),* so it should be invoked by the owner in order to be effective.



```
@public
def setLastHeight(last_height: int):
    if not verify():
        return
    put(LAST_HEIGHT_KEY, last_height)
```

This requirement keeps the logic of the *replenishPools* and *swapCallback* methods from properly completing, since it is unable to update the number of used blocks in the calculations of the *pool_recovery_period* and *delta* values.



```
def replenishPools():
    """
    Replenish the pools by reducing the magnitude of the delta based on the number of blocks
    elapsed since the last time applySwapToPool() was called
    """
    pool_recovery_period = getPoolRecoveryPeriod()
    delta = getDelta()
    last_height = getLastHeight()
    new_height = current_index
    diff_height = min(new_height - last_height, pool_recovery_period)
    new_delta = delta * (pool_recovery_period - diff_height) // pool_recovery_period

    setDelta(new_delta)
    setLastHeight(new_height)
    on_replenish_pools(delta, new_delta, last_height, new_height, pool_recovery_period)
```

By being unable to update the values among the different swaps, the behaviour becomes erratic with values that do not reflect the real value of the pools.

Source reference

- src/LyrebirdAviary.py#lines-324
- src/LyrebirdAviary.py#lines-448
- src/LyrebirdAviary.py#lines-630

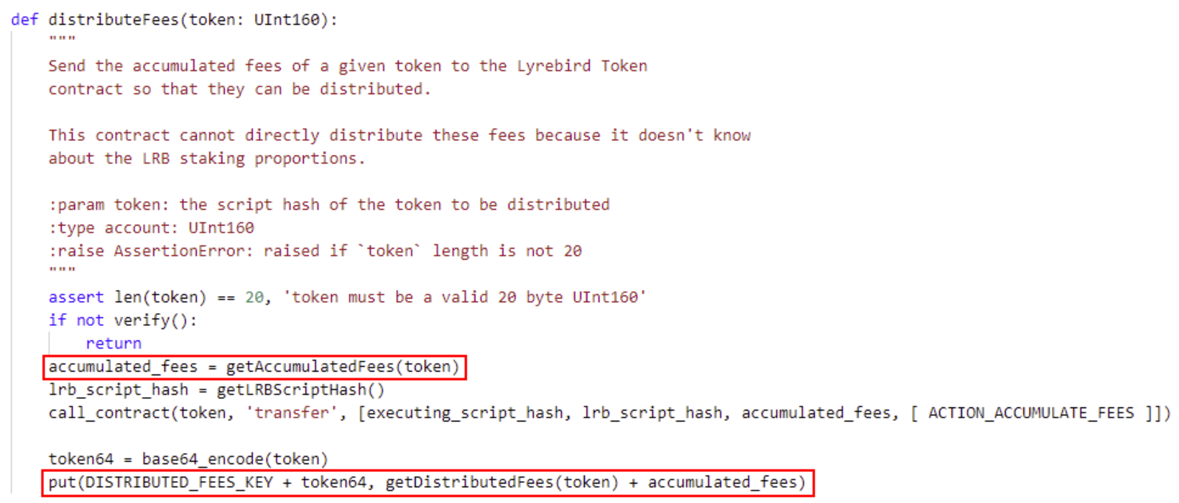
Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/ed23d70b7fc96c3e9c6741035d9530b44c4cc30b

LBD36 – Broken distributeFees Logi (Critical)

The *distributeFees* method, responsible for sending the piled fees to the LRB contract, does not deduct the number of tokens currently distributed before sending the fees.



```
def distributeFees(token: UInt160):
    """
    Send the accumulated fees of a given token to the Lyrebird Token
    contract so that they can be distributed.

    This contract cannot directly distribute these fees because it doesn't know
    about the LRB staking proportions.

    :param token: the script hash of the token to be distributed
    :type account: UInt160
    :raise AssertionError: raised if `token` length is not 20
    """
    assert len(token) == 20, 'token must be a valid 20 byte UInt160'
    if not verify():
        return
    accumulated_fees = getAccumulatedFees(token)
    lrb_script_hash = getLRBScriptHash()
    call_contract(token, 'transfer', [executing_script_hash, lrb_script_hash, accumulated_fees, [ ACTION_ACCUMULATE_FEES ]])

    token64 = base64_encode(token)
    put(DISTRIBUTED_FEES_KEY + token64, getDistributedFees(token) + accumulated_fees)
```

