SolidityScan | Powered by CRED SHIELDS

Security Assessment Report

OnChainGM

21 Feb 2025

This security assessment report was prepared by SolidityScan.com, a cloud-based Smart Contract Scanner.

Table of Contents

UNNECESSARY CHECKED ARITHMETIC IN LOOP
USE SELFBALANCE() INSTEAD OF ADDRESS(THIS).BALANCE

05 Scan History

06 Disclaimer

01. Vulnerability Classification and Severity

Description

To enhance navigability, the document is organized in descending order of severity for easy reference. Issues are categorized as *Fixed*, *Pending Fix*, or *Won't Fix*, indicating their current status. *Won't Fix* denotes that the team is aware of the issue but has chosen not to resolve it. Issues labeled as *Pending Fix* state that the bug is yet to be resolved. Additionally, each issue's severity is assessed based on the risk of exploitation or the potential for other unexpected or unsafe behavior.

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

High-severity vulnerabilities pose a significant risk to both the Smart Contract and the organization. They can lead to user fund losses, may have conditional requirements, and are challenging to exploit.

Medium

The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior.

Low

The issue has minimal impact on the contract's ability to operate.

Informational

The issue does not affect the contract's operational capability but is considered good practice to address.

Gas

This category deals with optimizing code and refactoring to conserve gas.

02. Executive Summary

OnChainGM

0×107d604098c19953eF8534f30248F207549974e7

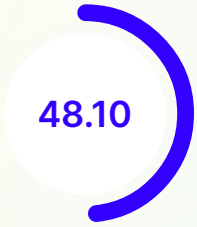
https://basescan.org/address/0×107d604098c19953eF8534f30...

Language	Audit Methodology	Contract Type
Solidity	**Static Scanning**	-

Website	Publishers/Owner Name	Organization
-	-	-

Contact Email

-



Security Score is AVERAGE

The SolidityScan score is calculated based on lines of code and weights assigned to each issue depending on the severity and confidence. To improve your score, view the detailed result and leverage the remediation solutions provided.

This report has been prepared for OnChainGM using SolidityScan to scan and discover vulnerabilities and safe coding practices in their smart contract including the libraries used by the contract that are not officially recognized. The SolidityScan tool runs a comprehensive static analysis on the Solidity code and finds vulnerabilities ranging from minor gas optimizations to major vulnerabilities leading to the loss of funds. The coverage scope pays attention to all the informational and critical vulnerabilities with over (100+) modules. The scanning and auditing process covers the following areas:

Various common and uncommon attack vectors will be investigated to ensure that the smart contracts are secure from malicious actors. The scanner modules find and flag issues related to Gas optimizations that help in reducing the overall Gas cost It scans and evaluates the codebase against industry best practices and standards to ensure compliance It makes sure that the officially recognized libraries used in the code are secure and up to date.

The SolidityScan Team recommends running regular audit scans to identify any vulnerabilities that are introduced after OnChainGM introduces new features or refactors the code.

03. Threat Summary

Threat Score

MEDIUM RISK

60/100

THREAT SUMMARY

Your smart contract has been assessed and assigned a **Medium Risk** threat score. The score indicates the likelihood of risk associated with the contract code.

Contract's source code is verified.

Source code verification provides transparency for users interacting with smart contracts. Block explorers validate the compiled code with the one on the blockchain. This also gives users a chance to audit the contracts, ensuring that the deployed code matches the intended functionality and minimizing the risk of malicious or erroneous contracts.

The contract cannot mint new tokens.

Minting functions are often utilized to generate new tokens, which can be allocated to specific addresses, such as user wallets or the contract owner's wallet. This feature is commonly employed in various decentralized finance (DeFi) and non-fungible token (NFT) projects to facilitate token issuance and distribution. The Presence of Minting Function module is designed to quickly identify the presence and implementation of minting functions in a smart contract. Mint functions play a crucial role in creating new tokens and transferring them to the designated user's or owner's wallet. This process significantly contributes to increasing the overall circulation of the tokens within the ecosystem.

The tokens cannot be burned in this contract.

The token contract incorporates a burn function that enables the intentional reduction of token amounts, consequently diminishing the total supply. The execution of this burn function contributes to the creation of scarcity within the token ecosystem, as the overall availability of the token decreases.

