Security Audit Report

06/24/2022

Genome Mining

Content

Introduction

Altered State Machine is a platform for the creation and training of A.I. Agents, owned and traded using NFTs. ASM aims to be a new generation of metaverses and gaming, where AI agents compete with each other, interact, and support human actors.

Altered State Machine is a decentralized protocol for creating ownership of an AI agent via an NFT. ASM can be used to create Agents for games, financial applications, virtual assistants, online worlds, and many more.

As solicited by **Altered State Machine** 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 **Genome Mining** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of Genome Mining. The performed analysis shows that the smart contract does contain 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 **Genome Mining** 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 **Altered State Machine**:

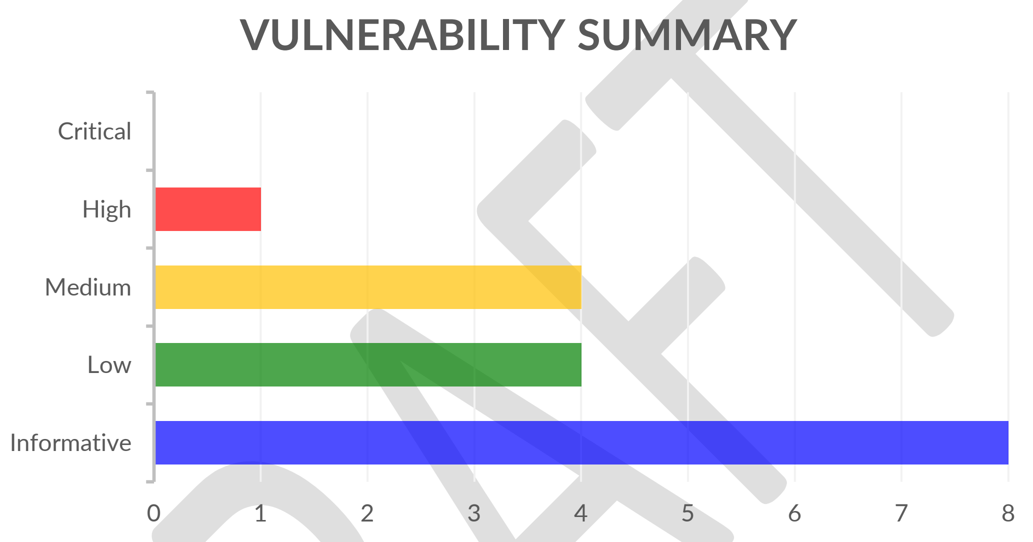
- https://github.com/altered-state-machine/genome-mining-contracts-audit
  - `Commit: 827ed278cb7a2cb93d596198e242b9ca2f5c3d74`

Executive Summary

The security audit against **Genome Mining** has been conducted between the following dates: **06/13/2022** and **06/24/2022**.

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

During the analysis, a total of **17 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Annex A.



Conclusions

To this date, **24/06/2022**, the general conclusion resulting from the conducted audit, is that the **Genome Mining project is not completely secure** and does present some vulnerabilities that could compromise the security of the project.
The general conclusions of the performed audit are:

- **One High-risk vulnerability** has been detected during the security audit. This vulnerability must be fixed as soon as possible.
- Certain methods **do not make the necessary input checks** in order to guarantee the integrity and expected arguments format.
- A **few low impact issues** were detected and classified only as informative, but they will continue to help **Genome Mining Team** improve the security and quality of its developments.
- The overall impression about code quality and organization is not optimal. The developed **code does not comply with code standards** and lacks essential security measures. Red4Sec has given some additional recommendations on how to continue improving and how to apply good practices.
- In order to deal with the detected vulnerabilities, **an action plan must be elaborated to guarantee its resolutio**n, prioritizing those vulnerabilities of greater risk and trying not to exceed the maximum recommended resolution times.

Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered upon the security audit.

List of vulnerabilities

Below, we have gathered a complete list of the vulnerabilities detected by Red4Sec, presented and summarized in a way that can be used for risk management and mitigation.

Table of vulnerabilities			
ID	**Vulnerability**	**Risk**	**State**
GMC-01	Wrong init logic	**High**	**Open**
GMC-02	Bypass not EOA restriction	**Medium**	**Open**
GMC-03	Inheritance design allows constraint bypass	**Medium**	**Open**
GMC-04	Unbounded loop in getHistory and calculateEnergy methods	**Medium**	**Open**
GMC-05	Wrong emitted event	**Medium**	**Open**
GMC-06	Project information leak	**Low**	**Open**
GMC-07	Not compatible with fee-based tokens	**Low**	**Open**
GMC-08	Wrong logic around getController	**Low**	**Open**
GMC-09	Discrepancy with documentation	**Low**	**Open**
GMC-10	Outdated compiler	**Informative**	**Open**
GMC-11	GAS optimization	**Informative**	**Open**
GMC-12	Code style	**Informative**	**Open**
GMC-13	Decentralization recommendation	**Informative**	**Open**
GMC-14	Lack of event index	**Informative**	**Open**
GMC-15	Solidity literals	**Informative**	**Open**
GMC-16	Lack of inputs validation	**Informative**	**Open**
GMC-17	Improvable design	**Informative**	**Open**

Vulnerability details

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

- Category
- Active
- Risk
- Description
- Recommendations

Wrong init logic

Identifier	Category	Risk	State
GMC-01	Business Logic Errors	**High**	**Open**

Significant errors have been detected during the initialization of the contracts that could lead to erroneous behavior.

Wrong DAO initialization

In the `Controller` contract, the `setDao` or `_setDao` methods perform actions such as:

```
_grantRole(MULTISIG_ROLE, dao);

_stakingLogic.setDao(dao);

_converterLogic.setDao(dao);
```

This logic is not performed during the initialization of the contract by the call to the `init` method. Therefore, the logic may not be correct since the `MULTISIG_ROLE` role is not correctly set to the `dao` address, nor calling to the `setDao` method in the `_stakingLogic` and `_converterLogic` contracts.

Source reference

- Controller.sol#L74
- Controller.sol#L135-L136

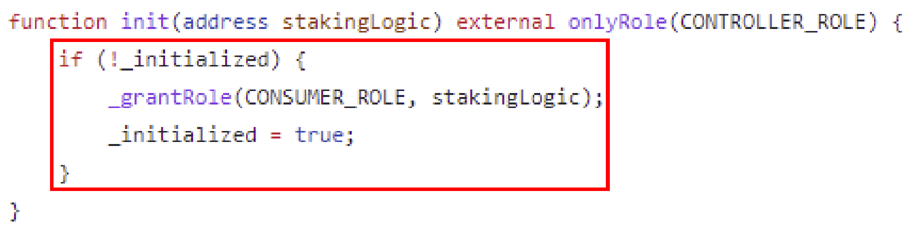
Discrepancies have also been found between the initialization and the `setDao` method in the `Converter` contract.

