soken

SECURITY AUDIT

SOKEN.IO

EXECUTIVE OVERVIEW

Date: 28.09.2024

Token Name: RagdollToken

Contract Address: 0xF5E89006CBeFf2dabCfda0Def5Bf45Ebe7f8429f

Contract Platform: basescan

Contract Chain: mainnet

CONTRACT SECURITY

PRECISION LOSS DURING DIVISION BY LARGE NUMBERS

In Solidity, when dividing large numbers, precision loss can occur due to limitations in the Ethereum Virtual Machine (EVM). Solidity lacks native support for decimal or fractional numbers, leading to truncation of division results to integers. This can result in inaccuracies or unexpected behaviors, especially when the numerator is not significantly larger than the denominator.

Recommendations:

To mitigate precision loss during division in Solidity, consider scaling your numbers to work with integers effectively. This can involve multiplying both numerator and denominator by a factor to preserve precision during division. Additionally, employing external libraries or implementing custom logic for decimal arithmetic can offer more accurate results when dealing with fractional values in smart contracts.

MISSING EVENTS

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction’s log—a special data structure in the blockchain.These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.The contract was found to be missing these events on the function which would make it difficult or impossible to track these transactions off-chain.

Recommendations:

Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.

DEFINE CONSTRUCTOR AS PAYABLE

Developers can save around 10 opcodes and some gas if the constructors are defined as payable.However, it should be noted that it comes with risks because payable constructors can accept ETH during deployment.

Recommendations:

It is suggested to mark the constructors as payable to save some gas. Make sure it does not lead to any adverse effects in case an upgrade pattern is involved.

USE OF SAFEMATH LIBRARY

"SafeMath" library is found to be used in the contract. This increases gas consumption than traditional methods and validations if done manually.Also, Solidity "0.8.0" includes checked arithmetic operations by default, and this renders "SafeMath" unnecessary.

Recommendations:

We do not recommend using "SafeMath" library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.The compiler should be upgraded to Solidity version "0.8.0+" which automatically checks for overflows and underflows.

USE OWNABLE2STEP

"Ownable2Step" is safer than "Ownable" for smart contracts because the owner cannot accidentally transfer the ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

Recommendations:

It is recommended to use the OpenZeppelin library's "Ownable2Step" or "Ownable2StepUpgradeable" for managing ownership in your smart contracts. These library ensure a safer ownership transfer process by requiring the new owner to explicitly accept the transfer, reducing the risk of accidental transfers to incorrect addresses.

OPTIMIZING ADDRESS ID MAPPING

Combining multiple address/ID mappings into a single mapping using a struct enhances storage efficiency, simplifies code, and reduces gas costs, resulting in a more streamlined and cost-effective smart contract design. It saves storage slot for the mapping and depending on the circumstances and sizes of types, it can avoid a Gsset (20000 gas) per mapping combined. Reads

and subsequent writes can also be cheaper when a function requires both values and they fit in the same storage slot.

Recommendations:

It is suggested to modify the code so that multiple mappings using the address->id parameter are combined into a struct.

VARIABLES SHOULD BE IMMUTABLE

Constants and Immutables should be used in their appropriate contexts."constant" should only be used for literal values written into the code. "immutable" variables should be used for expressions, or values calculated in, or passed into the constructor.

Recommendations:

It is recommended to use "immutable" instead of "constant".

NAMED RETURN OF LOCAL VARIABLE SAVES GAS AS COMPARED TO RETURN STATEMENT

The function having a return type is found to be declaring a local variable for returning, which causes extra gas consumption. This inefficiency arises because creating and manipulating local variables requires additional computational steps and memory allocation.

Recommendations:

It is advisable to use a named returns statement in the function itself to save gas. Named returns optimize gas usage by directly returning variables declared at the function's signature, eliminating the need for additional memory allocation and manipulation of local variables.

