



SECURITY AUDIT

Client: **RXT token**



TABLE OF CONTENTS

| | |
|--|----|
| Table of Contents | 2 |
| Executive overview | 5 |
| 1. Audit details | 5 |
| 2. Scope of the code and target data: | 5 |
| Audit Procedure | 6 |
| Terminology | 8 |
| Vulnerabilities by status: | 9 |
| Findings | 10 |
| summary | 10 |
| Security Findings | 11 |
| High severity | 11 |
| 1. Insufficient Gas Griefing Attack | 11 |
| Description: | 11 |
| Impact: | 11 |
| Mitigation: | 12 |
| 2. Reentrancy on meta TXs execution can lead to loss of funds | 13 |
| Description: | 13 |
| Impact: | 13 |
| Proof of Concept: | 13 |
| Mitigation: | 13 |
| 3. Risk of Incorrect Interface Reporting Due to Insufficient Gas | 14 |

| | |
|--|----|
| Description: | 14 |
| Impact: | 15 |
| Proof of Concept: | 15 |
| Mitigation: | 16 |
| 4. Gas limit DOS attack via unbounded operations | 17 |
| Description: | 17 |
| Impact: | 17 |
| Mitigation: | 18 |
| Medium severity | 19 |
| 1. No account existence check on low-level call | 19 |
| Description: | 19 |
| Impact: | 19 |
| Mitigation: | 19 |
| 2. The owner is a single point of failure and a centralization risk | 19 |
| Description: | 19 |
| Impact: | 20 |
| Mitigation: | 20 |
| Low severity | 21 |
| 1. Doesn't check if _ids and _quantities length is the same in batchBurn | 21 |
| Description: | 21 |
| Impact: | 21 |
| Mitigation: | 22 |
| 2. Owner can renounce ownership | 22 |
| Description: | 22 |

| | |
|---|-----------|
| Impact: | 22 |
| Mitigation: | 23 |
| 3. Lack of double step Transfer ownership pattern | 23 |
| Description: | 23 |
| Impact: | 23 |
| Mitigation: | 23 |
| Informational issues | 24 |
| 1. Consider using named mappings | 24 |
| Description: | 24 |
| 2. Save big amount of gas by using uint256 instead of bool in mapping | 25 |
| Description: | 25 |
| Impact: | 25 |
| 3. Cache array length outside of loop saves gas | 26 |
| Description: | 26 |
| Mitigation: | 26 |
| Soken Contact Info | 27 |

EXECUTIVE OVERVIEW

1. Audit details

| | |
|-------------------|--------------|
| Name | RXT Token |
| Type of Contracts | NFT contract |
| Platform | OpenSea |
| Language | Solidity |

2. Scope of the code and target data:

- Provided Smart Contract address: <https://polygonscan.com/address/0x2953399124f0cbb46d2cbacd8a89cf0599974963>
- Commit ID: -
- Smart Contracts in scope: <https://polygonscan.com/address/0x2953399124f0cbb46d2cbacd8a89cf0599974963>

AUDIT PROCEDURE

1. **Research into Architecture and Docs:** Soken conducts an initial overview of the smart contracts, its architecture, logic and documentation. This provides the audit team with a better understanding of the contract's intended behavior and its context within a broader ecosystem.
2. **Risk Assessment:** Using information gathered during the initial consultation and research phase, Soken's team performs a risk assessment to identify potential areas of concern and estimate the overall risk level of the project. This helps prioritize areas of focus during the audit.
3. **Manual Code Review and Walkthrough:** Soken team conducts a comprehensive manual review of the smart contract code. We analyze code quality, adherence to best practices, potential security vulnerabilities, and logic flaws.
4. **Static Analysis:** Soken security team uses tools like Slither and MythX for static analysis. We review the contract's code structure, coding standards, and general best practices. This includes analyzing function visibility, checking for common coding mistakes, and ensuring the contract adheres to the Solidity style guide.
5. **Graphing out Functionality and Contract Logic:** Using Solgraph, Soken's team visualizes the control flow of the contract to better understand its logic, connectivity, and functions. This helps identify any potential vulnerabilities in how the contract's functions interact.
6. **Manual Assessment and Custom Scripts:** The team manually assesses the use and safety of critical Solidity variables and functions in the scope to identify any arithmetic-related vulnerability classes. Custom scripts are also written and executed to perform unique tests tailored to the contract.
7. **Dynamic Analysis:** Soken's team conducts dynamic testing using automated tools and manual techniques to identify vulnerabilities. This can include unit testing, integration testing, and fuzzing.