Source reference

- Converter.sol#L365
- Converter.sol#L378

Complicate the error detection

The `init` method of the audited contracts does not throw an error in case of being already initialized, in addition, no event is emitted during its correct initialization. This behavior facilitates a front-running and complicates error detection during deployment.



```
function init(address stakingLogic) external onlyRole(CONTROLLER_ROLE) {
    if (!_initialized) {
        _grantRole(CONSUMER_ROLE, stakingLogic);
        _initialized = true;
    }
}
```

Source reference

- EnergyStorage.sol#L46
- StakingStorage.sol#L109
- Staking.sol#L91
- Controller.sol#L60
- Converter.sol#L356

Recommendations

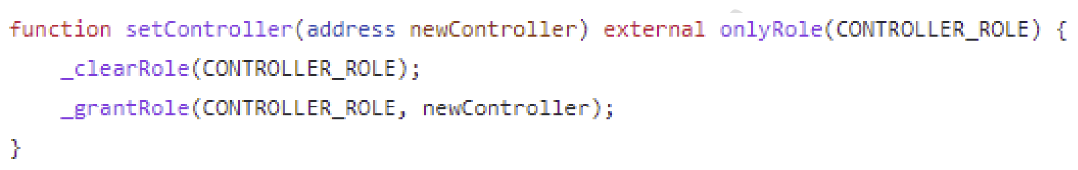
- Add an event during the call to the initialization method.
- Throw an error if it has already been initialized.

Bypass not EOA restriction

Identifier	Category	Risk	State
GMC-02	Improper Input Validation	**Medium**	**Open**

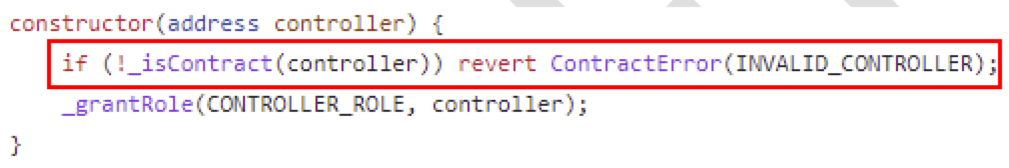
Certain methods of the `EnergyStorage` and `StakingStorage` contracts do not properly check the arguments, which can lead to avoiding requirements for new controllers.

The `setController` method does not verify that the `newController` is a contract, therefore a `CONTROLLER_ROLE` can be set to an EOA.



```
function setController(address newController) external onlyRole(CONTROLLER_ROLE) {
    _clearRole(CONTROLLER_ROLE);
    _grantRole(CONTROLLER_ROLE, newController);
}
```

This restriction does appear in the `constructor` of the `Staking`, `EnergyStorage` and `StakingStorage` contracts. Thus, it is a way to bypass the restrictions of the governance .



```
constructor(address controller) {
    if (!_isContract(controller)) revert ContractError(INVALID_CONTROLLER);
    _grantRole(CONTROLLER_ROLE, controller);
}
```

We can observe the same incidence in the `setMultisig` method of the `Controller` contract, so it is recommended to unify the settings of the contract by calling the `_setXXXX` method, which is created specifically for this, instead of doing it manually in the constructors/initializers and in a controlled way in public methods.

Recommendations

- Make sure that the restrictions are the same in the setting methods and in the constructor.

Source Code References

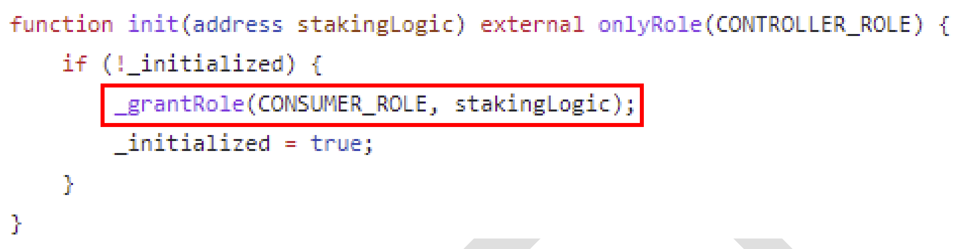
- EnergyStorage.sol#L58
- EnergyStorage.sol#L19
- StakingStorage.sol#L119
- StakingStorage.sol#L27
- Staking.sol#L123
- Controller.sol#L38
- Controller.sol#L139

Inheritance design allows constraint bypass

Identifier	Category	Risk	State
GMC-03	Design Weaknesses	**Medium**	**Open**

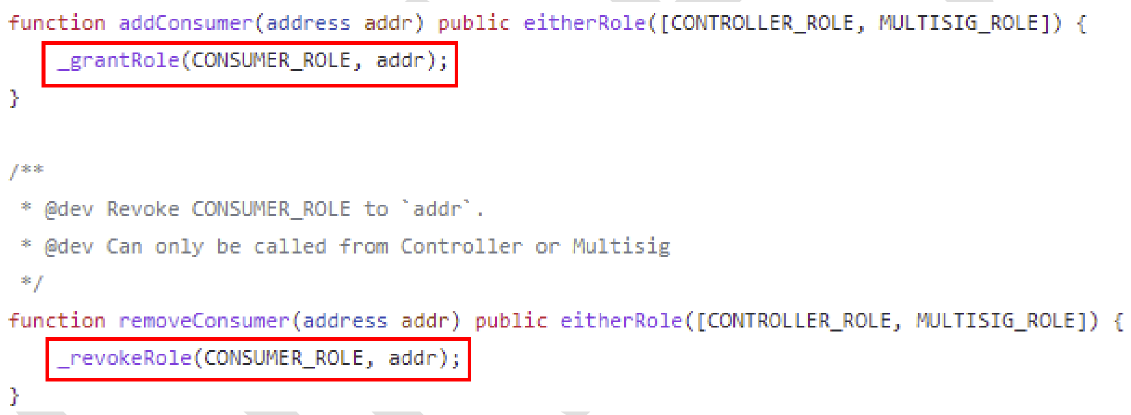
Due to methods exposed in the inherited contracts it is possible to bypass part of the initially established restrictions.

The `init` method makes a `grantRole(CONSUMER_ROLE)` of the address delievered to verify that it is a contract.



```
function init(address stakingLogic) external onlyRole(CONTROLLER_ROLE) {
    if (!_initialized) {
        _grantRole(CONSUMER_ROLE, stakingLogic);
        _initialized = true;
    }
}
```

However, since it inherits from the `PermissionControl` contract, the user with the `CONTROLLER_ROLE` role can also call the `addConsumer` and `removeConsumer` methods.



```
function addConsumer(address addr) public eitherRole([CONTROLLER_ROLE, MULTISIG_ROLE]) {
    _grantRole(CONSUMER_ROLE, addr);
}

/**
 * @dev Revoke CONSUMER_ROLE to `addr`.
 * @dev Can only be called from Controller or Multisig
 */
function removeConsumer(address addr) public eitherRole([CONTROLLER_ROLE, MULTISIG_ROLE]) {
    _revokeRole(CONSUMER_ROLE, addr);
}
```

This causes the logic of the `init` method to be duplicated, although it should be noted that the `addConsumer` method does not check that the address is a contract and not an EOA, so the restrictive logic of the `init` method can be bypassed by calling to the `addConsumer` method. In addition to being able to keep the `EnergyStorage` and `StakingStorage` contracts initialized and without `CONSUMER_ROLE`.

Recommendations

- Reduce the inheritance of the contracts or set the methods of the abstract contracts to internal.

Source Code References

- EnergyStorage.sol#L46
- StakingStorage.sol#L109

Unbounded loop in getHistory and calculateEnergy methods

Identifier	Category	Risk	State
GMC-04	DoS With Block Gas Limit	**Medium**	**Open**

The logic executed to get the staking history might trigger a denial of service (DoS) by GAS exhaustion because it iterates over the available stakes for an address without any limit.

Loops without limits are considered a bad practice in the development of Smart Contracts, since they can trigger a denial of service (DoS) or overly expensive executions, this is the case affecting `StakingStorage`.

A denial of Service to this function is possible, creating a large number of `Stakes` with the `stake` method, this will allow to force that the cost of iterating all the inputs surpasses the maximum allowed GAS per block, which currently is of 15 million approximately.