The *getAccumulatedFees* method obtains the value of the *ACCUMULATED_FEES_KEY* storage and this amount is stored at the end of the *distributeFees* function in the *DISTRIBUTED_FEES_KEY* storage. However, at no point is it deducted from the value to distribute, so the method is able to invoke multiple times and distribute a higher amount than initially collected.

Source reference

- src/LyrebirdAviary.py#lines-353

Fixes Review

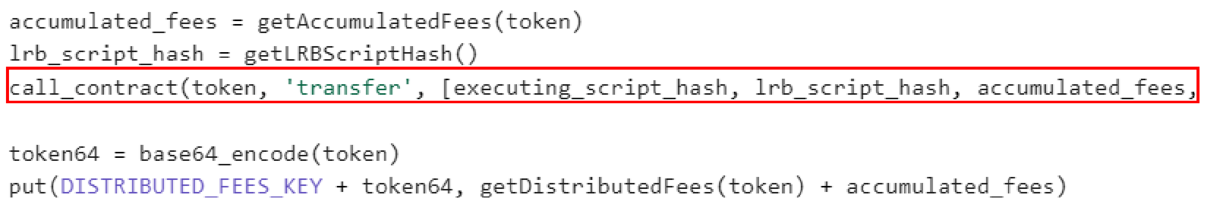
This issue has been fixed in the following commit:

- lyrebird-contract/commits/eaf4273a27b424fc52b883816b9d33dbcd07de22

LBD37 – Unchecked Transfer Return (High)

The NEP17[13] standard specifies that the transfer functions will return a boolean with the result of this operation.

The **LyrebirdAviary** contract does not contemplate this result, although some NEP-17 token implementations make a revert if these methods fail, the standard dictates that cases such as; not enough balance or wrong signatures, should return a false. This is the case for NEO, GAS and Lyrebird tokens.



```
accumulated_fees = getAccumulatedFees(token)
lrb_script_hash = getLRBScriptHash()
call_contract(token, 'transfer', [executing_script_hash, lrb_script_hash, accumulated_fees,

token64 = base64_encode(token)
put(DISTRIBUTED_FEES_KEY + token64, getDistributedFees(token) + accumulated_fees)
```