FUNCTION SHOULD BE EXTERNAL

A function with "public" visibility modifier was detected that is not called internally."public" and "external" differs in terms of gas usage. The former use more than the latter when used with large arrays of data. This is due to the fact that Solidity copies arguments to memory on a "public" function while "external" read from calldata which a cheaper than memory allocation.

Recommendations:

If you know the function you create only allows for "external" calls, use the "external" visibility modifier instead of "public". It provides performance benefits and you will save on gas.

LONG REQUIRE/REVERT STRINGS

The "require()" and "revert()" functions take an input string to show errors if the validation fails.This strings inside these functions that are longer than "32 bytes" require at least one additional "MSTORE", along with additional overhead for computing memory offset, and other parameters.

Recommendations:

It is recommended to short the strings passed inside "require()" and "revert()" to fit under "32 bytes". This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

CHEAPER CONDITIONAL OPERATORS

During compilation, "x != 0" is cheaper than "x > 0" for unsigned integers in solidity inside conditional statements.

Recommendations:

Consider using "x != 0" in place of "x > 0" in "uint" wherever possible.

OUTDATED COMPILER VERSION

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Recommendations:

It is recommended to use a recent version of the Solidity compiler that should not be the most recent version, and it should not be an outdated version as well.
Using very old versions of Solidity prevents the benefits of bug fixes and newer security checks.
Consider using the solidity version "${pragma_version}", which patches most solidity vulnerabilities.

INTERNAL FUNCTIONS NEVER USED

The contract declared internal functions but was not using them in any of the functions or contracts.Since internal functions can only be called from inside the contracts, it makes no sense to have them if they are not used. This uses up gas and causes issues for auditors when understanding the contract logic.

Recommendations:

Having dead code in the contracts uses up unnecessary gas and increases the complexity of the overall smart contract. It is recommended to remove the internal functions from the contracts if they are never used.

REDUNDANT STATEMENTS

The contract contains redundant statements where types or identifiers are declared but not used, leading to no action being performed with them. These statements do not generate any code and can be safely removed to clean up the contract.

Recommendations:

It is advisable to remove the redundant statements to improve the readability and maintainability of the code. By eliminating these unnecessary declarations, the code becomes clearer and easier to understand. This practice helps in maintaining a high standard of code quality and ensures that every part of the code has a specific purpose.

USE OF FLOATING PRAGMA

Solidity source files indicate the versions of the compiler they can be compiled with using a pragma directive at the top of the solidity file. This can either be a floating pragma or a specific compiler version.The contract was found to be using a floating pragma which is not considered safe as it can be compiled with all the versions described.

Recommendations:

It is recommended to use a fixed pragma version, as future compiler versions may handle certain language constructions in a way the developer did not foresee.Using a floating pragma may

introduce several vulnerabilities if compiled with an older version.The developers should always use the exact Solidity compiler version when designing their contracts as it may break the changes in the future.Instead of "${pragma_in_use}" use "pragma solidity ${pragma_version}", which is a stable and recommended version right now.

AVOID RE-STORING VALUES

The function is found to be allowing re-storing the value in the contract's state variable even when the old value is equal to the new value. This practice results in unnecessary gas consumption due to the "Gsreset" operation (2900 gas), which could be avoided. If the old value and the new value are the same, not updating the storage would avoid this cost and could instead incur a "Gcoldsload" (2100 gas) or a "Gwarmaccess" (100 gas), potentially saving gas.

Recommendations:

To optimize gas usage, add a check to compare the old value with the new value before updating the storage. Only perform the storage update if the new value is different from the old value. This approach will prevent unnecessary storage writes and reduce gas consumption.

APPROVE FRONT-RUNNING ATTACK

The method overrides the current allowance regardless of whether the spender already used it or not, so there is no way to increase or decrease allowance by a certain value atomically unless the token owner is a smart contract, not an account. This can be abused by a token receiver when they try to withdraw certain tokens from the sender's account. Meanwhile, if the sender decides to change the amount and sends another "approve" transaction, the receiver can notice this transaction before it's mined and can extract tokens from both transactions, therefore, ending up with tokens from both the transactions. This is a front-running attack affecting the "ERC20 Approve" function.