The contract can be compiled with a more recent Solidity version

The contract should be written using the latest Solidity pragma version as it comes with numerous bug fixes. Utilizing an outdated version exposes the contract to vulnerabilities associated with known issues that have been addressed in subsequent updates. Therefore, it is essential to stay current with the latest Solidity version to ensure the robustness and security of the contract against potential vulnerabilities.

This is not a proxy-based upgradable contract.

The Proxy-Based Upgradable Contract module is dedicated to identifying the presence of upgradeable contracts or proxy patterns within a smart contract. The utilization of upgradeable contracts or proxy patterns enables contract owners to make dynamic changes to various aspects, including functions, token circulation, and distribution, without requiring a complete redeployment of the contract.

Owners cannot blacklist tokens or users.

This module is designed to identify whether the owner of a smart contract has the capability to blacklist specific tokens or users. In a scenario where owners possess the authority to blacklist, all transactions related to the blacklisted entities will be immediately halted. Ownership privileges that include the ability to blacklist tokens or users can be a critical feature in certain use cases, providing the owner with control over potential malicious activities, compliance issues, or other concerns. However, in situations where this authority is abused or misapplied, it can lead to unintended consequences and user dissatisfaction.

Is not ERC-20 token.

A token is expected to adhere to the established standards of the ERC-20 token specification, encompassing the inclusion of all necessary functions with standardized names and arguments as defined by the ERC-20 standard.

This is not a Pausable contract.

Pausable contracts refer to contracts that can be intentionally halted by their owners, temporarily preventing token holders from engaging in buying or selling activities. This pause mechanism allows contract owners to exert control over the token's functionality, introducing a temporary suspension in trading activities for various reasons such as security concerns, updates, or regulatory compliance adjustments.

Critical functions that add, update, or delete owner/admin addresses are not detected.

A smart contract within the Web3 ecosystem that incorporates critical administrative functions can potentially compromise the transparency and intended objectives of the contract. It is imperative to conduct a thorough examination of these functions, especially in the realm of Web3 smart contracts. Minimizing administrative functions in a token contract within the Web3 framework can significantly reduce the likelihood of complications and enhance overall efficiency and clarity.

The contract cannot be self-destructed by owners.

The SELFDESTRUCT opcode is a critical operation in Ethereum smart contracts, allowing a contract to autonomously terminate itself. When invoked, this opcode deallocates the contract, freeing up storage and computational resources on the Ethereum blockchain. Notably, the remaining Ether in the contract is sent to a specified address, ensuring a responsible handling of funds.

The contract is not vulnerable to ERC-20 approve Race condition vulnerability.

The ERC-20 race condition arises when two or more transactions attempt to interact with the same ERC-20 token contract concurrently. This scenario can result in conflicts and unexpected behavior due to the non-atomic nature of certain operations in the contract. Atomicity refers to the concept that an operation is indivisible and occurs as a single, uninterruptible unit.

RENOUNCED OWNERSHIP

Renounced ownership indicates that the contract is truly decentralized, as the owner has relinquished control, ensuring that the contract's functionality and rules cannot be altered by administrators or any central authority.

No addresses contain more than 5% of circulating token supply.

Users with token balances exceeding 5% of the circulating token supply are critical to monitor, as their actions can significantly influence the token's price and ecosystem. Proper token distribution helps maintain a healthy market by preventing concentration of power and promoting fair participation.

The contracts are using functions that can only be called by the owners.

An overpowered owner risk occurs when a contract has numerous functions that only the owner can execute. This can lead to centralization issues and potential abuse, as the owner has disproportionate control over the contract's operations.

The contract does not have a cooldown feature.

Cooldown functions, a crucial aspect in the smart contract landscape, are employed to temporarily suspend trading activities or other contract workflows. The mechanism introduces a time-based delay, effectively preventing users from repeatedly executing transactions or engaging in rapid buying and selling of tokens. Cooldown functions are used to halt trading or other contract workflows for a certain amount of time so as to prevent users from repeatedly executing transactions or buying and selling tokens.

Owners cannot whitelist tokens or users.

This empowers the contract owner to selectively grant privileges to users, such as exemption from fees or access to unique contract features.

Owners cannot set or update Fees in the contract.