8. **Testnet Deployment:** Test deployments of the smart contract are carried out on a testnet. This allows for real-world testing of the contract's functionality and behavior.
9. **Reporting:** Soken provides a detailed audit report documenting all findings, the severity level of each identified issue, and recommended mitigation strategies. The report is designed to be understandable for both technical and non-technical stakeholders.
10. **Remediation Guidance and Retesting:** Post-report, Soken assists the client's team in understanding the identified issues and suggesting fixes. Once the fixes are implemented, Soken's team retests the solutions to ensure they effectively mitigate the risks.

TERMINOLOGY

The classification of security vulnerabilities can be broken down into **low**, **medium**, **high**, and **critical** risks. It's important to note that the classification often depends on the **severity of the potential impact** (how much is at risk) and **the ease of exploitation** (how likely the vulnerability is to be exploited).

| IMPACT | LIKELIHOOD OF OCCURRENCE | | | |
|------------|--------------------------|------------|----------|-------------|
| | Rare | Unlikely | Likely | Very Likely |
| Low / Info | Low / Info | Low / Info | Medium | Medium |
| Medium | Low / Info | Medium | Medium | High |
| High | Medium | Medium | High | Critical |
| Critical | Medium | High | Critical | Critical |

Vulnerabilities by impact and likelihood of occurrence:

- **Critical:** These vulnerabilities often expose severe weaknesses in the contract, leading to massive financial or operational losses. Exploiting these vulnerabilities typically does not require sophisticated strategies, just knowledge of their existence.
- **High:** Highs might pose challenges for attackers to exploit, but if successfully exploited, they can greatly disrupt the execution of the smart contract, sometimes with significant consequences.
- **Medium:** These types of vulnerabilities are critical to resolve but typically do not lead to financial loss or unauthorized data manipulation. They might, however, degrade the user experience or cause unexpected behavior.

- **Low:** These vulnerabilities are generally linked to unused code or outdated snippets. They do not typically pose a direct risk to the execution of the contract but should still be addressed.

Vulnerabilities by status:

Each vulnerability is assigned a status:

- **Open:** the issue has not been addressed by the project team
- **Resolved:** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- **Closed:** the issue was acknowledged by the project team but no further actions have been taken.

FINDINGS SUMMARY

| SEVERITY | NUMBER OF FINDINGS | STATUS |
|---------------|--------------------|--------|
| High | 4 | Open |
| Medium | 2 | Open |
| Low | 3 | Open |
| Informational | 3 | Open |
| TOTAL: 12 | | |

SECURITY FINDINGS

High severity

1. Insufficient Gas Griefing Attack

Description:

The NativeMetaTransaction contract demonstrates vulnerability to Insufficient Gas Griefing Attacks. In this vulnerability, an attacker can initiate transactions with insufficient gas, causing the transaction to fail after consuming the gas, which may lead to various risks including replay attacks.

Specifically, the contract fails to handle transactions where the gas provided is not enough to cover the transaction's completion. With the nonces only incrementing after the `require(verify(...), "Signer and signature do not match")` statement, failed transactions due to out-of-gas errors don't revert the nonce, leaving the contract open to potential replay attacks.