Recommendations:

Only use the approve function of the ERC/BEP standard to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved).Token owner just needs to make sure that the first transaction actually changed allowance from N to 0, i.e., that the spender didn't manage to transfer some of N allowed tokens before the first transaction was mined. Such checking is possible using advanced blockchain explorers such as [Etherscan.io](<a

href='https://etherscan.io/'>https://etherscan.io/)Another way to mitigate the threat is to approve token transfers only to smart contracts with verified source code that does not contain logic for performing attacks like described above, and to accounts owned by the people you may trust.

MISSING UNDERSCORE IN NAMING VARIABLES

Solidity style guide suggests using underscores as the prefix for non-external functions and state variables (private or internal) but the contract was not found to be following the same.

Recommendations:

It is recommended to use an underscore for internal and private variables and functions to be in accordance with the Solidity style guide which will also make the code much easier to read.

CHEAPER INEQUALITIES IN REQUIRE()

The contract was found to be performing comparisons using inequalities inside the "require" statement. When inside the "require" statements, non-strict inequalities "(>=, <=)" are usually costlier than strict equalities "(>, <)".

Recommendations:

It is recommended to go through the code logic, and, if possible, modify the non-strict inequalities with the strict ones to save "~3" gas as long as the logic of the code is not affected.

FUNCTIONS CAN BE IN-LINED

The internal function was called only once throughout the contract. Internal functions cost more gas due to additional "JUMP" instructions and stack operations.

Recommendations:

Creating a function for a single call is not necessary if it can be in-lined. It is recommended to implement the logic using in-line codes to save gas.

STORAGE VARIABLE CACHING IN MEMORY

The contract is using the state variable multiple times in the function. "SLOADs" are expensive (100 gas after the 1st one) compared to "MLOAD"/"MSTORE" (3 gas each).

Recommendations:

Storage variables read multiple times inside a function should instead be cached in the memory the first time (costing 1 "SLOAD") and then read from this cache to avoid multiple "SLOADs".

This is a comprehensive report based on our automated and manual examination of cybersecurity vulnerabilities and framework flaws of the project's smart contract. Reading the full analysis report is essential to build your understanding of project's security level. It is crucial to take note, though we have done our best to perform this analysis and report, that you should not rely on the our research and cannot claim what it states or how we created it. Before making any judgments, you have to conduct your own independent research. We will discuss this in more depth in the following disclaimer - please read it fully.

DISCLAIMER: You agree to the terms of this disclaimer by reading this report or any portion thereof. Please stop reading this report and remove and delete any copies of this report that you download and/or print if you do not agree to these conditions. This report is for non-reliability information only and does not represent investment advice. No one shall be entitled to depend on the report or its contents, and Soken and its affiliates shall not be held responsible to you or anyone else, nor shall Soken provide any guarantee or representation to any person with regard to the accuracy or integrity of the report.
Without any terms, warranties or other conditions other than as set forth in that exclusion and Soken excludes hereby all representations, warrants, conditions and other terms (including, without limitation, guarantees implied by the law of satisfactory quality, fitness for purposes and the use of reasonable care and skills).
The report is provided as "as is" and does not contain any terms and conditions. Except as legally banned, Soken disclaims all responsibility and responsibilities and no claim against Soken is made to any amount or type of loss or damages (without limitation, direct, indirect, special, punitive, consequential or pure economic loses or losses) that may be caused by you or any other person, or any damages or damages, including without limitations (whether innocent or negligent).
Security analysis is based only on the smart contracts. No applications or operations were reviewed for security. No product code has been reviewed.

SOKEN CONTACT INFO

WEBSITE: WWW.SOKEN.IO

TELEGRAM: @SOKEN_SUPPORT

X (TWITTER): @SOKEN_TEAM