SolidityScan | Powered by CRED SHIELDS

Security Assessment Report

PepeEvolutionsToken

9 Feb 2025

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

Table of Contents

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

PepeEvolutionsToken

0xB1EfA3A2fE00475228F816811F75aa0C5642B170

https://bscscan.com/address/0xB1EfA3A2fE00475228F816811F...

Language	Audit Methodology	Contract Type
Solidity	**Static Scanning**	-
Website	Publishers/Owner Name	Organization
-	-	-
Contact Email		
-		

Security Score is GREAT

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 PepeEvolutionsToken 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 PepeEvolutionsToken introduces new features or refactors the code.

03. Threat Summary

Threat Score

LOW RISK

86.21/100

THREAT SUMMARY

Your smart contract has been assessed and assigned a **Low 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 cannot 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 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 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.

✗ The contract's owner was not found.

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.

The Overpowered Owners module is dedicated to identifying situations where contract owners are endowed with excessive privileges through critical functions. Granting too many privileges to owners, especially via critical functions, might pose a significant risk to users' funds if the owners are compromised or if a rug-pulling attack occurs. In the context of smart contracts, owners often have access to critical functions that can impact the contract's functionality, token distribution, or other essential aspects. While providing owners with necessary permissions is crucial for contract management, it is equally important to avoid overempowering owners to mitigate potential risks.

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 not 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.

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.

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.

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 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.

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.

The token is not a 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.

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

ERC20 RACE CONDITION

Action Taken

Pending Fix

Description

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.

Remediation

Implement locking mechanisms or state variables to ensure that only one transaction can modify the token balances or allowances at a time, thereby preventing the race condition.

contract.sol L434 - L438

```
     */
    function approve(address spender, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
    }

    /**
```

contract.sol L565 - L567

```
     */
    function _approve(address owner, address spender, uint256 value) internal {
        _approve(owner, spender, value, true);
    }

    /**
```

contract.sol

L608 - L618

```
     */
    function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance < type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender, currentAllowance, value);
            }
            unchecked {
                _approve(owner, spender, currentAllowance - value, false);
            }
        }
    }
}

```

Issue Type

OVERPOWERED OWNERS

Action Taken

⚠ *Pending Fix*

Description

The Overpowered Owners module is dedicated to identifying situations where contract owners are endowed with excessive privileges through critical functions. Granting too many privileges to owners, especially via critical functions, might pose a significant risk to users' funds if the owners are compromised or if a rug-pulling attack occurs. In the context of smart contracts, owners often have access to critical functions that can impact the contract's functionality, token distribution, or other essential aspects. While providing owners with necessary permissions is crucial for contract management, it is equally important to avoid overempowering owners to mitigate potential risks.

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 L697 - L699

```
     */
    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }

    /**
```

contract.sol L705 - L710

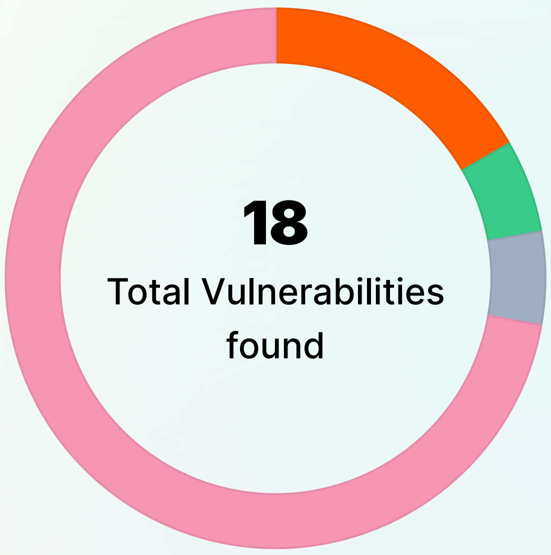
```
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        if (newOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(newOwner);
    }

    /**
```

04. Findings Summary

0xB1EfA3A2fE00475228F816811F75aa0C5642B170

BINANCE (Bsc Mainnet) | View on Bscscan

Security Score	Scan duration	Lines of code
95.53/100	21 secs	626



Crit	High	Med	Low	Info	Gas
0	3	0	1	1	13

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	18
✓ *Fixed*	*False Positive*	*Won't Fix*	⚠ *Pending Fix*

S. No.	Severity	Bug Type	Instances	Detection Method	Status
H001	● High	APPROVE FRONT-RUNNING ATTACK	3	Automated	⚠ *Pending Fix*
L001	● Low	USE OWNABLE2STEP	1	Automated	⚠ *Pending Fix*
I001	● Informational	IF-STATEMENT REFACTORING	1	Automated	⚠ *Pending Fix*
G001	● Gas	AVOID RE-STORING VALUES	2	Automated	⚠ *Pending Fix*
G002	● Gas	CHEAPER INEQUALITIES IN IF()	1	Automated	⚠ *Pending Fix*
G003	● Gas	DEFINE CONSTRUCTOR AS PAYABLE	1	Automated	⚠ *Pending Fix*
G004	● Gas	FUNCTIONS CAN BE IN-LINED	3	Automated	⚠ *Pending Fix*
G005	● Gas	INTERNAL FUNCTIONS NEVER USED	3	Automated	⚠ *Pending Fix*
G006	● Gas	OPTIMIZING ADDRESS ID MAPPING	2	Automated	⚠ *Pending Fix*
G007	● Gas	STORAGE VARIABLE CACHING IN MEMORY	2	Automated	⚠ *Pending Fix*