An unlimited loop issue is found in the `getHistory` method with less critical impact since they are view functions, although they will have the same problem if they have to be called from another contract.

If a user makes numerous calls to the `stake` method naturally, the `getHistory` and `calculateEnergy` methods could be denied.

Recommendations

- Limit the maximum number of `Stakes` that the user can have.

References

- https://ethgasstation.info/blog/ethereum-block-size

Source Code References

- Staking.sol#L181
- Converter.sol#L102-L103
- StakingStorage.sol#L57-L74

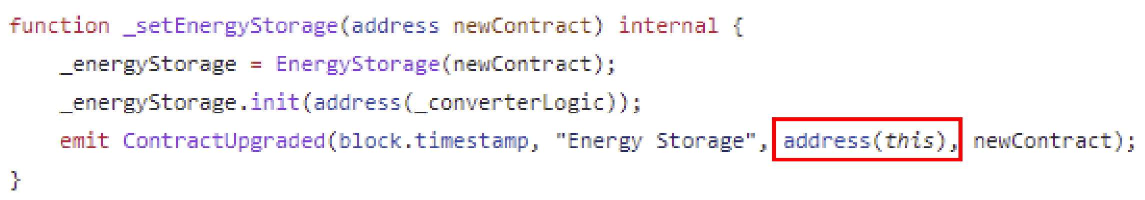
Wrong emitted event

Identifier	Category	Risk	State
GMC-05	Business Logic Errors	**Medium**	**Open**

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 important state changes are emitted in order to allow potential actors to monitor the blockchain; such as DApps, automated processes and users, can be notified of these significant state changes.

The problem resides in the `Controller` contract, which emits the `ContractUpgraded` event, this method has an argument called `oldAddress` designed to return the previous and the new values, however `address(this)` is always returned instead of the previous value. Thus, the information issued is incorrect.



```
function _setEnergyStorage(address newContract) internal {
    _energyStorage = EnergyStorage(newContract);
    _energyStorage.init(address(_converterLogic));
    emit ContractUpgraded(block.timestamp, "Energy Storage", address(this), newContract);
}
```

Recommendations

- Send the previous value when emitting the `ContractUpgraded` event.

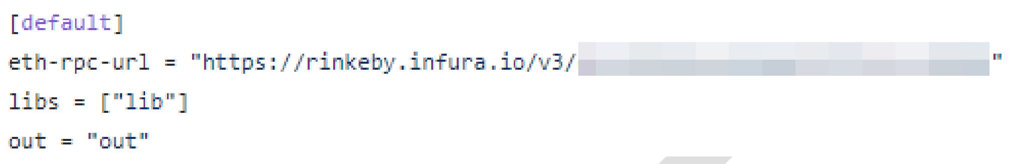
Source Code References

- Controller.sol#L35
- Controller.sol#L154
- Controller.sol#L179
- Controller.sol#L184
- Controller.sol#L190
- Controller.sol#L195
- Controller.sol#L201
- Controller.sol#L221
- Controller.sol#L227
- Controller.sol#L233

Project information leak

Identifier	Category	Risk	State
GMC-06	Unsafe Storage	**Low**	**Open**

During the review, it was detected that `foundry.toml` file has uploaded private information to the GitHub repository, such as Infura API key.



```
[default]
eth-rpc-url = "https://rinkeby.infura.io/v3/[redacted]"
libs = ["lib"]
out = "out"
```

This type of information should never be hardcoded and should be stored in a specific configuration file (.env) and should never be uploaded to the GitHub repository (even if it is development data).

Recommendations

- Set the `apiKey` as an enviroment variable.

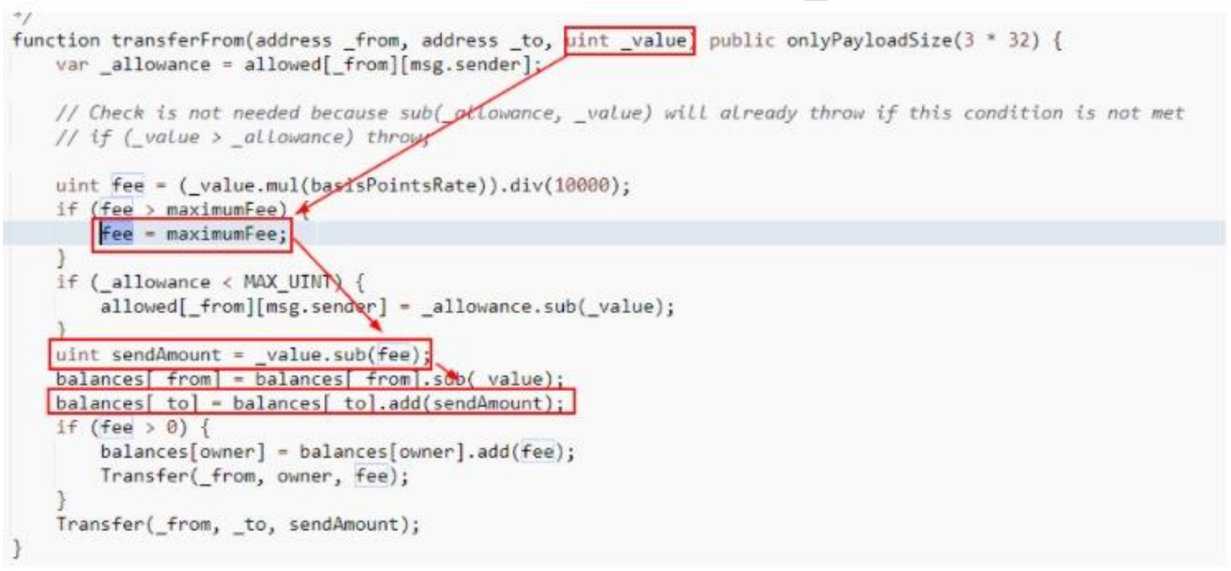
Source Code References

- foundry.toml#L2

Not compatible with fee-based tokens

Identifier	Category	Risk	State
GMC-07	Business Logic Errors	**Low**	**Open**

The current staking logic does not contemplate ERC20 tokens with fee during the *transferFrom*, therefore, the amount received by `stake` will be less than the expected to carry out the stake. Certain tokens may implement a fee during transfers, this is the case of `USDT`, even though the project has currently set it to 0. So, the `transferFrom` function would return `true` despite receiving less than expected.



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

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

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

By generating the stake without having said fee, the project will end up assuming the cost of the fee, which may cause economic losses to the project.

```
function stake(uint256 tokenId, uint256 amount) external whenNotPaused {
    if (tokenId > 1) revert InvalidInput(WRONG_TOKEN);
    if (amount == 0) revert InvalidInput(WRONG_AMOUNT);
    address user = msg.sender;
    uint256 tokenBalance = _token[tokenId].balanceOf(user);
    if (amount > tokenBalance) revert InvalidInput(INSUFFICIENT_BALANCE);

    _token[tokenId].safeTransferFrom(user, address(this), amount);

    uint256 lastStakeId = _storage[tokenId].getUserLastStakeId(user);
    uint256 stakeBalance = (_storage[tokenId].getStake(user, lastStakeId)).amount;
    uint256 newAmount = stakeBalance + amount;
    _storage[tokenId].updateHistory(user, newAmount);
    totalStakedAmount[tokenId] += amount;

    emit Staked(_tokenName[tokenId], user, block.timestamp, amount);
}
```

Recommendations

- Check the balance received by the difference between the balances before and after the call to the`transferFrom` method.

