QuillAudits

Audit Report
November, 2022

For

gamr

Table of Content

Table of Content

Executive Summary

Project Name	Gamr BEP20 Smart Contract
Overview	The given contract BEP20Token is an BEP20 token on bsc network. Which is modified to use staking and unstake functionality.
Timeline	May 16, 2022 - July 11, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze Gamr token Contract codebase for quality, security, and correctness. *https://github.com/techplus-ng/gamr-coin/blob/master/src/_dapp/contracts/BEP20.sol* Commit hash: 0c94458387816d22f6badc7f5e5dec42622b42b8
Fixed In	*https://github.com/techplus-ng/gamr-coin/blob/devcont/src/_dapp/contracts/BEP20.sol* **Commit hash**: 5351ae843d95a7fd6c9da477741d88ed51116386

High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	1	2	2	2

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of BEP-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - BEP20

High Severity Issues

A.1: Contract issues related to functionality

Description

The contract is a BEP20 token with some modifications done on it. Contract gets compiled but while testing the unStake() function does not pass. The reason being there is another token used to transfer but according to the client it was not supposed to be there.

Also it was supposed to be upgradeable contract so as to suggest use openzeppelin UUPSupgradeable proxy with OwnableUpgradeable and ERC20Upgradeable libraries.

Remediation

According to the client the burnFrom() which is called in the unStake() function is supposed to be called on the same token and not on another token.
Also it is supposed to decrease the tax amount from totalSupply. If the csr is not used you can remove that functionality. Also keep in mind the structure of how burning tokens should work through unStake() function by burning tokens from just totalSupply.

Status

Resolved

Medium Severity Issues

A.2: Users are unable to unstake tokens because of the tax values

Line	Function - unStake()

868	`function unStake(uint id1) external virtual {`
	`  require(_board.parties[msg.sender].staked > 0 && _board.parties[msg.sender].stakes.length > id1 && id1 >= 0, "CGM: stake not found");`
	
	`  StakeData memory pStake = _board.parties[msg.sender].stakes[id1];`
	
	`  require(pStake.redeemed == false, "CGM: you can't redeem stake twice");`
	`  _board.parties[msg.sender].stakes[id1].redeemed = true;`
	`  uint256 currentDays = currentDaysCount(pStake.duration, block.timestamp, pStake.timestamp);`
	
	`  require(currentDays > 0,"CGM: same day unstake not allowed");`
	
	`  uint256 reward = currentDays.mul(pStake.intPerDay).div(scale);  //  282828 / 2**64 = 0.`
	`  uint256 staked = pStake.totalStaked;`
	
	`  _board.totalStaked = _board.totalStaked.sub(staked);`
	`  _board.parties[msg.sender].staked = _board.parties[msg.sender].staked.sub(staked);`
	
	`  if(_board.parties[msg.sender].staked == 0){`
	`  _board.totalStakers = _board.totalStakers.sub(1);`
	`  }`
	
	`  if(currentDays < pStake.duration){`
	`  uint256 reward_tax = reward.mul(rewardTax).div(200);`
	`  uint256 staked_tax = staked.mul(stakeTax).div(300);`
	
	`  reward = reward.sub(reward_tax);`
	`  staked = staked.sub(staked_tax);`

Description

In function unStake() users can unstake their staked tokens. But because of the high tax value they are unable to do so.
For example,
Say if user1 staked 100e18 tokens in the calculation the reward value is multiplied like 110*500000/100 = 550000 so function overflows at line reward = reward.sub(reward_tax) and staked = staked.sub(staked_tax).

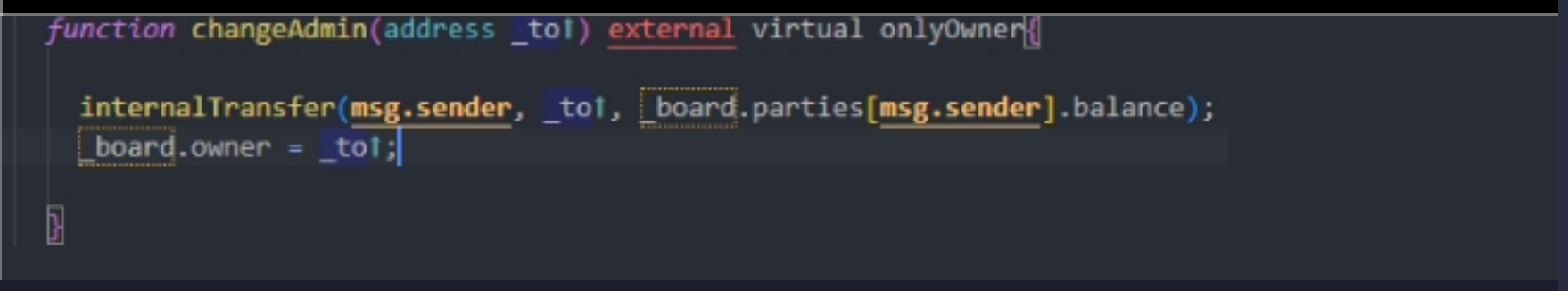
Remediation

To resolve the issue please put accurate values. If you want tax to be 0.5% then (100*5)/1000 you need to correctly place decimals

Status

Resolved

A.3: changeAdmin() should be two step process

Line	Function - changeAdmin()
770	

```
function changeAdmin(address _to1) external virtual onlyOwner{

  internalTransfer(msg.sender, _to1, board.parties[msg.sender].balance);
  board.owner = _to1;

}
```

Description

In contract changeAdmin() function helps to move the funds to another address. It is quite possible that it might be transferred to zero address mistakenly or any wrong address.