In the context of smart contracts, fees are essential components that may be associated with various functionalities, such as transactions, token transfers, or other specific actions. The ability for owners to set or update fees is particularly valuable in scenarios where fee adjustments are needed to align with market conditions, regulatory requirements, or project-specific considerations. The Owners Can Set or Update Fees module focuses on identifying the capability within a smart contract for owners to establish or modify fees. This feature allows contract owners to have control over the fee structure within the contract, providing flexibility and adaptability to changing circumstances.

Hardcoded addresses were found.

The inclusion of a fixed or hardcoded address within a smart contract has the potential to pose significant challenges in the future, particularly concerning the contract's adaptability and upgradability. This static reference to an address may impede the seamless implementation of updates or modifications to the contract, hindering its ability to evolve in response to changing requirements. Such rigidity may result in complications and obstacles when attempting to enhance or alter the smart contract's functionality over time.

The contract does not have any owner-controlled functions modifying token balances.

The Owners Updating Token Balance module is focused on identifying situations where a smart contract has functions controlled by owners that allow them to update token balances for other users or the contract. If a contract permits owners to manipulate token balances, it can have significant implications on user holdings and overall contract integrity. In some scenarios, contracts may provide owners with functions that enable the manual adjustment of token balances. While this feature can be legitimate for specific use cases, such as token distribution or rewards, it also introduces potential risks. Allowing owners to arbitrarily update token balances may lead to vulnerabilities, manipulation, or unintended changes in the token ecosystem.

OWNER WALLET TOKEN SUPPLY

A check on the owner's wallet balance exceeding a specific token amount can indicate a centralization risk, where the owner may have disproportionate control over the token supply, potentially leading to manipulation or abuse.

No such functions retrieving ownership were found.

The Function Retrieving Ownership module serves the purpose of swiftly and efficiently retrieving ownership-related information within a smart contract. This functionality is vital for projects seeking to access and manage ownership data seamlessly. Utilizing this module, developers can streamline the process of obtaining ownership details, contributing to the effective administration of ownership-related functions within the ecosystem.

IS SPAM CONTRACT

A spam NFT is an NFT that is considered low-quality or deceptive, cluttering the marketplace and potentially misleading users.

Absence of Malicious Typecasting.

Malicious typecasting, particularly the conversion of uint160 values to addresses, is a tactic often used by scammers to create deceptive addresses that can bypass standard detection mechanisms, facilitating fraudulent activities.

LIQUIDITY BURN STATUS

The liquidity burn status indicates whether the LP tokens for the scanned contract have been permanently removed or remain accessible. If burnt, the LP tokens are sent to an irrecoverable address, ensuring that the liquidity cannot be withdrawn, offering permanent stability and security to the project. If not burnt, the LP tokens could still be accessed and withdrawn, which might expose investors to risks of liquidity manipulation or removal.

LIQUIDITY LOCK STATUS

The liquidity status determines whether the liquidity for the scanned contract is securely locked or accessible. If locked, LP tokens are stored in a time-locked contract, preventing any withdrawals until the lock expires. This helps protect investors from sudden liquidity removal. If not locked, LP tokens remain accessible, allowing project developers or liquidity providers to withdraw liquidity at any time, potentially posing risks to investors.

No such functions having totalSupply function update were found.

A fixed supply token is critical when the token's value is tied to scarcity or when precise control over inflation or deflation is required. Without a fixed supply, the contract could introduce unexpected inflation, devalue the token, or erode trust in the token's consistency.

No such functions having gas abuse via malicious minting.

Gas abuse refers to patterns within smart contracts that manipulate gas consumption in ways that unnecessarily increase transaction costs for users. This can occur through various mechanisms designed to exploit gas inefficiencies or inflate gas usage, shifting the financial burden onto users without their knowledge.

No such functions having addresses with special access.

Special permissions granted to non-owner addresses allow them to execute specific functions with elevated access. This can introduce security risks, as these privileged addresses may perform critical operations that impact the contract's state or user funds. If not properly managed or monitored, these permissions could lead to unauthorized or malicious actions, compromising the contract's integrity.

No hidden owner detected

The Hidden Owner check identifies whether there are any hidden owner roles within the contract. Hidden ownership can allow unauthorized access and control over contract functions, which poses a risk to users and stakeholders.

COUNTERFEIT TOKEN

The contract is found to have the token symbol identical to that of official tokens, thereby falling under the category of counterfeit tokens. These counterfeit tokens can mislead users into believing they are interacting with legitimate, well-known cryptocurrencies, potentially leading to financial losses and damaging the reputation of the official token.