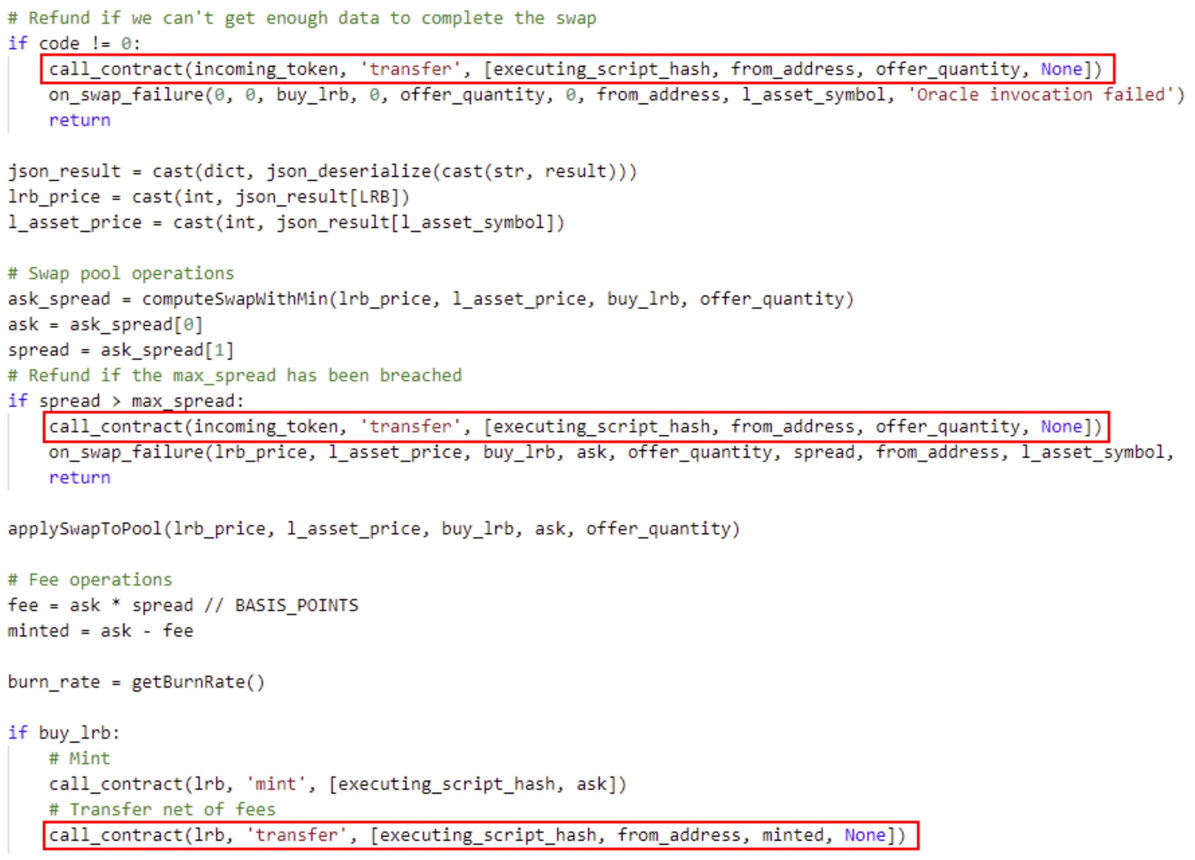
The ***distributeFees*** and ***fundHatchery*** methods do not guarantee that the contract has enough funds and if the transfer fails, before updating, the distributed fees will be registered.

Source reference

- src/LyrebirdAviary.py#lines-370
- src/LyrebirdAviary.py#lines-404

This lack of verifications can also be seen throughout the *swapCallback* method and should be fixed.

[13] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki#transfer



```
# Refund if we can't get enough data to complete the swap
if code != 0:
    call_contract(incoming_token, 'transfer', [executing_script_hash, from_address, offer_quantity, None])
    on_swap_failure(0, 0, buy_lrb, 0, offer_quantity, 0, from_address, l_asset_symbol, 'Oracle invocation failed')
    return

json_result = cast(dict, json_deserialize(cast(str, result)))
lrb_price = cast(int, json_result[LRB])
l_asset_price = cast(int, json_result[l_asset_symbol])

# Swap pool operations
ask_spread = computeSwapWithMin(lrb_price, l_asset_price, buy_lrb, offer_quantity)
ask = ask_spread[0]
spread = ask_spread[1]
# Refund if the max_spread has been breached
if spread > max_spread:
    call_contract(incoming_token, 'transfer', [executing_script_hash, from_address, offer_quantity, None])
    on_swap_failure(lrb_price, l_asset_price, buy_lrb, ask, offer_quantity, spread, from_address, l_asset_symbol,
    return

applySwapToPool(lrb_price, l_asset_price, buy_lrb, ask, offer_quantity)

# Fee operations
fee = ask * spread // BASIS_POINTS
minted = ask - fee

burn_rate = getBurnRate()

if buy_lrb:
    # Mint
    call_contract(lrb, 'mint', [executing_script_hash, ask])
    # Transfer net of fees
    call_contract(lrb, 'transfer', [executing_script_hash, from_address, minted, None])
```

The result of external contract calls should always be verified and It is mandatory to check that the returned value is *true* in all of the transfers.