05. Vulnerability Details

Issue Type

APPROVE FRONT-RUNNING ATTACK

S. No.	Severity	Detection Method	Instances
H001	● High	Automated	3

Description

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 n otice this transaction before it's mined and can extract tokens from both transactions, therefore, ending up with toke ns from both the transactions. This is a front-running attack affecting the `ERC20 Approve` function.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_16	contract.sol	L434 - L438	⚠ *Pending Fix*

contract.sol   L434 - L438

```
     */
    function approve(address spender, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
    }

    /**
```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_17	contract.sol	L565 - L567	⚠ *Pending Fix*

contract.sol L565 - L567

```
         */
        function _approve(address owner, address spender, uint256 value) internal {
            _approve(owner, spender, value, true);
        }

        /**
```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_18	contract.sol	L608 - L618	⚠ *Pending Fix*

contract.sol L608 - L618

```
         */
        function _spendAllowance(address owner, address spender, uint256 value) internal virtu
    al {
            uint256 currentAllowance = allowance(owner, spender);
            if (currentAllowance < type(uint256).max) {
                if (currentAllowance < value) {
                    revert ERC20InsufficientAllowance(spender, currentAllowance, value);
                }
                unchecked {
                    _approve(owner, spender, currentAllowance - value, false);
                }
            }
        }
    }

```

Issue Type

USE OWNABLE2STEP

S. No.	Severity	Detection Method	Instances
L001	● Low	Automated	1

Description

`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.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_9	contract.sol	L725 - L725	*Pending Fix*

contract.sol L725 - L725

```

contract PepeEvolutionsToken is ERC20,Ownable {
    constructor( address initialOwner)
        ERC20("Pepe Evolutions", "PEPEV") Ownable(initialOwner)
```

Issue Type

IF-STATEMENT REFACTORING

S. No.	Severity	Detection Method	Instances
I001	● Informational	Automated	1

Description

In Solidity, we aim to write clear, efficient code that is both easy to understand and maintain. If statements can be co nverted 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.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_3	contract.sol	L505 - L515	⚠ *Pending Fix*

contract.sol

L505 - L515

```
504
505         if (to == address(0)) {
506             unchecked {
507                 // Overflow not possible: value <= totalSupply or value <= fromBalance <= totalSupply.
508                 _totalSupply -= value;
509             }
510         } else {
511             unchecked {
512                 // Overflow not possible: balance + value is at most totalSupply, which we know fits into a uint256.
513                 _balances[to] += value;
514             }
515         }
516
517         emit Transfer(from, to, value);
```

Issue Type

AVOID RE-STORING VALUES

S. No.	Severity	Detection Method	Instances
G001	● 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 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.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_10	contract.sol	L587 - L598	⚠ *Pending Fix*

contract.sol L587 - L598

```
     */
    function _approve(address owner, address spender, uint256 value, bool emitEvent) internal virtual {
        if (owner == address(0)) {
            revert ERC20InvalidApprover(address(0));
        }
        if (spender == address(0)) {
            revert ERC20InvalidSpender(address(0));
        }
        _allowances[owner][spender] = value;
        if (emitEvent) {
            emit Approval(owner, spender, value);
        }
    }

    /**
```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_11	contract.sol	L716 - L720	⚠ *Pending Fix*

contract.sol L716 - L720

```
         */
        function _transferOwnership(address newOwner) internal virtual {
            address oldOwner = _owner;
            _owner = newOwner;
            emit OwnershipTransferred(oldOwner, newOwner);
        }
    }

```

Issue Type

CHEAPER INEQUALITIES IN IF()

S. No.	Severity	Detection Method	Instances
G002	● Gas	Automated	1

Description

The contract was found to be doing comparisons using inequalities inside the if statement.
When inside the `if` statements, non-strict inequalities `(>=, <=)` are usually cheaper than the strict equalities `(>, <).`

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_15	contract.sol	L610 - L610	⚠ *Pending Fix*

contract.sol

L610 - L610

```
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance < type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender, currentAllowance, value);
```

Issue Type

DEFINE CONSTRUCTOR AS PAYABLE

S. No.	Severity	Detection Method	Instances
G003	● Gas	Automated	1

Description

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 deployme nt.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_7	contract.sol	L726 - L730	⚠ *Pending Fix*

contract.sol L726 - L730

```
contract PepeEvolutionsToken is ERC20,Ownable {
    constructor( address initialOwner)
        ERC20("Pepe Evolutions", "PEPEV") Ownable(initialOwner)
    {
        _mint(initialOwner,8e9*1e18 );
    }
}
```