References

- https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code

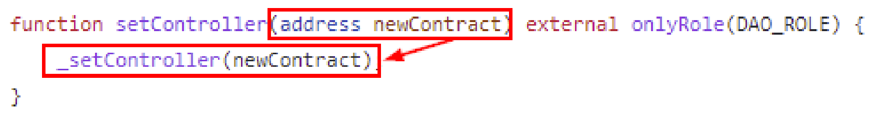
Source Code References

- Staking.sol#L176

Wrong logic around getController

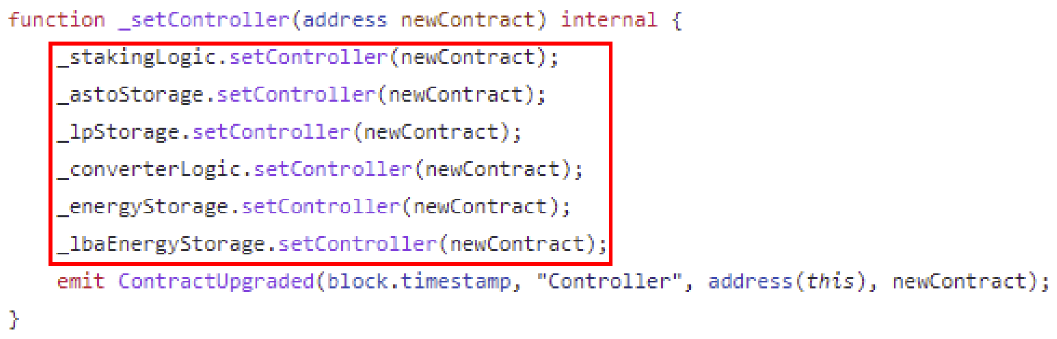
Identifier	Category	Risk	State
GMC-08	Business Logic Errors	**Low**	**Open**

The `getController` method of the `Controller` contract always returns `address(this)`, but it does not necessarily reflect the reality as it may have been modified by the `setController` method.



```
function setController(address newContract) external onlyRole(DAO_ROLE) {
    _setController(newContract);
}
```

There could be a scenario where the user with `DAO_ROLE` privileges calls `setController` with a different address than the one of the contract itself, which would imply updating the controller of all the underlying contracts.



```
function _setController(address newContract) internal {
    _stakingLogic.setController(newContract);
    _astoStorage.setController(newContract);
    _lpStorage.setController(newContract);
    _converterLogic.setController(newContract);
    _energyStorage.setController(newContract);
    _lbaEnergyStorage.setController(newContract);
    emit ContractUpgraded(block.timestamp, "Controller", address(this), newContract);
}
```

This would cause the contracts and controller to be out of sync and the current `Controller` to be denied.

Recommendations

- Remove the `setController` public method from the `Controller` contract.

Source Code References

- Controller.sol#L330

Discrepancy with documentation

Identifier	Category	Risk	State
GMC-09	Business Logic Errors	**Low**	**Open**

The documentation of the code helps to know the expected logic or the logic that the method has to perform, so a discrepancy between the logic of the method and the comment can lead to an error in its programming.

Comment discrepancy

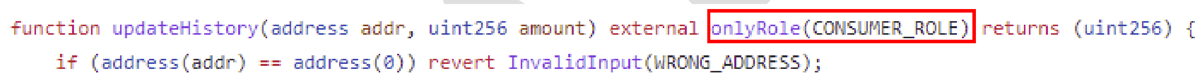
In the `updateHistory` method of the `StakingStorage` contract the following is mentioned:

```
@notice Function can be called only manager
```

But in reality, and in the documentation, they can only be called by `CONSUMER_ROLE`.

Source Reference

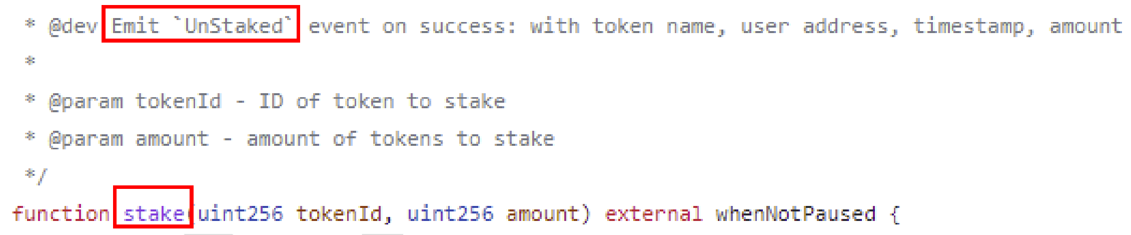
- https://github.com/altered-state-machine/genome-mining-contracts-audit/blob/main/audit/roles.md#consumer_role



```
function updateHistory(address addr, uint256 amount) external onlyRole(CONSUMER_ROLE) returns (uint256) {
    if (address(addr) == address(0)) revert InvalidInput(WRONG_ADDRESS);
```

Similarly, in the `stake` method, the following comment is observed:

`@dev Emit `UnStaked` event on success: with token name, user address, timestamp, amount`
However, the emitted event and the correct one, is `Staked` instead of `UnStaked`.



```
 * @dev Emit `UnStaked` event on success: with token name, user address, timestamp, amount
 *
 * @param tokenId - ID of token to stake
 * @param amount - amount of tokens to stake
 */
function stake(uint256 tokenId, uint256 amount) external whenNotPaused {
```

Source reference

- StakingStorage.sol#L37
- main/audit/roles.md#consumer_role
- Staking.sol#L164

Readme discrepancy

According to the Roles documentation (https://github.com/altered-state-machine/genome-mining-contracts-audit/blob/main/audit/roles.md), the `DAO_ROLE` can *"Set the DAO address for all contracts"*, however, we can observe in the `Staking` and `Converter` contracts through the `setDao` method, that the authorized role for it is the `CONTROLLER_ROLE`, causing inconsistency between the code and the documentation.

Reference

- main/audit/roles.md

Source reference

- Converter.sol#L377
- Staking.sol#L114

Recommendations

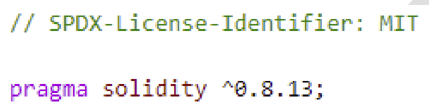
- Check that the code is related to the documentation provided.

Outdated compiler

Identifier	Category	Risk	State
GMC-10	Outdated Software	**Informative**	**Open**

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

We have detected that the audited contract uses the following version of Solidity pragma `0.8.13`:



```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.13;
```

Recommendations

- Solidity branch 0.8 has important bug fixes related to the treatment of arrays, so it is recommended to use the most up to date version of the pragma.

References

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

GAS optimization

Identifier	Category	Risk	State
GMC-11	Bad Coding Practices	**Informative**	**Open**

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

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

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

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

Avoid the use of SafeMath

The use of the OpenZeppelin SafeMath library has been detected, this library validates if an arithmetic operation results in an integer overflow/underflow. If this is the case, an exception would be thrown, reversing the transaction.

Since Solidity 0.8, the overflow/underflow check is implemented on the language level, so, we consider that the implementation of the SafeMath library for Solidity 0.8+ is not necessary.