Source reference

- src/LyrebirdAviary.py#lines-634
- src/LyrebirdAviary.py#lines-648
- src/LyrebirdAviary.py#lines-664
- src/LyrebirdAviary.py#lines-675

Fixes Review

This issue has been fixed in the following commit:

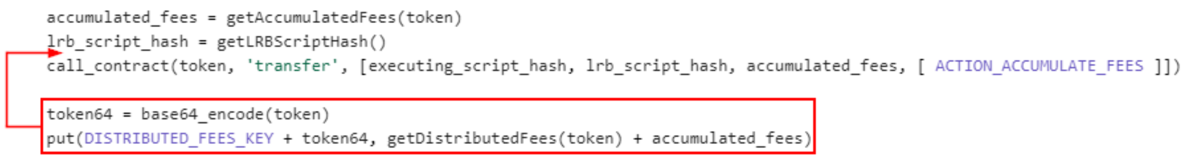
- lyrebird-contract/commits/17acfa9a46f8a5235e0dc7d4331ed69ab15ecd77

LBD38 – DistributeFees Reentrancy (High)

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[14] and NEP11[15] 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.

Therefore, 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.



```
accumulated_fees = getAccumulatedFees(token)
lrb_script_hash = getLRBScriptHash()
call_contract(token, 'transfer', [executing_script_hash, lrb_script_hash, accumulated_fees, [ ACTION_ACCUMULATE_FEES ]])

token64 = base64_encode(token)
put(DISTRIBUTED_FEES_KEY + token64, getDistributedFees(token) + accumulated_fees)
```

In the case of the *distributeFees* method of the **LyrebirdAviary** contract, the values are updated after the transfers are made so the call can be redirected to the *distributeFees* method before updating the values. In this case it can be used to distribute infinite fees of the contract, as can be seen in the image, it is necessary to *put* into storage the distributed fees before calling external contracts.

The risk of this vulnerability has been reduced from critical to high, since it is required to be an owner.

Source reference

- src/LyrebirdAviary.py#lines-352:373

[14] https://github.com/neo-project/proposals/blob/master/nep-17.mediawiki
[15] https://github.com/neo-project/proposals/blob/master/nep-11.mediawiki

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/22edabd0efc3f2b500c7eb8ab8e0afc88a02e0cb

LBD39 – Lack of Inputs Validation (Medium)

Some methods of the different contracts in the **Lyrebird Token** contract do not properly check the arguments, which can lead to major errors.

It is advisable to always check the format and type 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.

It is important to know that the `type` of the arguments from an invocation are not natively verified by NEO VM and that neo3-boa does not add any type of verification during its compilation, therefore, all the arguments must be carefully verified.

The integrity of the `UInt160` types is only verified by its size (20 bytes) however, it is more convenient to use the *isinstance()* method since it makes verifications by type and size.

Additionally, in some methods it is convenient to check that the address value is not address `zero`, which could lead to unwanted behaviours in certain occasions.

It is convenient to add a method that unifies the format verification of the addresses, as suggested below:

```python
def validateAddress(address: UInt160) -> bool:
    if not isinstance(address, UInt160):
        return False
    if address == 0:
        return False
    return True
```

Source reference

- src/LyrebirdAviary.py#lines-196
- src/LyrebirdAviary.py#lines-204
- src/LyrebirdAviary.py#lines-210
- src/LyrebirdAviary.py#lines-218
- src/LyrebirdAviary.py#lines-231
- src/LyrebirdAviary.py#lines-330
- src/LyrebirdAviary.py#lines-347
- src/LyrebirdAviary.py#lines-353
- src/LyrebirdAviary.py#lines-377
- src/LyrebirdAviary.py#lines-553

Besides the verifications of types for *UInt160* arguments, it must be mentioned that there are three methods where the integer arguments are not checked to be greater than *0*, *setOracleFee, setDelta, setMinSpread, setBasePool, setPoolRecoveryPeriod* and *setBurnRate*, in all the methods the amount received should be checked to be positive.

Source reference

- src/LyrebirdAviary.py#lines-243
- src/LyrebirdAviary.py#lines-267
- src/LyrebirdAviary.py#lines-279
- src/LyrebirdAviary.py#lines-291
- src/LyrebirdAviary.py#lines-303
- src/LyrebirdAviary.py#lines-553
    - amount, max_spread, and data must be different than null.