Impact:

- **Transaction Failure:** Deliberately sending transactions with insufficient gas can cause transaction failure, wasting the gas used and causing disruption to the contract's operation.
- **Replay Attacks:** Since nonces do not revert upon transaction failure due to out-of-gas errors, this allows for potential replay attacks. An attacker can use the same nonce with a different transaction, leading to unauthorized or unintended transactions being executed.
- **Denial of Service (DoS):** Persistent transaction failures can lead to a denial-of-service scenario where legitimate users cannot successfully interact with the contract.

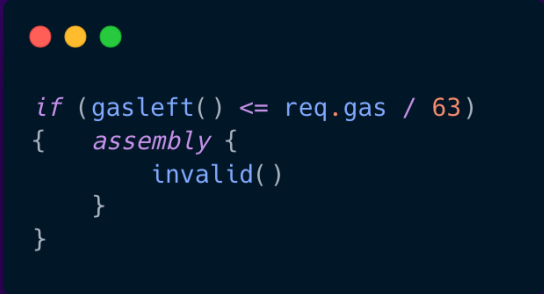
For reference: <https://solodit.xyz/issues/h-04-eip712metatransactionexecutemetatransaction-failed-txs-are-open-to-replay-attacks-code4rena-rolla-rolla-contest-git>

Mitigation:

To protect the contract against Insufficient Gas Griefing Attacks, consider the following mitigation measures:

- **Gas Checks:** Implement mechanisms to check if the transactions are supplied with sufficient gas, and reject those that don't meet the minimum required gas.
- **Enhanced Nonce Management:** Consider deploying enhanced nonce management mechanisms to ensure that nonces of failed transactions are handled securely, preventing potential replay attacks. This could involve invalidating the nonces of failed transactions or implementing a more secure nonce generation and validation method.

Consider adding the following verification between require and return lines of executeMetaTransaction function:



```
if (gasleft() <= req.gas / 63)
{
    assembly {
        invalid()
    }
}
```


2. Reentrancy on meta TXs execution can lead to loss of funds

Description:

The `executeMetaTransaction` function of the `NativeMetaTransaction` contract is at risk of reentrancy attacks due to its design. During the execution of this function, external calls are made without the implementation of safeguards, like the reentrancy guard.

Specifically, the code uses the `address(this).call` method without the necessary precautions to prevent reentrancy. Attackers could exploit this vulnerability by initiating a recursive call back into the function, manipulating the contract's state (like nonces), and leading to unexpected behavior or even loss of funds.

Impact:

- **Unauthorized Actions:** A reentrancy attack might allow malicious actors to perform unauthorized actions within the contract, leading to unexpected behavior of the `executeMetaTransaction` function.
- **State Manipulation:** An attacker can manipulate the contract's state, including the nonces mapping, facilitating subsequent attacks, including replay attacks.
- **Loss of Funds:** Successful reentrancy attacks may lead to the loss of funds held within the contract if the function is dealing with value transfers, affecting both users and the contract owner negatively.

Proof of Concept:

The attacker's contract would look something like this:

Mitigation:

Reentrancy Guard: Implement a reentrancy guard modifier to secure the function against reentrancy attacks. The reentrancy guard ensures that a function cannot be re-entered during its execution.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract AttackerReentrant {
    address payable target;
    bytes functionSignature;

    constructor(address payable _target, bytes memory _functionSignature) {
        target = _target;
        functionSignature = _functionSignature;
    }

    fallback() external payable {
        if (address(target).balance > 0) {
            // Call the vulnerable function again (reentrancy)
            (bool success, ) = target.call(functionSignature);
            require(success, "Reentrant call failed");
        }
    }