Remediation

A two factor mechanism could be adopted. checkAdmin() to assign a role to a new address, while the original admin still has a role.
Then
updateAdmin should be a function for acceptance of roles and the former admin is revoked and transfers the tokens.

Status

Resolved

Low Severity Issues

A.4: Wide scope in solidity pragma

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. Also in the contract the version used has wide scope from 0.4.12 to 0.9.0. 0.9.0 is not here and using old versions of solidity can affect the system.

Remediation

Here the in-scope contract has an unlocked pragma with wide scope, it is recommended to lock the same and use the latest version. If necessary, refactor the current code base to only use locked pragma with the stable version.

Status

Resolved

A.5: Unstaking does not delete the stake length for the user

Description

In function unStake() when user unstakes the token. The tokens do get transferred to the user but the function getStakesOf() returns the same number of stake length.

Remediation

If the stake is unstaked then the length of the stakes also should be decreased.

Status

Resolved

Informational Issues

A.6: Function name does not match what it does

Line	Function - getStakesOf()

770	`function getStakesOf(address account) public view returns (uint256) {`
	`  return _board.parties[account].stakes.length;`
	`}`

Description

In function getStakesOf() the contract gets the length of the stakes which the user has staked. But the function implies that it is returning all the stakes and not length of the stakes

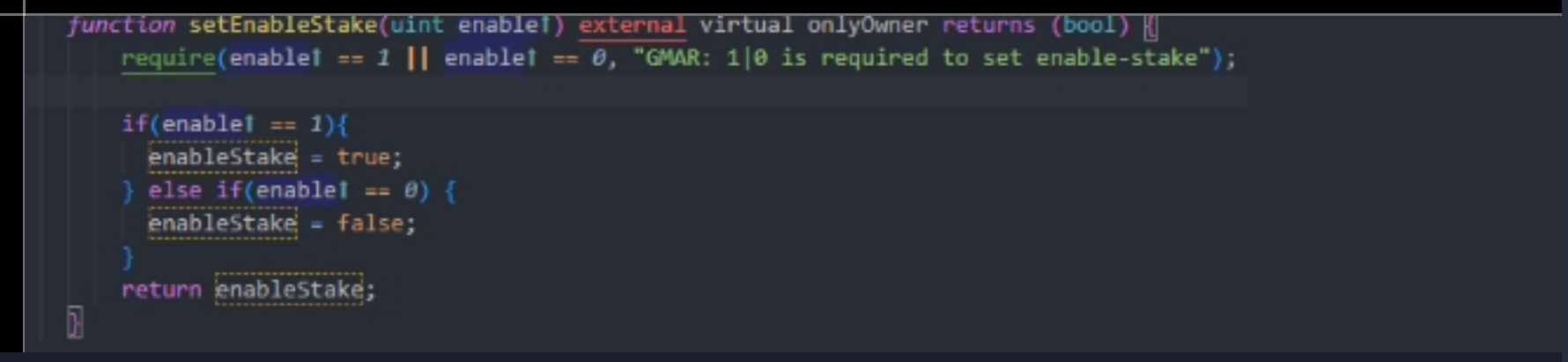
Remediation

You can rename the function to be getLengthOfStakes() or getUserStakesLength()

Status

Resolved

A.7: No need of else if in setEnableStake()

Line	Function - setEnableStake()
936	

```
function setEnableStake(uint enable1) external virtual onlyOwner returns (bool) {
    require(enable1 == 1 || enable1 == 0, "GMAR: 1|0 is required to set enable-stake");

    if(enable1 == 1){
      enableStake = true;
    } else if(enable1 == 0) {
      enableStake = false;
    }
    return enableStake;
}
```

Description

In function setEnableStake() the user/owner of the contract tries to set value for enabling or disabling stake. There is no need to use else if

Remediation

You can just use
if(enable == 1) {
enableStake = true;
} else {
enableStake = false;
}
Or you can just directly set the true or false value to enableStake variable.

Status

Resolved

Functional Testing

- [PASS] testBurn() (gas: 52126)
- [PASS] testChangeAdmin() (gas: 43355)
- [PASS] testDecreaseAllowance() (gas: 42077)
- [FAIL. Counterexample: calldata=0x678876d800000000000000000000000000000000000000000000000000000000, args=[0]] testFailFuzzTransfer(uint256) (runs: 5, μ: 13052, ~: 13052)
- [PASS] testFailSetMinStake() (gas: 12626)
- [PASS] testGetBoardOwner() (gas: 9855)
- [PASS] testIncreaseAllowance() (gas: 42036)
- [PASS] testSetCsrParty() (gas: 17973)
- [PASS] testSetMinStake() (gas: 18615)
- [PASS] testStake() (gas: 303914)
- [PASS] testTransfer() (gas: 41905)
- [PASS] testTransferFrom() (gas: 53292)
- [FAIL] Reason: Revert] testUnStake() (gas: 356339)
- [FAIL] Reason: Revert] testUnStakeLengthAndIdCheck() (gas: 759720)
- [PASS] testViewFunction() (gas: 30394)

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Gamr token Smart Contract. We performed our audit according to the procedure described above.

Some issues of High,Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.In the end, Gamr Team resolved all Issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Gamr Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Gamr Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

600+
Audits Completed

$15B
Secured

600K
Lines of Code Audited

Follow Our Journey

Audit Report
November, 2022

For

gamr

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com