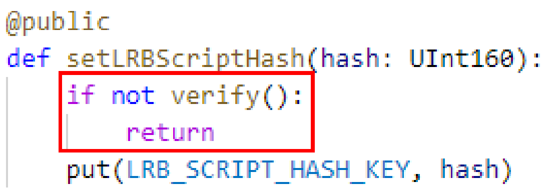
Fixes Review

This issue has been partially fixed in the following commit:

- lyrebird-contract/commits/7eefb4f7ba153b4ba7b42c7448de5f5b3d00c602

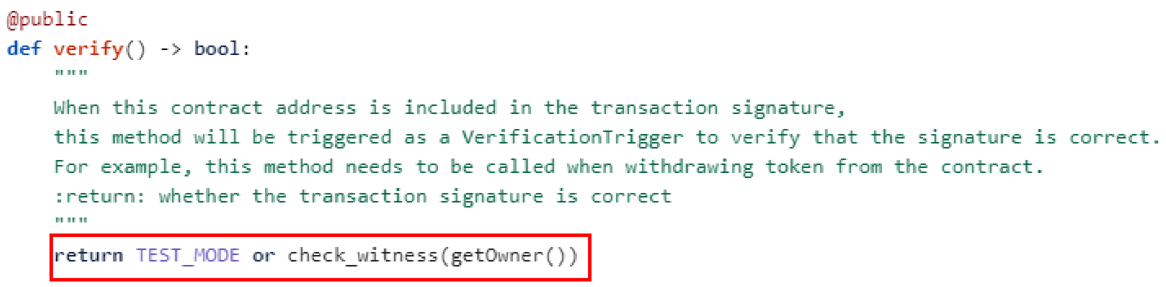
LBD40 – Lack of Abort on missing authorization (Medium)

Throughout the **LyrebirdAviary** contract, several calls are made to the *verify* method to authenticate the identity of the owner.



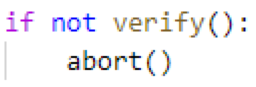
```
@public
def setLRBScriptHash(hash: UInt160):
    if not verify():
        return
    put(LRB_SCRIPT_HASH_KEY, hash)
```

This method makes a *check_witness*[16] in order to verify if the sender is the *owner*, returning a boolean with the result of whether the script hash has been verified or not.



```
@public
def verify() -> bool:
    """
    When this contract address is included in the transaction signature,
    this method will be triggered as a VerificationTrigger to verify that the signature is correct.
    For example, this method needs to be called when withdrawing token from the contract.
    :return: whether the transaction signature is correct
    """
    return TEST_MODE or check_witness(getOwner())
```

The problem is that when the *verify* method is unable to verify the owner's identity it returns a *false* but fails to abort the execution. So, it is recommended to replace it with an *Abort*, since it would stop the execution of the Smart Contract and properly inform the user.



```
if not verify():
    abort()
```

This may not seem like a relevant issue, but this bad practice can trigger other important problems in the Smart Contract, as can be seen in the vulnerabilities "*Broken Last Height update logic*" and "*Broken distributeFees Logic*".

Source reference

- src/LyrebirdAviary.py#lines-197:198

[16] https://dojo.coz.io/neo3/boa/boa3/builtin/interop/runtime/boa3-builtin-interop-runtime.html?highlight=check_witness#boa3.builtin.interop.runtime.check_witness

- src/LyrebirdAviary.py#lines-211:212
- src/LyrebirdAviary.py#lines-219:220
- src/LyrebirdAviary.py#lines-232:233
- src/LyrebirdAviary.py#lines-244:245
- src/LyrebirdAviary.py#lines-256:257
- src/LyrebirdAviary.py#lines-268:269
- src/LyrebirdAviary.py#lines-280:281
- src/LyrebirdAviary.py#lines-292:293
- src/LyrebirdAviary.py#lines-304:305
- src/LyrebirdAviary.py#lines-325:326
- src/LyrebirdAviary.py#lines-366:367
- src/LyrebirdAviary.py#lines-715:716