NO INTERACTION WITH THE SMART CONTRACT IN 30 DAYS

This check identifies if there has been any interaction with the smart contract within the past 30 days. Contracts that remain inactive for prolonged periods may indicate abandonment or lack of use, which could affect users' confidence and the contract's ongoing functionality.

Absence of external call risk in critical functions.

This check identifies risks associated with external calls within critical functions. External calls can introduce vulnerabilities such as unexpected state changes, or dependencies on external contracts, which may compromise the integrity and reliability of the function's execution.

Issue Type

SOLIDITY PRAGMA VERSION

Action Taken

Pending Fix

Description

The contract should be written using the latest Solidity pragma version as it comes with numerous bug fixes. Utilizing an outdated version exposes the contract to vulnerabilities associated with known issues that have been addressed in subsequent updates. Therefore, it is essential to stay current with the latest Solidity version to ensure the robustness and security of the contract against potential vulnerabilities.

Remediation

Update the Solidity pragma version to the latest stable version to benefit from the latest bug fixes and security enhancements.

contract.sol L2 - L2

```
1       // SPDX-License-Identifier: MIT
2       pragma solidity ^0.8.0;
3
4       contract OnChainGM {
```

Issue Type

OVERPOWERED OWNERS

Action Taken

Pending Fix

Description

An overpowered owner risk occurs when a contract has numerous functions that only the owner can execute. This can lead to centralization issues and potential abuse, as the owner has disproportionate control over the contract's operations.

Remediation

Review and minimize the number of critical functions accessible to owners, ensuring that these functions are necessary for contract management and do not pose undue risk to users' funds in the event of compromise or misuse. Implement multi-signature or governance mechanisms for critical actions to distribute authority and mitigate risk.

contract.sol L69 - L71

```
68          // Admin function to update GM fee
69          function updateGMFee(uint256 newFee) external onlyAdmin {
70              GM_FEE = newFee;
71          }
72
73          // Admin function to update GM multiplier (x)
```

contract.sol L74 - L76

```
73          // Admin function to update GM multiplier (x)
74          function updateGMointMultiplier(uint256 newMultiplier) external onlyAdmin {
75              GM_MULTIPLIER = newMultiplier;
76          }
77
78          // Function to get GMoint points for a user
```

Issue Type

HARDCODED ADDRESSES

Action Taken

Pending Fix

Description

The inclusion of a fixed or hardcoded address within a smart contract has the potential to pose significant challenges in the future, particularly concerning the contract's adaptability and upgradability. This static reference to an address may impede the seamless implementation of updates or modifications to the contract, hindering its ability to evolve in response to changing requirements. Such rigidity may result in complications and obstacles when attempting to enhance or alter the smart contract's functionality over time.

Remediation

To remediate, consider using configuration variables or function parameters instead of hardcoded addresses to allow for flexibility and easier updates in the future. If a fixed address is necessary, ensure that it is clearly documented and easily modifiable in the contract.

contract.sol L6 - L6

```
5           mapping(address => uint256) public lastGM; // Stores the last GM timestamp for each user
6           address public feeRecipient = 0x7500A83DF2aF99B2755c47B6B321a8217d876a85; // Address to receive the transaction fee
7           uint256 public GM_FEE = 0.000029 ether; // Fee amount for each GM transaction (now not constant)
8           uint256 public constant TIME_LIMIT = 24 hours; // Time limit of 24 hours for sending a GM
```

contract.sol L6 - L6

```
5           mapping(address => uint256) public lastGM; // Stores the last GM timestamp for each user
6           address public feeRecipient = 0x7500A83DF2aF99B2755c47B6B321a8217d876a85; // Address to receive the transaction fee
7           uint256 public GM_FEE = 0.000029 ether; // Fee amount for each GM transaction (now not constant)
8           uint256 public constant TIME_LIMIT = 24 hours; // Time limit of 24 hours for sending a GM
```

contract.sol L10 - L10

```
9
10          address public admin = 0x102f479312F69157Df8B804905A20FE5025881a5; // Admin address
11          uint256 public GM_MULTIPLIER = 1; // Multiplier for GMoint points (starts at 1x)
12
```

contract.sol L10 - L10

```
9
10          address public admin = 0x102f479312F69157Df8B804905A20FE5025881a5; // Admin address
11          uint256 public GM_MULTIPLIER = 1; // Multiplier for GMoint points (starts at 1x)
12
```