Source reference

- Converter.sol#L124-L127
- Converter.sol#L151

References

- https://docs.soliditylang.org/en/v0.8.13/080-breaking-changes.html

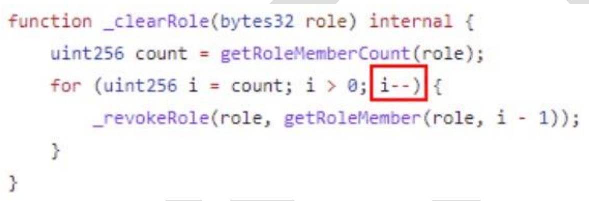
Increase operation optimization

During the audit, it was found that it is possible to optimize all the for loops in the project. There are two main ways to increment/decrement variables in solidity:

```
--i/++i will decrement/increment the value of i, and then return the decremented/
incremented value.

i--/i++ will decrement/increment the value of i, but return the original value
that i held before being decremented/incremented.
```

Since the return of the decrement/increment operation of the for loop is indifferent and discarded, both `i--/i++` or `--i/++i` instructions are completely valid instructions, however, the `--i/++i` instruction has a considerably lower cost compared to decrementing/incrementing a variable using `i--/i++`, for this reason it is convenient to use `--i/++i` for the decrements/increments of the for loops.

An example of this behaviour can be seen below:



```
function _clearRole(bytes32 role) internal {
    uint256 count = getRoleMemberCount(role);
    for (uint256 i = count; i > 0; i--) {
        _revokeRole(role, getRoleMember(role, i - 1));
    }
}
```

Source reference

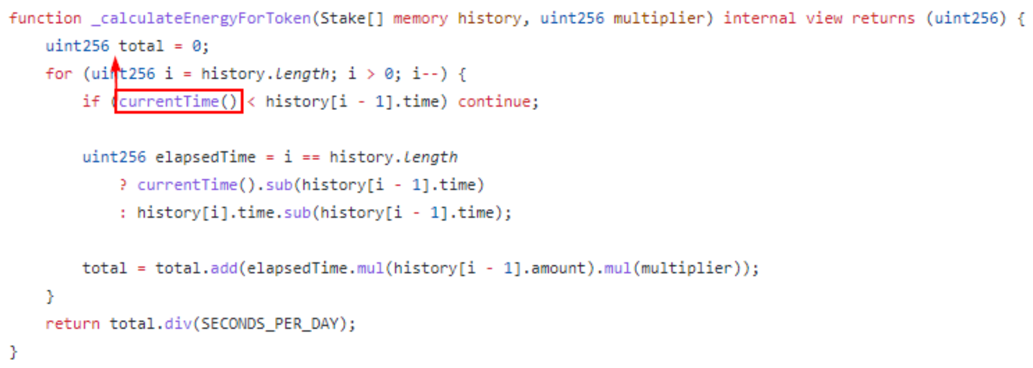
- PermissionControl.sol#L39
- StakingStorage.sol#L63
- StakingStorage.sol#L69
- Converter.sol#L120
- Converter.sol#L242
- Converter.sol#L308

References

- https://docs.soliditylang.org/en/latest/types.html#compound-and-increment-decrement-operators

Executions Cost

In various methods of the `Converter` contract, there are calls made to the `currentTime()` method within a loop. Since the logic of this method only returns `block.timestamp`, it is recommended to store its value in a variable outside the main loop, avoiding unnecessary and successive calls, thus the execution cost is much lower.

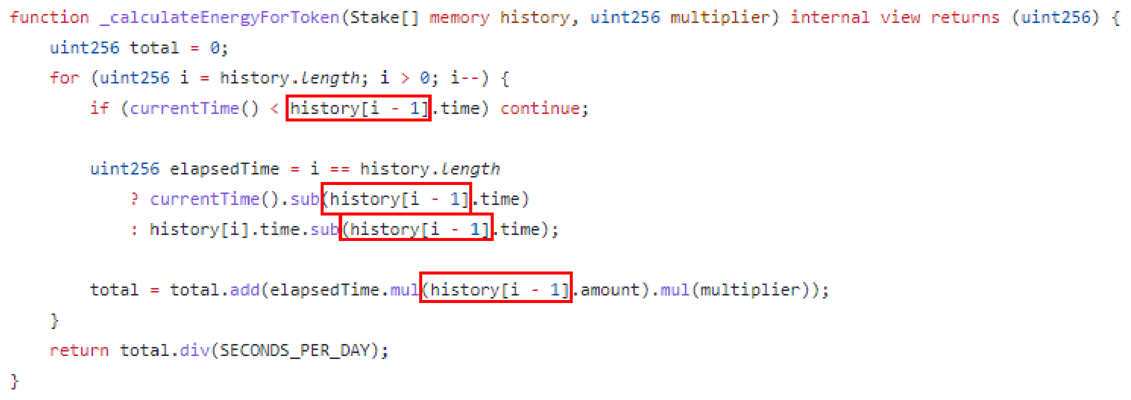


```
function _calculateEnergyForToken(Stake[] memory history, uint256 multiplier) internal view returns (uint256) {
    uint256 total = 0;
    for (uint256 i = history.length; i > 0; i--) {
        if (currentTime() < history[i - 1].time) continue;

        uint256 elapsedTime = i == history.length
            ? currentTime().sub(history[i - 1].time)
            : history[i].time.sub(history[i - 1].time);

        total = total.add(elapsedTime.mul(history[i - 1].amount).mul(multiplier));
    }
    return total.div(SECONDS_PER_DAY);
}
```

Source reference

- Converter.sol#L121
- Converter.sol#L124
- Converter.sol#L244

Likewise, it is recommended to cache the elements used during the executed loops, avoiding unnecessary calculations and calls, which optimizes the gas execution of the function.

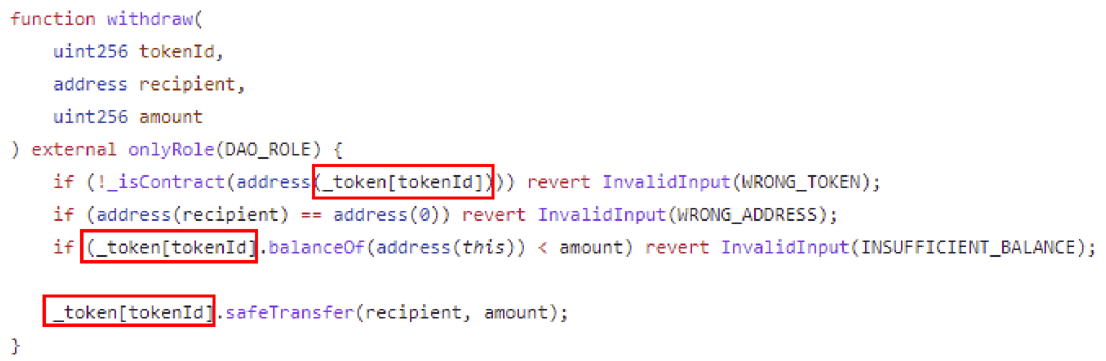


```
function _calculateEnergyForToken(Stake[] memory history, uint256 multiplier) internal view returns (uint256) {
    uint256 total = 0;
    for (uint256 i = history.length; i > 0; i--) {
        if (currentTime() < history[i - 1].time) continue;

        uint256 elapsedTime = i == history.length
            ? currentTime().sub(history[i - 1].time)
            : history[i].time.sub(history[i - 1].time);

        total = total.add(elapsedTime.mul(history[i - 1].amount).mul(multiplier));
    }
    return total.div(SECONDS_PER_DAY);
}
```

Source reference

- Converter.sol#L121

Another example of this casuistry could be the one found in the `withdraw` method of the `Staking` contract, we advise to cache the value of `_token[tokenId]` in a `token` variable, generating gas savings during the execution of the method.



```
function withdraw(
    uint256 tokenId,
    address recipient,
    uint256 amount
) external onlyRole(DAO_ROLE) {
    if (!_isContract(address(_token[tokenId]))) revert InvalidInput(WRONG_TOKEN);
    if (address(recipient) == address(0)) revert InvalidInput(WRONG_ADDRESS);
    if (_token[tokenId].balanceOf(address(this)) < amount) revert InvalidInput(INSUFFICIENT_BALANCE);

    _token[tokenId].safeTransfer(recipient, amount);
}
```

Source reference

- Staking.sol#L61-L65

Dead Code

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

`INSUFFICIENT_STAKED_AMOUNT` constant of the `Util` contract is not used during the execution of the contract, so it would be convenient to either remove it or to use it.

