

SMART CONTRACT AUDIT FOR ERC-20 GOVERNANCE TOKEN. (GST)

Jaymin S Chandaria

Contact information

Email: [jchandaria.work@gmail.com](mailto:jchandaria.work@gmail.com)

Phone/WhatsApp: +91-9483014950

### Abstract

The GST token is an ERC-20 token that implements several advanced features, including pausing, minting, voting, and role-based access control. This document provides an audit of the smart contract's code, the token's functionality and how to use it.

*Keywords:* Governance, ERC-20, Access Control

### Pre-requisites

Before you can interact with the GST token, you must have a basic understanding of the following concepts:

1. ERC-20 tokens: A standard interface for fungible tokens on the Ethereum blockchain.
2. Pausable tokens: Tokens that can be paused by an authorized account, preventing transfers and other transactions.
3. Minting: The process of creating new tokens.
4. Voting: The process of casting votes on proposals.
5. Role-based access control: A mechanism for controlling access to different functions based on assigned roles.

Here, the libraries from OpenZeppelin have been made use of.

## GST token smart contract audit

This will be the document to refer to for the detailed suit of the smart contract

### Token Features

The GST token implements the following features:

- ➔ Pausing: The token can be paused by an account with the `'PAUSER_ROLE'`. When paused, transfers and other transactions are prevented.
- ➔ Minting: New tokens can be minted by an account with the `'MINTER_ROLE'`.
- ➔ Voting: Token holders can vote on proposals by calling the `'vote'` function.
- ➔ Role-based access control: The token uses role-based access control to control access to different functions.

The following roles are defined:

- `'DEFAULT_ADMIN_ROLE'`: Has full control over the token.
- `'PAUSER_ROLE'`: Can pause and unpause the token.
- `'MINTER_ROLE'`: Can mint new tokens.

Sepolia Etherscan link: [GirlScriptToken \(GST\) Token Tracker | Etherscan](#)

Contract Address: 0x110Aa39698DFee9B894AEe808929B87838e7174

From: 0x00

To: 0xad71560099fbC33D35c726e75f69aA66d6dbBE69

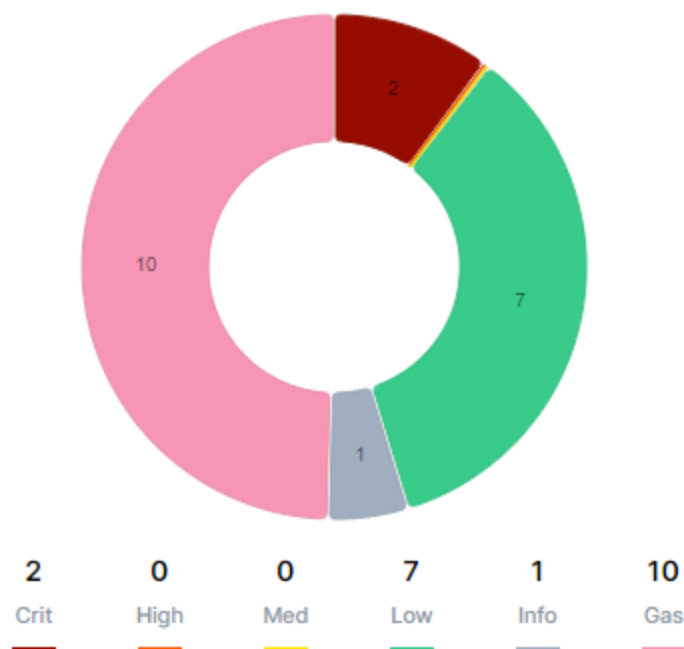
Quantity: 0.6942

Transaction hash of the contract deployment:

0x2e34530d0a9da61efa335e987bcaad596c9304e887a5246dc98d94beb3bcd799

Tool used for Audit: <https://solidityscan.com/>

Fig 1: Pie Chart distribution of the vulnerabilities by types/level.



According to this figure we infer that the contract does not have good gas optimization practices.

Fig 2: Scan Statistics

SCAN STATISTICS	
Security Score	67.89/100
Issue Count	19
Duration	0 second(s)
Lines of code	109

We see that:

1. There are **109** lines of code in the smart contract.
2. Security rating is ~**68%**.
3. The total count of issues/bugs are **19**.
4. The contract executes in 0 seconds/ the security scan was done in no time.