04. Findings Summary

0×107d604098c19953eF8534f30248F207549974e7
BASE (Base Mainnet) | View on Basescan

Security Score	Scan duration	Lines of code
48.10/100	**19 secs**	**79**

2	0	0	6	7	12
Crit	High	Med	Low	Info	Gas

This audit report has not been verified by the SolidityScan team. To learn more about our published reports. click here

ACTION TAKEN

0	0	0	27
Fixed	False Positive	Won't Fix	Pending Fix

S. No.	Severity	Bug Type	Instances	Detection Method	Status
C001	Critical	INCORRECT ACCESS CONTROL	2	Automated	Pending Fix
L001	Low	EVENT BASED REENTRANCY	2	Automated	Pending Fix
L002	Low	USE OF FLOATING PRAGMA	1	Automated	Pending Fix
L003	Low	MISSING EVENTS	2	Automated	Pending Fix
L004	Low	OUTDATED COMPILER VERSION	1	Automated	Pending Fix
I001	Informational	HARD-CODED ADDRESS DETECTED	2	Automated	Pending Fix
I002	Informational	BLOCK VALUES AS A PROXY FOR TIME	4	Automated	Pending Fix
I003	Informational	MISSING UNDERSCORE IN NAMING VARIABLES	1	Automated	Pending Fix
I004	Informational	RETURN INSIDE LOOP	1	Automated	Pending Fix
I005	Informational	USE CALL INSTEAD OF TRANSFER OR SEND	2	Automated	Pending Fix
G001	Gas	ARRAY LENGTH CACHING	1	Automated	Pending Fix
G002	Gas	AVOID RE-STORING VALUES	2	Automated	Pending Fix
G003	Gas	CHEAPER INEQUALITIES IN REQUIRE()	2	Automated	Pending Fix
G004	Gas	GAS OPTIMIZATION IN INCREMENTS	1	Automated	Pending Fix
G005	Gas	LONG REQUIRE/REVERT STRINGS	2	Automated	Pending Fix
G006	Gas	PUBLIC CONSTANTS CAN BE PRIVATE	1	Automated	Pending Fix

S. No.	Severity	Bug Type	Instances	Detection Method	Status
G008	Gas	UNNECESSARY CHECKED ARITHMETIC IN LOOP	1	Automated	*Pending Fix*
G009	Gas	USE SELFBALANCE() INSTEAD OF ADDRESS(THIS).BALANCE	1	Automated	*Pending Fix*

05. Vulnerability Details

Issue Type

INCORRECT ACCESS CONTROL

S. No.	Severity	Detection Method	Instances
C001	Critical	Automated	2

Upgrade your Plan to view the full report

2 Critical Issues Found

Please upgrade your plan to view all the issues in your report.

Upgrade

Issue Type

EVENT BASED REENTRANCY

S. No.	Severity	Detection Method	Instances
L001	Low	Automated	2

Bug ID	File Location	Line No.	Action Taken

Upgrade your Plan to view the full report

2 Low Issues Found

Please upgrade your plan to view all the issues in your report.

Upgrade

Issue Type

HARD-CODED ADDRESS DETECTED

S. No.	Severity	Detection Method	Instances
I001	Informational	Automated	2

Bug ID	File Location	Line No.	Action Taken

Upgrade your Plan to view the full report

2 Informational Issues Found

Please upgrade your plan to view all the issues in your report.

Upgrade

Issue Type

ARRAY LENGTH CACHING

S. No.	Severity	Detection Method	Instances
G001	Gas	Automated	1

Description

During each iteration of the loop, reading the length of the array uses more gas than is necessary. In the most favora ble scenario, in which the length is read from a memory variable, storing the array length in the stack can save about 3 gas per iteration. In the least favorable scenario, in which external calls are made during each iteration, the amount of gas wasted can be significant.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_22	contract.sol	L86 - L90	*Pending Fix*

contract.sol L86 - L90

```
85        function isUserExists(address user) private view returns (bool) {
86            for (uint256 i = 0; i < uniqueUsers.length; i++) {
87                if (uniqueUsers[i] == user) {
88                    return true;
89                }
90            }
91            return false;
92        }
```