Source reference

- Util.sol#L29

Unnecessary Method

During the audit, the existence of the `currentTime` method of the `StakingStorage` contract was detected. This method is unnecessary since it only returns the result of `block.timestamp` and can implement its logic where necessary. Additionally, the execution of this type of code consumes unnecessary GAS during deployment.

Source reference

- StakingStorage.sol#L90

Avoid use of unnecessary map

The `_token` map of the `Staking` contract is used to store the `tokenID` and its value can only be 0 or 1, so using a map can be redundant, confusing and therefore unnecessary. It is considered that a better way to store the two tokens would be to do so in variables of independant state, so one or the other are used based on a boolean argument instead of a `tokenId`.

Source reference

- Staking.sol#L35

Optimize revert messages

Ethereum Virtual Machine operates under a 32-byte word memory model where an additional gas cost is paid by any operation that expands the memory that is in use.

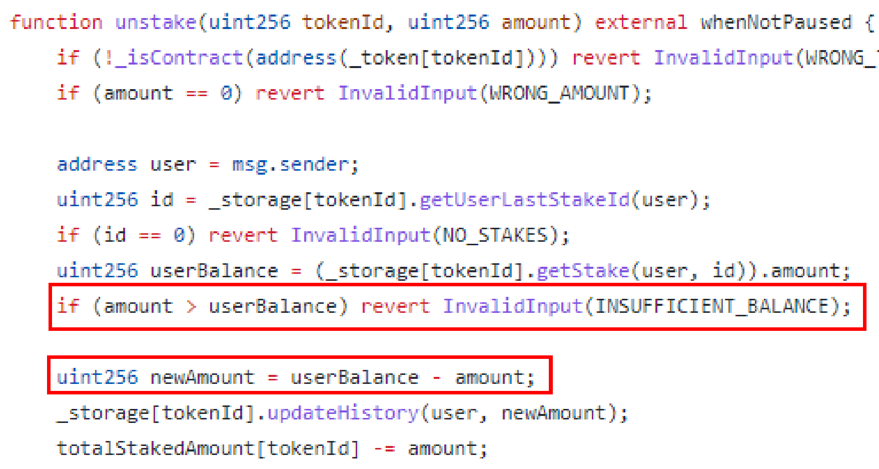
Therefore, exceeding error messages of this length means increasing the number of slots necessary to process the `revert`, reducing the error messages to 32 bytes or less would lead to saving gas.

Source reference

- Util.sol#L12
- Util.sol#L20
- Util.sol#L29

Use of unchecked keyword

From Solidity 0.8.0 version, all the arithmetic operations revert to overflow/underflow by default, as can be seen in the following image, the `newAmount` variable of the `unstake` method is calculated using the default "**checked**" behavior, so it costs more gas when performing arithmetic operations. Since it has already been verified above that `amount` cannot be greater than `userBalance`, an "**unchecked**" block could be used in order to save gas.



```
function unstake(uint256 tokenId, uint256 amount) external whenNotPaused {
    if (!_isContract(address(_token[tokenId]))) revert InvalidInput(WRONG_TOKEN);
    if (amount == 0) revert InvalidInput(WRONG_AMOUNT);

    address user = msg.sender;
    uint256 id = _storage[tokenId].getUserLastStakeId(user);
    if (id == 0) revert InvalidInput(NO_STAKES);
    uint256 userBalance = (_storage[tokenId].getStake(user, id)).amount;
    if (amount > userBalance) revert InvalidInput(INSUFFICIENT_BALANCE);

    uint256 newAmount = userBalance - amount;
    _storage[tokenId].updateHistory(user, newAmount);
    totalStakedAmount[tokenId] -= amount;
```

Source reference

- Staking.sol#L210

Logic Optimization

Instead of checking if `_isContract` token is a contract, it is more optimal in terms of gas to check if the address is equal to `address(0)` because the mapping is only established in the `init` method, which is where these checks have to be performed.

An example of this optimization would be as follows:

```
    if (address(_token[tokenId]) == address(0)) revert InvalidInput(WRONG_TOKEN);
```

Source reference

- Staking.sol#L201
- Controller.sol#L159
- Controller.sol#L206

The `eitherRole` modifier of the `PermissionControl` contract can be optimized since the current logic requires checking both roles, however, it is more optimal to check them separately, as is done in the following code snippet.

```
modifier eitherRole(bytes32[2] memory roles) {
        address sender = _msgSender();
        if (hasRole(roles[0], sender)) { _; return; }
        if (hasRole(roles[1], sender)) { _; return; }
        revert AccessDenied(MISSING_ROLE);
    }
```

Source reference

- PermissionControl.sol#L26-L31

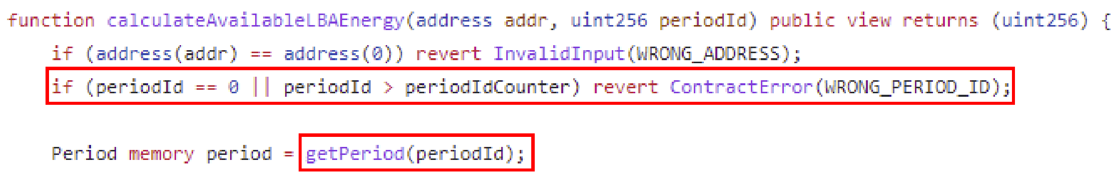
Avoid duplicate logic

In the `init` method of the `Controller` contract, the state variables are initialized, then the `_upgradeContracts` method is called, which will initialize them again.
Aditionally, since they are pre-initialized, some methods such as `_setStakingLogic` which check the previous value to remove certain roles will further increase the initialization gas cost.
One possible optimization can be achieved by removing the initialization in the `init` method of that contract and calling to the `_setDao` method instead.

Source reference

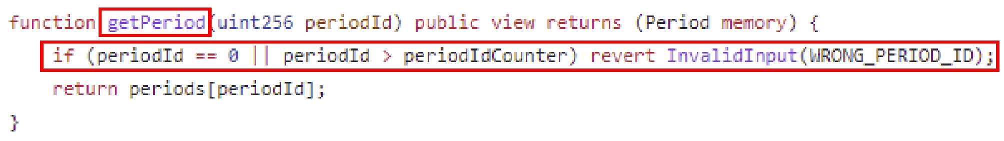
- Controller.sol#L75-L82

The `calculateEnergy`, `calculateAvailableLBAEnergy` and `useEnergy` methods perform a check in order to validate if the `periodId` received by argument provided by the user, has the expected value. This validation is as follows:



```
function calculateAvailableLBAEnergy(address addr, uint256 periodId) public view returns (uint256) {
    if (address(addr) == address(0)) revert InvalidInput(WRONG_ADDRESS);
    if (periodId == 0 || periodId > periodIdCounter) revert ContractError(WRONG_PERIOD_ID);

    Period memory period = getPeriod(periodId);
```

However, as can be seen, after performing this check, a call is made to the `getPeriod` method, where the same validation is performed, thus being redundant and unnecessary.



```
function getPeriod(uint256 periodId) public view returns (Period memory) {
    if (periodId == 0 || periodId > periodIdCounter) revert InvalidInput(WRONG_PERIOD_ID);
    return periods[periodId];
}
```

Source reference

- Converter.sol#L98
- Converter.sol#L141
- Converter.sol#L193

Code style

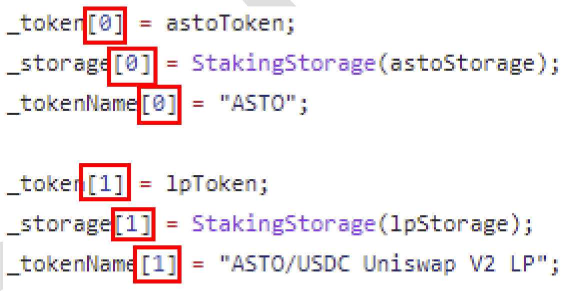
Identifier	Category	Risk	State
GMC-12	Bad Coding Practices	**Informative**	**Open**

It has been possible to verify that, despite the good quality of the code, there is a lack of order and structure that makes reading and analyzing the code difficult.