Fixes Review

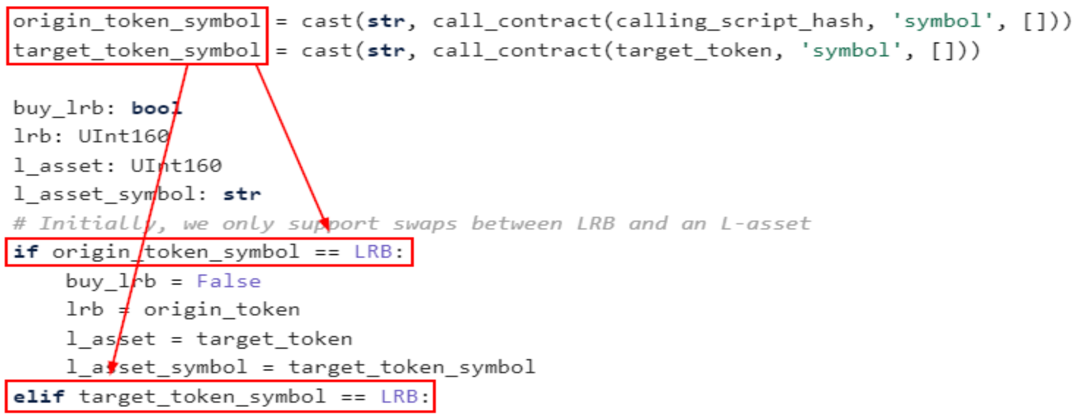
This issue has been fixed in the following commit:

- lyrebird-contract/commits/4c978f85771051addd3b5cb6bc25234cadb74464

LBD41 – Unsafe token detection (Medium)

The **LyrebirdAviary** contract uses the result of an insecure call to detect if it is a LRB token at the source or at destination, this result can be forged and return a different *symbol*, even though the contract has to previously be in the Aviary whitelist. It is important to highlight that in N3 the contracts do not modify their hash when updated, so it is recommended to use the hash of the contract to detect if it is a LRB token, otherwise it would facilitate *rug pull*[17] type attacks.

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



```
origin_token_symbol = cast(str, call_contract(calling_script_hash, 'symbol', []))
target_token_symbol = cast(str, call_contract(target_token, 'symbol', []))

buy_lrb: bool
lrb: UInt160
l_asset: UInt160
l_asset_symbol: str
# Initially, we only support swaps between LRB and an L-asset
if origin_token_symbol == LRB:
    buy_lrb = False
    lrb = origin_token
    l_asset = target_token
    l_asset_symbol = target_token_symbol
elif target_token_symbol == LRB:
```

Source reference

- src/LyrebirdAviary.py#lines-579:592

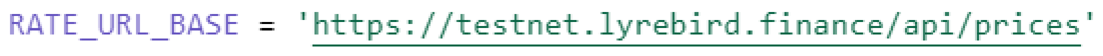
Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/786d698d01d9c2a217e6f30d3ee7e977940951e8

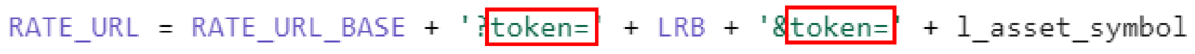
LBD42 – Centralized Oracle Price (Low)

The API used to obtain the prices belongs to the Lyrebird project. It is advisable to use several price endpoints and compare the prices received, in order to obtain possible failures in the price obtained. By having a single point of collection, it is less resilient to errors and connection failures, in addition to price changes made by the person in charge of maintaining the website.



```
RATE_URL_BASE = 'https://testnet.lyrebird.finance/api/prices'
```

A failure in the parameters sent to the API has also been detected, since the two tokens to be compared are sent using the same name *"token"*, which makes it difficult or even impossible for the API to extract the second parameter to compare. Although the website is out of scope, this may indicate that it is vulnerable to Http Parameter Pollution and other vulnerabilities.



```
RATE_URL = RATE_URL_BASE + '?token=' + LRB + '&token=' + l_asset_symbol
```

Source reference

- src/LyrebirdAviary.py#lines-599

LBD43 – Limit Call Rights (Low)

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, a reuse or a signature scope; so, it is advisable to use the Principle of Least Privilege (PoLP)[18] during all the external processes or during the calls to contracts.