Issue Type

FUNCTIONS CAN BE IN-LINED

S. No.	Severity	Detection Method	Instances
G004	● Gas	Automated	3

Description

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

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_4	contract.sol	L528 - L533	⚠ *Pending Fix*

contract.sol

L528 - L533

```
     */
    function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }

    /**
```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_5	contract.sol	L608 - L618	⚠ *Pending Fix*

contract.sol L608 - L618

```
     */
    function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance < type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender, currentAllowance, value);
            }
            unchecked {
                _approve(owner, spender, currentAllowance - value, false);
            }
        }
    }
}

```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_6	contract.sol	L684 - L688	⚠ *Pending Fix*

contract.sol L684 - L688

```
     */
    function _checkOwner() internal view virtual {
        if (owner() != _msgSender()) {
            revert OwnableUnauthorizedAccount(_msgSender());
        }
    }

    /**
```

Issue Type

INTERNAL FUNCTIONS NEVER USED

S. No.	Severity	Detection Method	Instances
G005	● Gas	Automated	3

Description

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 u sed. This uses up gas and causes issues for auditors when understanding the contract logic.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_12	contract.sol	L134 - L136	⚠ *Pending Fix*

contract.sol L134 - L136

```
133
134        function _msgData() internal view virtual returns (bytes calldata) {
135            return msg.data;
136        }
137
138        function _contextSuffixLength() internal view virtual returns (uint256) {
```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_13	contract.sol	L138 - L140	⚠ *Pending Fix*

contract.sol L138 - L140

```

    function _contextSuffixLength() internal view virtual returns (uint256) {
        return 0;
    }
}

```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_14	contract.sol	L543 - L548	⚠ *Pending Fix*

contract.sol L543 - L548

```
     */
    function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }

    /**
```

Issue Type

OPTIMIZING ADDRESS ID MAPPING

S. No.	Severity	Detection Method	Instances
G006	● Gas	Automated	2

Description

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 (2 0000 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.

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_1	contract.sol	L337 - L337	⚠ *Pending Fix*

contract.sol

L337 - L337

```
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {
    mapping(address account => uint256) private _balances;

    mapping(address account => mapping(address spender => uint256)) private _allowances;
```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_2	contract.sol	L339 - L339	⚠ *Pending Fix*

contract.sol L339 - L339

```

    mapping(address account => mapping(address spender => uint256)) private _allowances;

    uint256 private _totalSupply;
```

Issue Type

STORAGE VARIABLE CACHING IN MEMORY

S. No.	Severity	Detection Method	Instances
G007	● Gas	Automated	2

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_2246848_8	contract.sol	L490 - L518	*Pending Fix*

contract.sol

L490 - L518

```
     */
    function _update(address from, address to, uint256 value) internal virtual {
        if (from == address(0)) {
            // Overflow check required: The rest of the code assumes that totalSupply neve
r overflows
            _totalSupply += value;
        } else {
            uint256 fromBalance = _balances[from];
            if (fromBalance < value) {
                revert ERC20InsufficientBalance(from, fromBalance, value);
            }
            unchecked {
                // Overflow not possible: value <= fromBalance <= totalSupply.
                _balances[from] = fromBalance - value;
            }
        }

        if (to == address(0)) {
            unchecked {
                // Overflow not possible: value <= totalSupply or value <= fromBalance <=
totalSupply.
                _totalSupply -= value;
            }
        } else {
```

contract.sol L490 - L518

```
            }
        } else {
            unchecked {
                // Overflow not possible: balance + value is at most totalSupply, which we know fi
ts into a uint256.
                _balances[to] += value;
            }
        }

        emit Transfer(from, to, value);
    }

```

Bug ID	File Location	Line No.	Action Taken
SSB_2246848_8	contract.sol	L490 - L518	⚠ *Pending Fix*

contract.sol L490 - L518

```
     */
    function _update(address from, address to, uint256 value) internal virtual {
        if (from == address(0)) {
            // Overflow check required: The rest of the code assumes that totalSupply neve
r overflows
            _totalSupply += value;
        } else {
            uint256 fromBalance = _balances[from];
            if (fromBalance < value) {
                revert ERC20InsufficientBalance(from, fromBalance, value);
            }
            unchecked {
                // Overflow not possible: value <= fromBalance <= totalSupply.
                _balances[from] = fromBalance - value;
            }
        }

        if (to == address(0)) {
            unchecked {
                // Overflow not possible: value <= totalSupply or value <= fromBalance <=
totalSupply.
                _totalSupply -= value;
            }
        } else {
            unchecked {
```

06. Scan History

● Critical ● High ● Medium ● Low ● Informational ● Gas

No	Date	Security Score	Scan Overview
1.	2025-02-09	**95.53**	● 0 ● 3 ● 0 ● 1 ● 1 ● 13

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.