This is a very common bad practice, especially in these types of projects that are continually changing and improving. This is not a vulnerability in itself, but it helps to improve the code and to reduce the appearance of new vulnerabilities.

Use of constants instead of values

The `ASTO_TOKEN_ID` and `LP_TOKEN_ID` constants define the positions where these tokens are stored in the mapping of the contract, despite using them, sometimes we can observe that the value has been entered manually.



```
_token[0] = astoToken;
_storage[0] = StakingStorage(astoStorage);
_tokenName[0] = "ASTO";

_token[1] = lpToken;
_storage[1] = StakingStorage(lpStorage);
_tokenName[1] = "ASTO/USDC Uniswap V2 LP";
```

Recommendations

As a reference, it is always recommendable to apply some coding style/good practices that can be found in multiple standards such as:

- "Solidity Style Guide" (https://docs.soliditylang.org/en/v0.8.0/style-guide.html).

These references are very useful to improve the quality of the smart contract. A few of these practices are generally known and accepted forms to develop software.

References

- https://docs.soliditylang.org/en/v0.8.0/style-guide.html

Source Code References

- Staking.sol#L92-L98

Decentralization recommendation

Identifier	Category	Risk	State
GMC-13	Bad Coding Practices	**Informative**	**Open**

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

The team maintains some centralized parts that imply trust in the project, which are indeed necessary. A few of the administrative functionalities are under the control of the project, above you can see an example of what could be fixed in order to improve the decentralization and trust of the project.

Governance Denial of Service

Denial of Service by CONSUMER_ROLE

The `getHistory` method can be denied by the user with the role `CONSUMER_ROLE`. If the user's history is updated by calling the `updateHistory` method multiple times, numerous ids would be created in `stakeIds` and when getting the user's `totalStakes` go over it, it would fail due to out of gas.

Source reference

- StakingStorage.sol#L58-L60
- Staking.sol#L262-L268

Denial of Service by MANAGER_ROLE

If enough periods are added, there can be a denial of service to the user for the use of the `getCurrentPeriodId` method. It should be noted that there is no possibility of deleting a period, so any human error could lead to having to migrate `Converter`.

Source reference

- Converter.sol#L276
- Converter.sol#L241

Periods issues

Nothing ensures that the periods are added consecutively, which would cause the logic of the `getCurrentPeriodId` method to fail. There is also no control over the multiplier set in the periods, the use of extremely high multipliers can cause an overflow error during the energy calculation in the `_calculateEnergyForToken` method.

It is recommended to remove the `startTime` value from the `Period` structure, so that a period starts when the previous period ends. Additionally check that when adding a period, the `endTime` is greater than the previous one in both the `updatePeriod` and in the `addPeriod`, as well as checking that the multipliers are in the expected ranges.

Source reference

- IConverter.sol#L10

- Converter.sol#L241
- Converter.sol#L127
- IConverter.sol#L13-L14

Multiple unique role

It is possible to grant to different accounts the role of `DAO_ROLE`. If the `CONTROLLER_ROLE` calls `setDao` first and then calls `init`, the contract will have multiple accounts with the `DAO_ROLE` role, because `_clearRole(DAO_ROLE)` or `setDao` methods are not called during initialization of the contract.

Likewise, it can happen with `MULTISIG_ROLE`, because during the initialization of the contract, the cleaning of both roles does not take place.

Source reference

- Converter.sol#L365
- Converter.sol#L365-L367

Lack of whenNotPaused

The `StakingStorage` contract can be paused, but certain administrative methods that affect account balances are allowed to be called during the paused state.

The `updateHistory` method does not check that the contract is not paused and allows the `CONSUMER_ROLE` to make changes during a pause.

Source reference

- StakingStorage.sol#L43

The `DAO_ROLE` can withdraw the funds with the contract paused.

Source reference

- Staking.sol#L56

Uncontrolled values

Nothing prevents the `CONSUMER_ROLE` from setting arbitrary values in `updateHistory`, the `amount` does not reflect any balance associated with the account so it can be faked.

Source reference

- StakingStorage.sol#L43

The values of the total in staking are also defined arbitrarily and do not represent the real result of the balance of an account.

Source reference

- Staking.sol#L88-L89

Ensure TimeLock use

It is not checked that the `DAO_ROLE` role is a contract and not an EOA, and since it has control of the funds, it can pose a risk to decentralization if a multisignature governance system or `TimeLock` is not used.

Source reference

- Staking.sol#L100-L101
- Staking.sol#L65

Ensure Funds

Nothing ensures that you have enough funds to `unstake` because the dao can use the `withdraw` method at any time to remove the tokens from the contract.

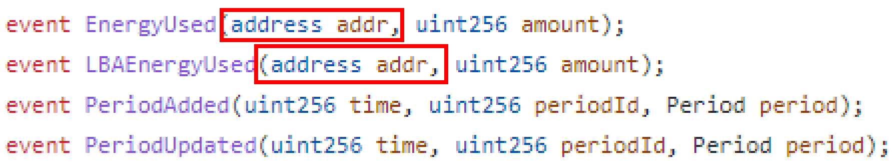
Source reference

- Staking.sol#L56
- Staking.sol#L200

Lack of event index

Identifier	Category	Risk	State
GMC-14	Bad Coding Practices	**Informative**	**Open**

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



```
event EnergyUsed(address addr, uint256 amount);
event LBAEnergyUsed(address addr, uint256 amount);
event PeriodAdded(uint256 time, uint256 periodId, Period period);
event PeriodUpdated(uint256 time, uint256 periodId, Period period);
```

Recommendations

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

Source Code References

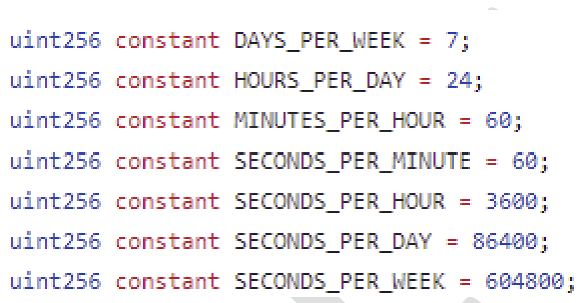
- Converter.sol#L43-L44

Solidity literals

Identifier	Category	Risk	State
GMC-15	Code Optimization	**Informative**	**Open**

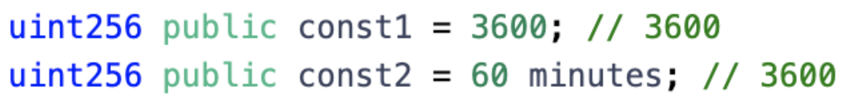
In order to make the code easier to read and to minimize human errors, Solidity recommends the use of literals which consequently makes it more user friendly.

