soken

SECURITY AUDIT

SOKEN.IO

EXECUTIVE OVERVIEW

Date: 06.09.2024

Token Name: FXGuys

Contract Address: 0x75D8e15F4cD5620FEa7147Ed0972B811d5ffF075

Contract Platform: etherscan

Contract Chain: mainnet

CONTRACT SECURITY

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.

FUNCTION RETURNS TYPE AND NO RETURN

This function specifies a "returns" keyword in the function signature but does not mention what to return anywhere in the function. This forces the function to always return a default value that was specified in the signature despite the calculations inside the function.

Recommendations:

f you don’t need the return value of the function, do not specify "returns" in the function signature.

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.

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.

HASH COLLISIONS WITH BYTES ARGUMENT ON ABI.ENCODEPACKED

Passing "bytes" data types into "abi.encodePacked()" can cause hash collisions when the byte arrays are user-controlled. The concatenation performed by "abi.encodePacked()" doesn't distinguish between byte array boundaries, leading to potential hash collisions.

Recommendations:

There can be multiple remediation's based on the code logic -
Avoid passing user-controlled byte arrays into abi.encodePacked().
Use abi.encode() instead of abi.encodePacked() to ensure that byte arrays are encoded with length prefixes.

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.

USE SELFBALANCE() INSTEAD OF ADDRESS(THIS).BALANCE

In Solidity, efficient use of gas is paramount to ensure cost-effective execution on the Ethereum blockchain. Gas can be optimized when obtaining contract balance by using "selfbalance()" rather than "address(this).balance" because it bypasses gas costs and refunds, which are not required for obtaining the contract's balance.

Recommendations:

To rectify this issue, developers are encouraged to replace instances of "address(this).balance" with "selfbalance()" wherever applicable. This optimization not only ensures streamlined gas operations but also contributes to substantial cost savings during contract execution.

IF-STATEMENT REFACTORING

In Solidity, we aim to write clear, efficient code that is both easy to understand and maintain. If statements can be converted to ternary operators. While using ternary operators instead of if/else statements can sometimes lead to more concise code, it's crucial to understand the trade-offs involved.

Recommendations:

To optimize your Solidity code, consider converting simple if/else statements to ternary operators, particularly for single-line arithmetic or logical operations. Utilizing ternary operators can improve code conciseness and readability. However, be mindful of code complexity and readability concerns. If the if/else statement is not single-line or involves multiple operations, retaining it for clarity is advisable.

GAS OPTIMIZATION FOR STATE VARIABLES

Plus equals ("+=") costs more gas than addition operator. The same thing happens with minus equals ("-=").

Recommendations:

Consider addition operator over plus equalssubtraction operator over minus equalsdivision operator over divide equalsmultiplication operator over multiply equals

PUBLIC BURN

The contract was found to be using public or an external "burn" function. The function was missing access control to prevent another user from burning their tokens. Also, the burn function was found to be using a different address than msg.sender.

Recommendations:

Consider adding access control modifiers to the burn function to prevent unauthorized users from burning tokens. Use the "onlyOwner" modifier from the OpenZeppelin "Ownable" contract to restrict access. The burn function should also use "msg.sender" in the "_from" argument to ensure that only the owner can call the function.

IN-LINE ASSEMBLY DETECTED

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. This should only be used for tasks that need it and if there is confidence in using it.Multiple vulnerabilities have been detected previously when the assembly is not properly used within the Solidity code; therefore, caution should be exercised while using them.

Recommendations:

Avoid using inline assembly instructions if possible because it might introduce certain issues in the code if not dealt with properly because it bypasses several safety features that are already implemented in Solidity.

USE BYTES.CONCAT() INSTEAD OF ABI.ENCODEPACKED

The contract is found to use "abi.encodePacked" for concatenating byte variables, which is less gas-efficient compared to using "bytes.concat". When concatenation isn&#39;t used for hashing operations, preferring "bytes.concat" can result in more optimized and cost-effective gas consumption.

Recommendations:

Refactor instances of "abi.encodePacked" used solely for concatenation to employ "bytes.concat". This change will improve gas efficiency for these operations. Reserve "abi.encodePacked" for cases where hashing operations follow the concatenation process.

CACHE ADDRESS(THIS) WHEN USED MORE THAN ONCE

The repeated usage of "address(this)" within the contract could result in increased gas costs due to multiple executions of the same computation, potentially impacting efficiency and overall transaction expenses.