The gas vulnerabilities:

1. SSP\_3986\_2, SSP\_3986\_3, SSP\_3986\_4

Description: FUNCTION SHOULD BE EXTERNAL

A function with public visibility modifier was detected that is not called internally.

public and external differs in terms of gas usage. The former use more than the latter when used with large arrays of data. This is due to the fact that Solidity copies arguments to memory on a public function while external read from calldata which a cheaper than memory allocation. Here are the vulnerable code snippets:

```
1.
function unpause() public onlyRole(PAUSER_ROLE) {
    _unpause(); }
```

```

2.
function mint(address to, uint256 amount) public
onlyRole(MINTER_ROLE) {
    _mint(to, amount);
}

3.
function pause() public onlyRole(PAUSER_ROLE) {
    _pause();
}

```

## 2. SSP\_3986\_7

Description: 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. Here is the vulnerable code snippet:

```

    constructor(address defaultAdmin, address pauser, address
minter)
    ERC20("GirlScriptToken", "GST")
    ERC20Permit("GirlScriptToken")
    {
        _mint(msg.sender, 694200000000000000); // Set an initial
total supply and assign all initial tokens to the owner
        _grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
        _grantRole(PAUSER_ROLE, pauser);
        _grantRole(MINTER_ROLE, minter);
    }

```

## 3. SSP\_3986\_9

Description: LONG REQUIRE/REVERT STRINGS

The require() and revert() functions take an input string to show errors if the validation fails. These 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. Here is the vulnerable code snippet:

```
require(balanceOf(msg.sender) > 0, "Not enough tokens to  
create a proposal");
```

## 4. SSP\_3986\_16

Description: 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.

The following variable is affected: PAUSER\_ROLE. Vulnerable code snippet:

```
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
```

## 5. SSP\_3986\_17

Description: 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. The following variable is affected: MINTER\_ROLE. :::

```
bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
```

## 6. SSP\_3986\_18

Description: CHEAPER CONDITIONAL OPERATORS

During compilation,  $x \neq 0$  is cheaper than  $x > 0$  for unsigned integers in solidity inside conditional statements. Vulnerable code :::

```
require(balanceOf(msg.sender) > 0, "Not enough tokens to
create a proposal");
```

## 7. SSP\_3986\_19

Description - 1 : STORAGE VARIABLE CACHING IN MEMORY

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

Description - 2 : STORAGE VARIABLE CACHING IN MEMORY

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

Here is the vulnerable code snippet:

```
function vote(uint256 proposalId, bool inSupport) external {
    require(!hasVoted[msg.sender], "Already voted");
    require(proposalId < proposals.length, "Invalid
proposalId");

    hasVoted[msg.sender] = true;
    Proposal storage proposal = proposals[proposalId];

    if (inSupport) {
```



```

        proposal.votesFor += balanceOf(msg.sender);
    } else {
        proposal.votesAgainst += balanceOf(msg.sender);
    }

    emit Voted(msg.sender, inSupport);
}

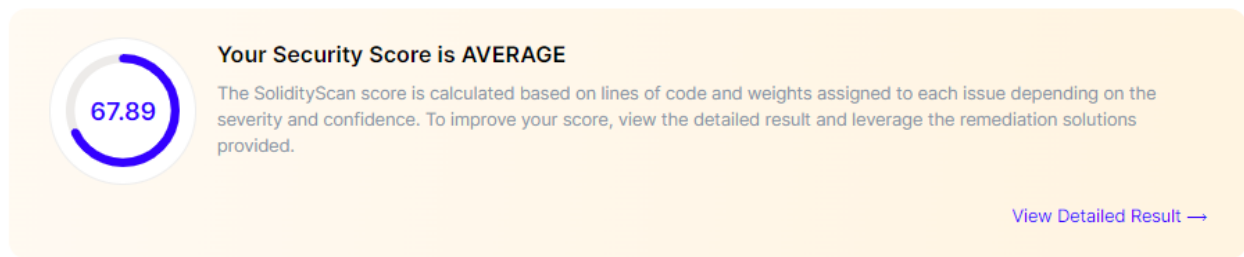
```

Fig 3: The remaining vulnerabilities

● INCORRECT ACCESS CONTROL	2	►
● USE OF FLOATING PRAGMA	1	►
● OUTDATED COMPILER VERSION	1	►
● MISSING EVENTS	4	►
● LONG NUMBER LITERALS	1	►

- INCORRECT ACCESS CONTROL
- USE OF FLOATING PRAGMA
- OUTDATED COMPILER VERSION
- MISSING EVENTS
- LONG NUMBER LITERALS

Fig 4: The scan summary



Note: I haven't used any paid services for auditing.