Issue Type

AVOID RE-STORING VALUES

S. No.	Severity	Detection Method	Instances
G002	Gas	Automated	2

Description

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

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_8	contract.sol	L69 - L71	*Pending Fix*

contract.sol L69 - L71

```
68        // Admin function to update GM fee
69        function updateGMFee(uint256 newFee) external onlyAdmin {
70            GM_FEE = newFee;
71        }
72
73        // Admin function to update GM multiplier (x)
```

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_9	contract.sol	L74 - L76	⚠ *Pending Fix*

contract.sol L74 - L76

```
73        // Admin function to update GM multiplier (x)
74        function updateGMointMultiplier(uint256 newMultiplier) external onlyAdmin {
75            GM_MULTIPLIER = newMultiplier;
76        }
77
78        // Function to get GMoint points for a user
```

Issue Type

CHEAPER INEQUALITIES IN REQUIRE()

S. No.	Severity	Detection Method	Instances
G003	Gas	Automated	2

Description

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 `(>, <)`.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_12	contract.sol	L27 - L27	*Pending Fix*

contract.sol L27 - L27

```
26            require(msg.value == GM_FEE, "Incorrect ETH fee");
27            require(block.timestamp >= lastGM[msg.sender] + TIME_LIMIT, "Wait 24 hours before
      sending another GM");
28
29            lastGM[msg.sender] = block.timestamp; // Update the last GM timestamp
```

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_12	contract.sol	L47 - L47	⚠ *Pending Fix*

contract.sol L47 - L47

```
46            require(recipient != address(0), "Cannot send to zero address");
47            require(block.timestamp >= lastGM[msg.sender] + TIME_LIMIT, "Wait 24 hours before
      sending another GM");
48
49            lastGM[msg.sender] = block.timestamp; // Update the last GM timestamp
```

Issue Type

GAS OPTIMIZATION IN INCREMENTS

S. No.	Severity	Detection Method	Instances
G004	Gas	Automated	1

Description

`++i` costs less gas compared to `i++` or `i += 1` for unsigned integers. In `i++`, the compiler has to create a temporary variable to store the initial value. This is not the case with `++i` in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_13	contract.sol	L86 - L86	*Pending Fix*

contract.sol L86 - L86

```
85        function isUserExists(address user) private view returns (bool) {
86            for (uint256 i = 0; i < uniqueUsers.length; i++) {
87                if (uniqueUsers[i] == user) {
88                    return true;
```

Issue Type

LONG REQUIRE/REVERT STRINGS

S. No.	Severity	Detection Method	Instances
G005	Gas	Automated	2

Description

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 wit
h additional overhead for computing memory offset, and other parameters.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_20	contract.sol	L27 - L27	*Pending Fix*

contract.sol L27 - L27

```
26            require(msg.value == GM_FEE, "Incorrect ETH fee");
27            require(block.timestamp >= lastGM[msg.sender] + TIME_LIMIT, "Wait 24 hours before
      sending another GM");
28
29            lastGM[msg.sender] = block.timestamp; // Update the last GM timestamp
```

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_20	contract.sol	L47 - L47	⚠ *Pending Fix*

contract.sol L47 - L47

```
46            require(recipient != address(0), "Cannot send to zero address");
47            require(block.timestamp >= lastGM[msg.sender] + TIME_LIMIT, "Wait 24 hours before
      sending another GM");
48
49            lastGM[msg.sender] = block.timestamp; // Update the last GM timestamp
```

Issue Type

PUBLIC CONSTANTS CAN BE PRIVATE

S. No.	Severity	Detection Method	Instances
G006	Gas	Automated	1

Description

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.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_21	contract.sol	L8 - L8	*Pending Fix*

contract.sol L8 - L8

```
7         uint256 public GM_FEE = 0.000029 ether; // Fee amount for each GM transaction (now not constant)
8         uint256 public constant TIME_LIMIT = 24 hours; // Time limit of 24 hours for sending a GM
9
10        address public admin = 0x102f479312F69157Df8B804905A20FE5025881a5; // Admin address
```

Issue Type

STORAGE VARIABLE CACHING IN MEMORY

S. No.	Severity	Detection Method	Instances
G007	Gas	Automated	3

Description

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).