    receive() external payable {}

    // Function to initiate the attack
    function attack() external payable {
        (bool success, ) = target.call{value: msg.value}(functionSignature);
        require(success, "Initial call failed");
    }
}

```

Considering the reentrancy guard is already present. It would require adding the nonReentrant modifier to the function.

3. Risk of Incorrect Interface Reporting Due to Insufficient Gas

Description:

The supportsInterface function implemented in ERC1155 contract is exposed to potential risks due to its reliance on the amount of gas supplied to it, without incorporating proper checks and safeguards against insufficient gas errors as per the EIP-165 specification. According to the EIP-165 specification, supportsInterface is allowed to consume up to 30,000 gas.

However, due to the intricacies of Ethereum's gas mechanism (specifically EIP-150), the gas sent to the function can be less than expected, causing the function to throw an "out of gas" error even when there's technically adequate gas for the function to execute correctly.

This situation may lead to inaccurate results from the `supportsInterface` function, causing it to incorrectly signal that a contract does not support a certain interface when, in reality, it does.

Impact:

- **Incorrect Interface Reporting:** The function might inaccurately report that a contract does not support a certain interface due to out-of-gas errors, even if the contract implements the interface. This can lead to improper interaction with the contract, as relying functions or contracts may misinterpret the supported interfaces.
- **Transaction Failure:** Transactions may fail due to the function throwing out-of-gas errors, leading to disrupted operations and user dissatisfaction.

Proof of Concept:

This issue has been discussed under ERC-165 Standard Interface Detection and a PoC has been provided with contracts and tests inside https://github.com/wighawag/ethereum_gas/blob/24bd188002e8fd077ad0dcebb6bc6de97c0d9bdd/test/testERC165.js

For clarity, I'm adding the part of it where it tests the issue:

```
tap.test('9600 gas', async(t) => {  
  t.test('should throw', async () => {  
    const {contract} = await deploy({from:deployer, gas}, 'ERC165MoreGasExample',  
9600); await expectThrow(tx(  
      {from: deployer, gas: 34249},  
      TestERC165, 'test',  
      contract.options.address,  
      '0xffffffff'));  
  });  
  
  t.test('it works when given enough gas', async() => {  
    const {contract} = await deploy({from:deployer, gas}, 'ERC165MoreGasExample',  
9600); await expectThrow(tx(  
      {from: deployer, gas: 1000000},  
      TestERC165, 'test',  
      contract.options.address,  
      '0xffffffff'));  
  })  
});
```

Mitigation:

Gas Checking Before Call: Implement a mechanism that checks the `gasleft()` before making the call and ensures it's sufficient according to the requirements. For example:

```
uint256 gasAvailable = gasleft();  
require(gasAvailable - gasAvailable / 64 >= 30000, "Not enough gas  
provided");
```

This method may require accurate computation of the gas required between `gasleft()` and the CALL operation.

Gas Checking After Call: Check `gasleft()` after the call to ensure there is sufficient gas remaining. For instance:

```
// Execute STATIC_CALL with 30,000 gas  
require(gasleft() > 30000/63, "Not enough gas  
left");
```

This approach works if the call throws due to insufficient gas. However, it demands careful coding of `supportsInterface` to avoid interference with the `gasleft()` check.

4. Gas limit DOS attack via unbounded operations

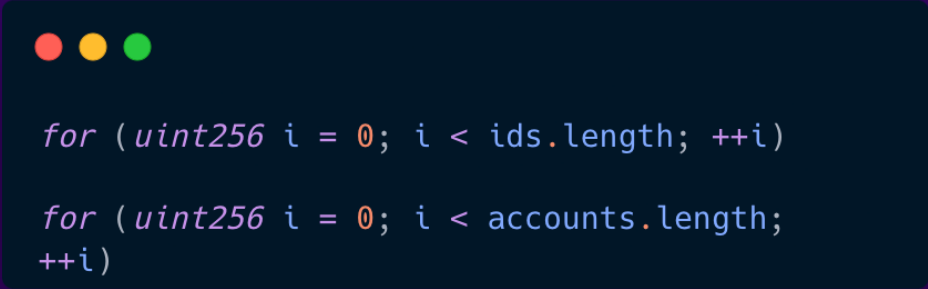
Description:

The project includes multiple functions that utilize loops without a defined or limited number of iterations, where the number of iterations depends on input from storage or external function calls. As Ethereum transactions have a gas limit, loops with an excessively high number of iterations could exhaust the transaction's available gas, leading to a transaction failure.

Functions such as `balanceOfBatch`, `safeBatchTransferFrom`, `_batchMint`, `_burnBatch`, `batchBurn`, and `batchMint` have unbounded loops that iterate based on the length of input arrays, which is a risky design pattern.

This issue is particularly concerning as it can inadvertently lead to a denial-of-service (DOS) condition where the functions become unresponsive or unusable if they consistently run out of gas, effectively stalling the contract's operation and potentially locking funds or assets.

These two loops are implemented in the above-mentioned functions:



```
for (uint256 i = 0; i < ids.length; ++i)

for (uint256 i = 0; i < accounts.length; ++i)
```

Impact:

- **Denial of Service (DOS):** A malicious actor or an unintentional scenario might lead to a transaction running out of gas, causing the function to fail and preventing further interactions with the contract.

- **Funds Lock:** If the contract consistently fails due to out-of-gas errors, users might be unable to withdraw or transfer their assets, leading to locked funds.
- **Increased Gas Costs:** Legitimate users might need to use higher gas fees to ensure their transactions do not fail, increasing the cost of interacting with the contract.

Mitigation:

Limiting Iterations:

- Implement a limit on the number of iterations allowed in each function call. This limit should be well-documented and communicated to users.
- Implement paginated functions to handle operations that would otherwise require excessive iterations, allowing users to process transactions in manageable chunks.

Gas Checks:

- Perform checks to ensure sufficient gas is available before executing operations prone to high gas consumption, thereby preventing unintentional out-of-gas errors.
- preventing potential replay attacks. This could involve invalidating the nonces of failed transactions or implementing a more secure nonce generation and validation method.

Medium severity

1. No account existence check on low-level call

Description:

Low-level calls `call`/`delegatecall`/`staticcall` return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling.

Impact:

The function `executeMetaTransaction` uses `call` (which returns boolean), however there is no account existence check for the destination address to.

If it doesn't exist, for some reason, `call` will still return true (not throw an exception) and successfully pass the return value check on the next line.

Considering the fact that this function is external and can be executed by anyone, there is risk of exploit or misuse.

For reference, see this related high-risk severity finding from Trail of Bit's audit of Hermes Network: <https://github.com/trailofbits/publications/blob/master/reviews/hermez.pdf>

Mitigation:

Check for account-existence before the `call()` to make this safely extendable to user-controlled address contexts in future.

2. The owner is a single point of failure and a centralization risk

Description:

Having a single EOA as the only owner of contracts is a large centralization risk and a single point of failure.

A single private key may be taken in a hack, or the sole holder of the key may become unable to retrieve the key when necessary. Consider changing to a multi-signature setup, or having a role-based authorization model.

Impact:

owner() is not behind a multisig and changes are not behind a timelock.

Even if protocol admins/developers are not malicious there is still a chance for Owner keys to be stolen. In such a case, the attacker can cause serious damage to the project due to important functions. There are 16 instances of this issue across the project.

Mitigation:

Add a time lock to critical functions. Admin-only functions that change critical parameters should emit events and have timelocks.

Allow only multi-signature wallets to call the function to reduce the likelihood of an attack.

<https://twitter.com/danielvf/status/1572963475101556738?s=20&t=V1kvzfJlsx-D2hfnG0OmuQ>

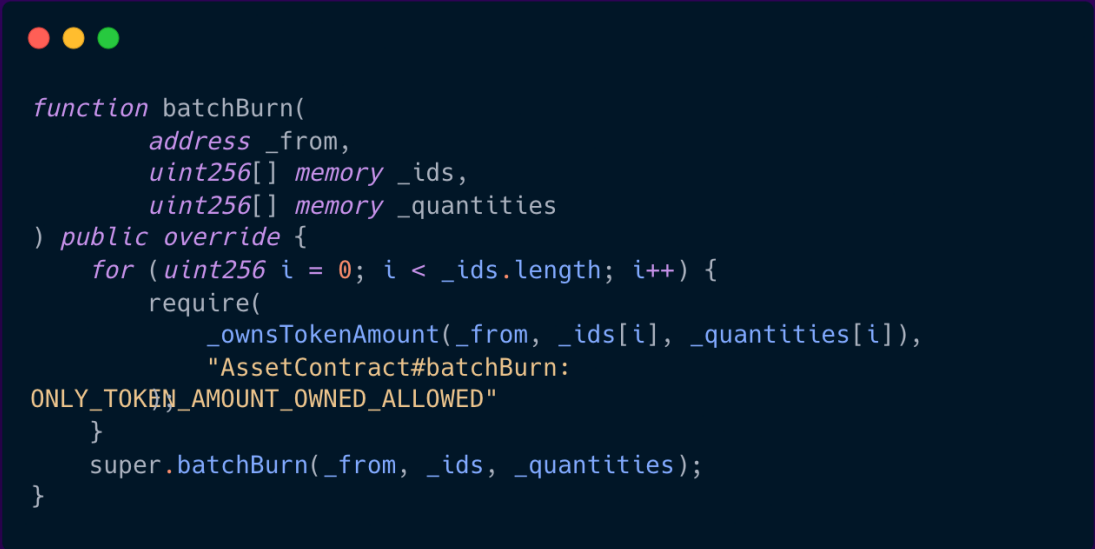
Also, detail them in the documentation and NatSpec comments.

Low severity

1. Doesn't check if `_ids` and `_quantities` length is the same in `batchBurn`

Description:

In the public function `batchBurn()` there is a check missing to make sure that both `uint256` parameters passed have the same length.



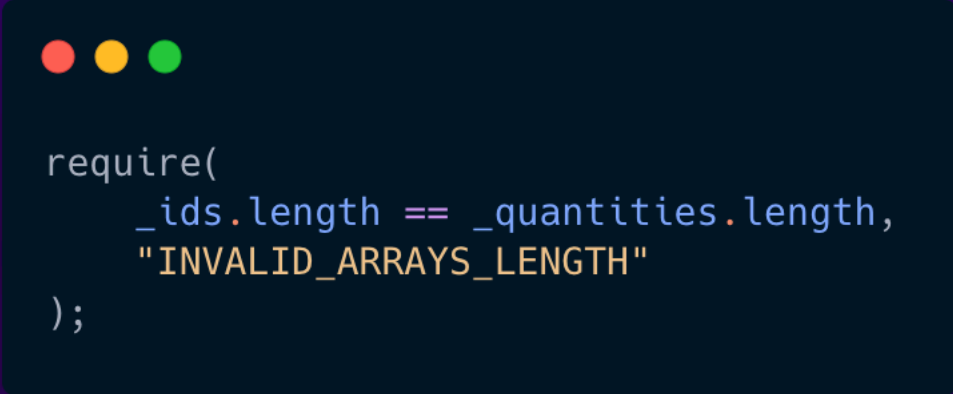
```
function batchBurn(
    address _from,
    uint256[] memory _ids,
    uint256[] memory _quantities
) public override {
    for (uint256 i = 0; i < _ids.length; i++) {
        require(
            _ownsTokenAmount(_from, _ids[i], _quantities[i]),
            "AssetContract#batchBurn:
ONLY_TOKEN_AMOUNT_OWNED_ALLOWED"
        )
        super.batchBurn(_from, _ids, _quantities);
    }
}
```

Impact:

It's assuming that for each iteration of `_ids.length` there is going to be a valid number for `_quantities`. The lack of a verification could cause unexpected values to be sent further to function `_ownsTokenAmount()`.

Mitigation:

Consider adding the verification before entering the loop with



```
require(  
    _ids.length == _quantities.length,  
    "INVALID_ARRAYS_LENGTH"  
);
```

2. Owner can renounce ownership

Description:

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The OpenZeppelin's Ownable used in this project contract implements `renounceOwnership`. This can represent a certain risk if the ownership is renounced for any other reason than by design.

Impact:

Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner. In this codebase there are 16 functions that would be affected in such scenario.

Mitigation:

It's recommended that the Owner is not able to call `renounceOwnership` without transferring the Ownership to other address before.

In addition, if a multi-signature wallet is used, calling `renounceOwnership` function should be confirmed for two or more users.

3. Lack of double step Transfer ownership pattern

Description:

The current ownership transfer process on the project involves the current owner calling `transferOwnership()` function.