It is possible to improve the reading of dates by using time units: Suffixes like `seconds`, `minutes`, `hours`, `days` and `weeks`, which are less prone to errors.



```
uint256 constant DAYS_PER_WEEK = 7;
uint256 constant HOURS_PER_DAY = 24;
uint256 constant MINUTES_PER_HOUR = 60;
uint256 constant SECONDS_PER_MINUTE = 60;
uint256 constant SECONDS_PER_HOUR = 3600;
uint256 constant SECONDS_PER_DAY = 86400;
uint256 constant SECONDS_PER_WEEK = 604800;
```

An example of this would be the optimization of some of the previous constants, such as `SECONDS_PER_HOUR`, being able to replace its value of 3600 with 60 minutes.



```
uint256 public const1 = 3600; // 3600
uint256 public const2 = 60 minutes; // 3600
```

References

- https://docs.soliditylang.org/en/latest/units-and-global-variables.html#units-and-globally-available-variables

Source Code References

- TimeConstants.sol#L9-L15

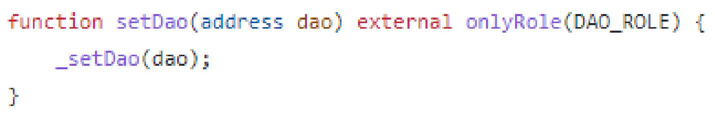
Lack of inputs validation

Identifier	Category	Risk	State
GMC-16	Improper Input Validation	**Informative**	**Open**

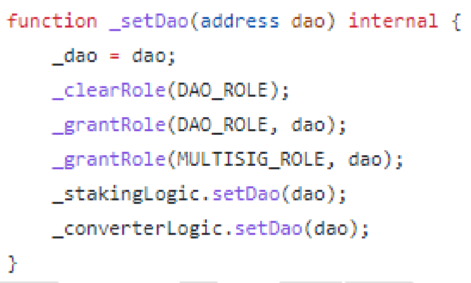
Certain methods of the different contracts in the `Genome Mining` project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

In the `Staking`, the `init` method does not verify that the address sent through the arguments are valid, so it allows the address to be established to `address(0)`.

The same thing happens in the `Controller` contract, where it is allowed to establish the DAO as `address(0)`, being able to deny the service of all the contracts.



```
function setDao(address dao) external onlyRole(DAO_ROLE) {
    _setDao(dao);
}
```

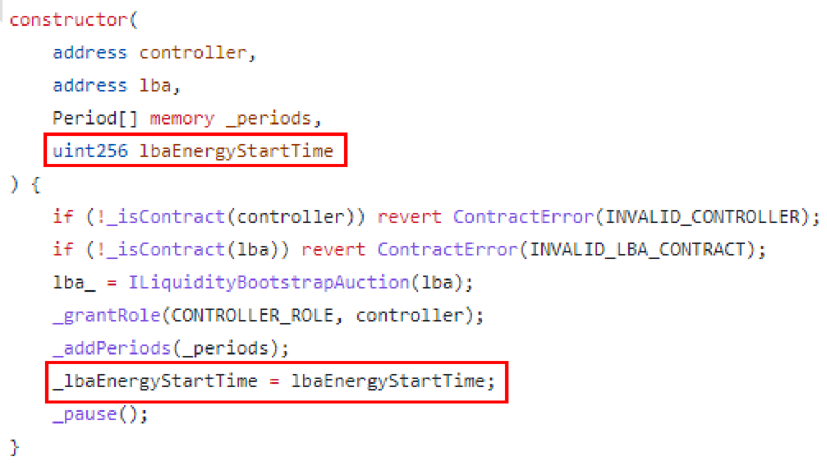


```
function _setDao(address dao) internal {
    _dao = dao;
    _clearRole(DAO_ROLE);
    _grantRole(DAO_ROLE, dao);
    _grantRole(MULTISIG_ROLE, dao);
    _stakingLogic.setDao(dao);
    _converterLogic.setDao(dao);
}
```

Source references

- Staking.sol#L84-L87
- Controller.sol#L267-L301

In the `constructor` of the `Converter` contract it is recommended to check that the value of `lbaEnergyStartTime` received by arguments is in the expected range.



```
constructor(
    address controller,
    address lba,
    Period[] memory _periods,
    uint256 lbaEnergyStartTime
) {
    if (!_isContract(controller)) revert ContractError(INVALID_CONTROLLER);
    if (!_isContract(lba)) revert ContractError(INVALID_LBA_CONTRACT);
    lba_ = ILiquidityBootstrapAuction(lba);
    _grantRole(CONTROLLER_ROLE, controller);
    _addPeriods(_periods);
    _lbaEnergyStartTime = lbaEnergyStartTime;
    _pause();
}
```

Source references

- Converter.sol#L52

Improvable design

Identifier	Category	Risk	State
GMC-17	Design Weaknesses	**Informative**	**Open**

Following there are a few examples that can help improve the code design. Improving code design can help make the code more consistent, readable and auditable.

Unnecessary hierarchy

The use of inheritance in contracts often makes the code easier to understand, reduces it and makes it clearer, however it is not convenient to abuse it because it can also make it less readable and auditable, finding the exact balance is important when designing the contracts.

There are languages such as vyper that eliminate inheritance, considering it less auditable.

> *Class inheritance requires people to jump between multiple files to understand what a program is doing, and requires people to understand the rules of precedence in case of conflicts ("Which class's function* `X` *is the one that's actually used?"). Hence, it makes code too complicated to understand which negatively impacts auditability.*

In this case, the `Util` contract, which is not `abstract`, contains only one function that is already imported into the Open Zeppelin libraries, so it is convenient to use the library instead of increasing the complexity of the code in order to inherit only that function.

It is convenient to use existing libraries as long as they help to reduce the complexity of the code. Keep in mind that incidents related to inheritance have already been reported, such as the Inheritance design allows constraint bypass.

Reference

- Address.sol#L41

Source reference

- Util.sol#L46-L47

Inconsistence logic

When the `Controller` contract is initiated, if the argument to an address is not a contract, the contract is reversed with an error. However, when the values are modified using the `upgradeContracts` method, these values are ignored but no error is produced, so the user might think that they have been modified when in fact they have not.

```
function _upgradeContracts(
    address astoToken,
    address astoStorage,
    address lpToken,
    address lpStorage,
    address stakingLogic,
    address converterLogic,
    address energyStorage,
    address lbaEnergyStorage
) internal {
    if (_isContract(astoToken)) _setAstoToken(astoToken);
    if (_isContract(astoStorage)) _setAstoStorage(astoStorage);
    if (_isContract(lpToken)) _setLpToken(lpToken);
    if (_isContract(lpStorage)) _setLpStorage(lpStorage);
    if (_isContract(stakingLogic)) _setStakingLogic(stakingLogic);
    if (_isContract(energyStorage)) _setEnergyStorage(energyStorage);
    if (_isContract(lbaEnergyStorage)) _setLBAEnergyStorage(lbaEnergyStorage);
    if (_isContract(converterLogic)) _setConverterLogic(converterLogic);
    _setController(address(this));
}
```

All of the variable setting methods must have the same behavior and the same checks, so that they are not confusing or bypassed.

Source references

- Controller.sol#L61-L69
- Controller.sol#L245

Annexes

Annex A – Vulnerabilities Severity

Red4Sec determines the vulnerabilities severity 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):

Vulnerability Severity

The risk classification has been made on the following 5 value 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