Recommendations:

Optimize gas usage by caching the value of "address(this)" and reusing it throughout the contract, reducing redundant computations and thereby enhancing efficiency.

SUPPORTSINTERFACE() CALLS MAY REVERT

Be cautious when using "supportsInterface()" on a contract without ERC-165 implementation, as it may cause the call to revert. Even if the caller supports the function, there's a risk of malicious contracts consuming all available gas. Ensure safety by confirming that involved contracts adhere to ERC-165 before using "supportsInterface()".

Recommendations:

Verify contracts comply with ERC-165 before calling "supportsInterface()". This simple step minimizes the risk of reversion and potential gas consumption by malicious contracts, promoting a secure interaction with the function. It is advisable to use OZ "ERC165Checker".

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.

DUPLICATED REQUIRE/REVERT SHOULD BE A MODIFIER

The contract is found to have redundant "require()" or "if" and "revert()" statements verifying the same conditions multiple times. This not only increases the complexity but also results in higher gas costs during deployment and execution. Using Solidity's modifiers or internal functions, we can refactor these redundant checks to enhance code reusability and save deployment gas costs.

Recommendations:

Refactor the redundant "require()" or "revert()" statements into Solidity modifiers or reusable internal functions. This refactoring simplifies the code, makes it more readable, and can significantly reduce gas consumption during both deployment and execution by avoiding repeated condition checks.

REQUIRE INSTEAD OF REVERT

The "require" Solidity function guarantees the validity of the condition(s) passed as a parameter that cannot be detected before execution. It checks inputs, contract state variables, and return values from calls to external contracts.Using "require" instead of "revert" improves the overall readability of the contract code. The construction "if (condition) &#123; revert(); &#125;" is equivalent to "require(!condition);"

Recommendations:

Depending on the contract's validation checks, it is recommended to use the "require" function to handle input validations.

ERROR-PRONE TYPECASTING

Thevalue is found to be typecasted more than once which is not standard, making the contract’s logic more complex, and error-prone.

Recommendations:

It is recommended to go through the code logic to make sure the typecasting is working as expected and the resultant values are whole.

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.

ABI.ENCODEPACKED MAY CAUSE COLLISION

The contract is found to be using "abi.encodePacked()", which may lead to hash collisions when handling multiple variable-length arguments. Hash collisions occur because "abi.encodePacked()" concatenates all elements in order without distinguishing between different data types or their lengths. This issue is particularly concerning when the concatenated data is controlled or influenced by external users, but it can also arise internally due to repetitive patterns or complex data structures.

Recommendations:

To mitigate this risk, avoid allowing user-controlled data to be passed into "abi.encodePacked()". Use fixed-length arrays instead of dynamic arrays to reduce variability. Prefer using "abi.encode()"over "abi.encodePacked()" as it includes length prefixes for dynamic data types, providing more reliable encoding. Additionally, ensure that data structures include unique identifiers to differentiate between similar sequences of data, further reducing the potential for hash collisions.

PUBLIC CONSTANTS CAN BE PRIVATE

Public constant variables cost more gas because the EVM automatically creates getter functions for them and adds entries to the method ID table. The values can be read from the source code instead.

Recommendations:

If reading the values for the constants are not necessary, consider changing the "public" visibility to "private".

CUSTOM ERRORS TO SAVE GAS

The contract was found to be using "revert()" statements. Since Solidity "v0.8.4", custom errors have been introduced which are a better alternative to the revert.This allows the developers to pass custom errors with dynamic data while reverting the transaction and also making the whole implementation a bit cheaper than using "revert".

Recommendations:

It is recommended to replace all the instances of "revert()" statements with "error()" to save gas.

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.

VARIABLES DECLARED BUT NEVER USED

The contract has declared a variable but it is not used anywhere in the code. This represents dead code or missing logic. Unused variables increase the contract's size and complexity, potentially leading to higher gas costs and a larger attack surface

Recommendations:

To remediate this vulnerability, developers should perform a code review and remove any variables that are declared but never used.

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/</a>)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".

NAME MAPPING PARAMETERS

After Solidity 0.8.18, a feature was introduced to name mapping parameters. This helps in defining a purpose for each mapping and makes the code more descriptive.

Recommendations:

It is recommended to name the mapping parameters if Solidity 0.8.18 and above is used.

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