Therefore, when making the call to any contract it is expected to be read-only; as it is the case of obtaining the user's balance, this call should always be made with the *READ_ONLY* flag instead of *CallFlags.ALL*.

```
call_contract(token, 'balanceOf', [ executing_script_hash ], CallFlags.READ_ONLY)
```

Source reference

- src/LyrebirdAviary.py#lines-579:580

Fixes Review

This issue has been fixed in the following commit:

- lyrebird-contract/commits/951bd4271342fa988c08c31e9accfac8940f4b40

LBD44 – 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 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:

[18] https://en.wikipedia.org/wiki/Principle_of_least_privilege

- 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.

Unnecessary on-chain Metrics

The *NUM_ACCOUNTS_KEY* and *MINTED_KEY* storage keys are intended to store metrics of the contract usage, however this information is unnecessary for the proper functioning of the contract, and it should not be stored on-chain, since it increases the GAS and storage consumption. These statistics can be easily resolved by querying the nodes of the blockchain.

Therefore, it is recommended to eliminate both the storage keys and the query methods associated with them, such as: *getNumTransactions* and *getValueTransacted*.

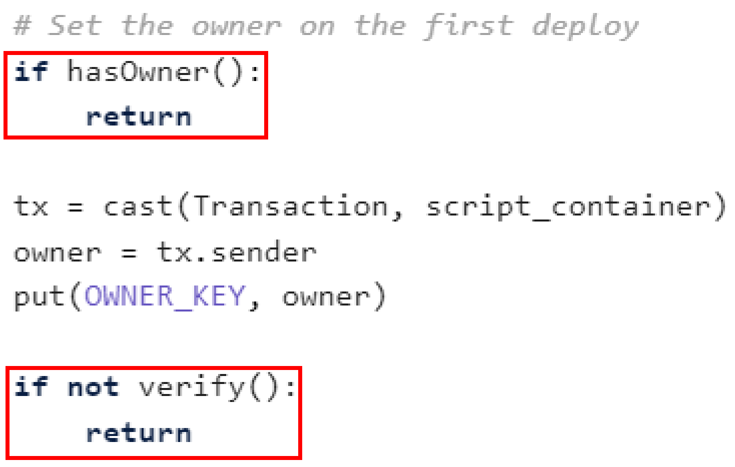
Source reference

- src/LyrebirdAviary.py#lines-383:389
- src/LyrebirdAviary.py#lines-687:688

Unnecessary Logic

During the execution of the *_deploy* method, a check of the *hasOwner* is performed and it also checks that the owner signed the transaction, these verifications result unnecessary since the method can only be executed during the contract's deployment, so the owner will never be established, thus there is no need for an identity verification.

Likewise, the *owner* variable can be eliminated and directly use tx.sender to establish the value in *OWNER_KEY.*



```
# Set the owner on the first deploy
if hasOwner():
    return

tx = cast(Transaction, script_container)
owner = tx.sender
put(OWNER_KEY, owner)

if not verify():
    return
```

Source reference

- src/LyrebirdAviary.py#lines-708:716

Unused code

The *on_transfer* event and the constants *TOKEN_DECIMALS and PRICE_DECIMALS* are not used throughout the **LyrebirdAviary** contract. NeoBoa compiles all of the constants as static variables, so the cost of execution increases since they are initialized during the call to the _*initialize* method.

Source reference

- src/LyrebirdAviary.py#lines-53:55
- src/LyrebirdAviary.py#lines-100

Fixes Review

This issue has been partially fixed in the following commit:

- lyrebird-contract/commits/1b356b83b5398eb04261aa3ef74ba5efdff27a16

RED4SEC

Invest in Security, invest in your future