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_3	contract.sol	L25 - L41	*Pending Fix*

contract.sol L25 - L41

```
24        // Allows a user to send a GM to themselves, with a 24-hour restriction
25        function onChainGM() external payable {
26            require(msg.value == GM_FEE, "Incorrect ETH fee");
27            require(block.timestamp >= lastGM[msg.sender] + TIME_LIMIT, "Wait 24 hours before
      sending another GM");
28
29            lastGM[msg.sender] = block.timestamp; // Update the last GM timestamp
30
31            // Send the fee to the recipient address
32            payable(feeRecipient).transfer(GM_FEE);
33
34            // Increment the transaction count and add to unique users if new
35            if (!isUserExists(msg.sender)) {
36                uniqueUsers.push(msg.sender);
37            }
38            successfulTransactionsCount++;
39
40            emit OnChainGMEvent(msg.sender, msg.sender);
41        }
42
43        // Allows a user to send a GM to another user, with a 24-hour restriction
```

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_4	contract.sol	L44 - L61	⚠ *Pending Fix*

contract.sol L44 - L61

```
43        // Allows a user to send a GM to another user, with a 24-hour restriction
44        function onChainGMTo(address recipient) external payable {
45            require(msg.value == GM_FEE, "Incorrect ETH fee");
46            require(recipient != address(0), "Cannot send to zero address");
47            require(block.timestamp >= lastGM[msg.sender] + TIME_LIMIT, "Wait 24 hours before
    sending another GM");
48
49            lastGM[msg.sender] = block.timestamp; // Update the last GM timestamp
50
51            // Send the fee to the recipient address
52            payable(feeRecipient).transfer(GM_FEE);
53
54            // Increment the transaction count and add to unique users if new
55            if (!isUserExists(msg.sender)) {
56                uniqueUsers.push(msg.sender);
57            }
58            successfulTransactionsCount++;
59
60            emit OnChainGMEvent(msg.sender, recipient);
61        }
62
63        // Function to check the contract's balance
```

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_5	contract.sol	L85 - L92	⚠ *Pending Fix*

contract.sol L85 - L92

```
84        // Helper function to check if user exists in unique users array
85        function isUserExists(address user) private view returns (bool) {
86            for (uint256 i = 0; i < uniqueUsers.length; i++) {
87                if (uniqueUsers[i] == user) {
88                    return true;
89                }
90            }
91            return false;
92        }
93
94        // Function to get total successful transactions and unique users
```

Issue Type

UNNECESSARY CHECKED ARITHMETIC IN LOOP

S. No.	Severity	Detection Method	Instances
G008	Gas	Automated	1

Description

Increments inside a loop could never overflow due to the fact that the transaction will run out of gas before the varia ble reaches its limits. Therefore, it makes no sense to have checked arithmetic in such a place.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_2	contract.sol	L86 - L86	*Pending Fix*

contract.sol L86 - L86

```
85        function isUserExists(address user) private view returns (bool) {
86            for (uint256 i = 0; i < uniqueUsers.length; i++) {
87                if (uniqueUsers[i] == user) {
88                    return true;
```

Issue Type

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

S. No.	Severity	Detection Method	Instances
G009	Gas	Automated	1

Description

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.

Bug ID	File Location	Line No.	Action Taken
SSB_2990638_7	contract.sol	L65 - L65	*Pending Fix*

contract.sol L65 - L65

```
64      function contractBalance() public view returns (uint256) {
65          return address(this).balance;
66      }
67
```

06. Scan History

Critical High Medium Low Informational Gas

No	Date	Security Score	Scan Overview
1.	2025-02-21	48.10	2 0 0 6 7 12

07. Disclaimer

The Reports neither endorse nor condemn any specific project or team, nor do they guarantee the security of any specific project. The contents of this report do not, and should not be interpreted as having any bearing on, the economics of tokens, token sales, or any other goods, services, or assets.

The security audit is not meant to replace functional testing done before a software release.

There is no warranty that all possible security issues of a particular smart contract(s) will be found by the tool, i.e., It is not guaranteed that there will not be any further findings based solely on the results of this evaluation.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. There is no warranty or representation made by this report to any Third Party in regards to the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business.

In no way should a third party use these reports to make any decisions about buying or selling a token, product, service, or any other asset. It should be noted that this report is not investment advice, is not intended to be relied on as investment advice, and has no endorsement of this project or team. It does not serve as a guarantee as to the project's absolute security.

The assessment provided by SolidityScan is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. SolidityScan owes no duty to any third party by virtue of publishing these Reports.

As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent manual audits including manual audit and a public bug bounty program to ensure the security of the smart contracts.