```
function transferOwnership(address newOwner) public virtual onlyOwner
{
    require(
        newOwner != address(0),
        "Ownable: new owner is the zero address"
    );
    _setOwner(newOwner);
}
```

Impact:

If the nominated EOA account is not a valid one, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier

Mitigation:

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed.

This ensures the nominated EOA account is a valid and active account. This can be easily achieved by using OpenZeppelin's Ownable2Step contract instead of Ownable:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Ownable} from "./Ownable.sol";

abstract contract Ownable2Step is Ownable {
    address private _pendingOwner;

    event OwnershipTransferStarted(address indexed previousOwner, address indexed newOwner);
    function pendingOwner() public view virtual returns (address) {
        return _pendingOwner;
    }

    function transferOwnership(address newOwner) public virtual override onlyOwner {
        _pendingOwner = newOwner;
        emit OwnershipTransferStarted(owner(), newOwner);
    }

    function _transferOwnership(address newOwner) internal virtual override {
        delete _pendingOwner;
        super._transferOwnership(newOwner);
    }

    function acceptOwnership() public virtual {
        address sender = _msgSender();
        if (pendingOwner() != sender) {
            revert OwnableUnauthorizedAccount(sender);
        }
        _transferOwnership(sender);
    }
}
```

Informational issues

1. Consider using named mappings


Description:

Consider moving to solidity version 0.8.18 or later, and using named mappings to make it easier to understand the purpose of each mapping

2. Save big amount of gas by using uint256 instead of bool in mapping

Description:

Booleans are more expensive than uint256 or any type that takes up a full word because each write operation emits an extra SLOAD to first read the slot's contents, replace the bits taken up by the boolean, and then write back.



```
mapping(address => mapping(address => bool)) private
_operatorApprovals;
mapping(address => bool) public sharedProxyAddresses;
```

Impact:

By just using mapping(address => uint256), and having:

- uint256 NOT_APPROVED = 1 instead of false
- uint256 NOT_APPROVED = 2 instead of true,

there can be a considerable difference on gas saved.

Recommended read: <https://medium.com/@bloqarl/save-over-a-hundred-thousand-gas-with-this-solidity-gas-optimization-tip-ba791d6acafd?sk=43fc0ae06bef828126c85deec91ed3f>

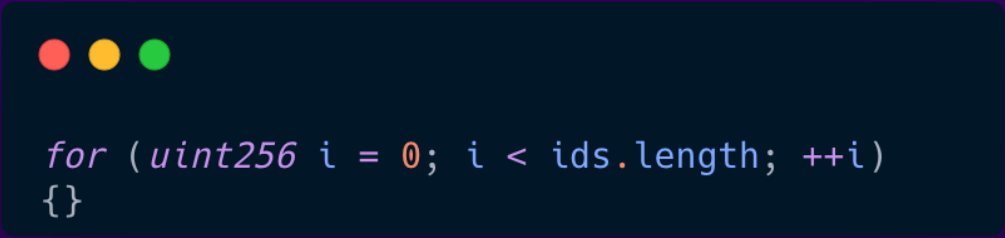
3. Cache array length outside of loop saves gas

Description:

If not cached, the solidity compiler will always read the length of the array during each iteration.

That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

This scenario is repeated many times throughout the code



```
for (uint256 i = 0; i < ids.length; ++i)
{ }
```

Mitigation:

Consider extracting that to a local variable outside the loop.

SOKEN CONTACT INFO

Website: www.soken.io

Telegram: @team_soken

GitHub: sokenteam

Twitter: @soken_team

