audita

BottoDAO

Security Assessment

March 20, 2023

Prepared for Botto - A Decentralized Autonomous Artist

Presented By audita.io

Document

The contents of this document may include confidential information pertaining to the IT systems, intellectual property, and possible vulnerabilities along with methods of exploitation that the Client may possess. The report that contains this confidential information can be utilized internally by the Client, and can be made available to the public after all vulnerabilities are addressed, depending on the decision of the Client.

Approved by: Reni Dimitrova @ Audita.io
Contracts: Token, Governance, Airdrop, Liquidity Mining, Rewards Distribution, Active Reward, Retroactive Reward V2, Access Passes
Network: Ethereum
Programming language: Solidity
Method: Manual Audit by Solidity experts
Client Website: https://www.botto.com/
Client Documentation: https://docs.botto.com/
Timeline: 07/03/2023 - 14/03/2023

Table of Contents

Executive Summary

Documentation

Botto documentation is consistent with the code base. Adding NatSpec in the code will be a significant improvement.

`Code Quality`

The code is well-written, functional and logical.

Most contracts are forks from known protocols. As such, we managed to outline some minor efficiency improvements, with nothing too alarming present.

Manual Audit

During the manual audit conducted by our experts, we did not identify any `Critical` or `High` severity vulnerabilities.

We identified only 2 `Medium` and 5 `Low` code vulnerabilities.

Additionally, we have indicated 21 `Informational` issues, relating to:

- Documentation,
- Code Quality,
- Gas Optimization,
- General improvements.

Overall Assessment

Severity	Count	Acknowledged	Addressed
Medium	2	TBD	TBD
Low	5	TBD	TBD
Informational	16	TBD	TBD
Code Quality	2	TBD	TBD
Gas Optimization	3	TBD	TBD

Audita Vulnerability Classifications

Audita follows the most recent standards for vulnerability severities, taking into consideration both the possible impact and the likelihood of an attack occurring due to a certain vulnerability.

Severity	Description
Critical	Critical vulnerability is one where the attack is more straightforward to execute and can lead to exposure of users' data, with catastrophic financial consequences for clients and users of the smart contracts.
High	The vulnerability is of high importance and impact, as it has the potential to reveal the majority of users' sensitive information and can lead to catastrophic financial consequences for clients and users of the smart contracts.
Medium	The issue at hand poses a potential risk to the sensitive information of a select group of individual users. If exploited, it has the potential to cause harm to the client's reputation and could result in significant financial consequences.
Low	The vulnerability is relatively minor and not likely to be exploited repeatedly, or is a risk that the client has indicated is not impactful or significant, given their unique business situation.
Informational	The issue may not pose an immediate threat to ongoing operation or utilization, but it's essential to consider implementing security and software engineering best practices, or employing backup measures as a safety net.

Scope

The security assessment was scoped to the following smart contracts:

Token

BOTTO.sol

Governance

BottoGovernance.sol
BottoGovernanceV2.sol

Airdrop

BottoAirdrop.sol

Liquidity Mining

BottoLiquidityMining.sol
BottoLiquidityMiningV2.sol

Reward Distribution

BottoRewardDistributor.sol
BottoActiveReward.sol
BottoRetroactiveRewardV2.sol

Access Control

ActivePasses.sol

Findings

Summary

Code	Description	Severity
[AIR-M-01]	selfdestruct() will not be available after EIP4758	Medium
[RD-M-02]	Use call() instead of transfer() when transferring ETH	Medium
[B-L-01]	Missing checks for address(0x0)	Low
[B-L-02]	No storage gap for upgradeable contracts might lead to storage slot collision	Low
[B-L-03]	abi.encodePacked() should not be used with dynamic types when passing the result to a hash function such as keccak256()	Low
[GOV-L-04]	Use Ownable2StepUpgradeable instead of OwnableUpgradeable contract	Low
[AIR-L-05]	A malicious user can sabotage another user to call claim() function	Low
[B-I-01]	Use latest Solidity version	Informational
[B-I-02]	It is possible that the totalClaimed state variable returns an incorrect calculation	Informational
[B-I-03]	Different pragma directives are used	Informational
[B-I-04]	Use stable pragma statement	Informational
[B-I-05]	Use named imports instead of importfile.sol	Informational
[B-I-06]	Update external dependency to latest version	Informational
[B-I-07]	Solidity compiler optimizations can be problematic - its is to save gas, but some problems could happen - we not sure here	Informational

[B-I-08]	Event is missing indexed fields	`Informational`
[B-I-09]	NatSpec incomplete	`Informational`
[B-I-10]	You can use named parameters in mapping types	`Informational`
[B-I-11]	Claim is best to not be possible after the airdrop period has passed	`Informational`
[B-I-12]	Use TransferHelper.safeTransfer to keep the same convention	`Informational`
[B-I-13]	Use TransferHelper.safeTransferFrom to keep the same convention	`Informational`
[B-I-14]	Too many dependencies in the package.json	`Informational`
[B-I-15]	Emit addresses and amounts for query services	`Informational`
[B-I-16]	Owner deposit function is a bit non-conventional - use `transferFrom`	`Informational`
[B-I-17]	[Gas Optimization] The nonReentrant modifier should occur before all other modifiers	`Informational`
[B-I-18]	[Gas Optimization] Require statement for staked amount only increases size and gas costs	`Informational`
[B-I-19]	[Gas Optimization] Cache array length outside of loop	`Informational`
[B-I-20]	[Code Efficiency] Keep contract size small by removing the recover function	`Informational`
[AIR-I-21]	[Code Efficiency] Remove getBalanceOf airdrop, as one can simply ask the token itself - useless	`Informational`

Detailed Findings

[AIR-M-01]	selfdestruct() will not be available after EIP4758	Medium

Details:

This EIP will rename the SELFDESTRUCT opcode and replace its functionality.
It will no longer destroy code or storage, hence the contract will still be available. In this case it will break the logic of the project, as it will not work as expected:

```
function end(address payable recipient_) public onlyOwner {
        require(block.timestamp > endsAfter, "Cannot end yet");
        _recover(address(botto), getBalance(), recipient_);
        selfdestruct(recipient_);
}
```

Recommendation:

According to EIP4758, the SELFDESTRUCT opcode is renamed to SENDALL, and now only immediately moves all ETH in the account to the target. It no longer destroys code, storage or alters the nonce.

All refunds related to SELFDESTRUCT are removed.

[RD-M-02]	Use call() instead of transfer() when transferring ETH	Medium

Details:

The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs.

The use of the deprecated `transfer()` function when transfer ETH will inevitably make the transaction fail when:

The claimer smart contract does not implement a payable function. The claimer smart contract does implement a payable fallback which uses more than 2300 gas.

The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

The problem occurs in `BottoRewardDistributor.sol`

```
function claim(address payable claimant_) public virtual
nonReentrant {
        require(rewards[claimant_] > 0, "Nothing to reward");
         uint256 _amount = rewards[claimant_];
         rewards[claimant_] = 0;
         claimant_.transfer(_amount);
        emit Claim(claimant_, _amount);
    }
```

Recommendation:

Use `call()` instead of `transfer()` in `BottoRewardDistributor.sol`

[B-L-01]	Missing checks for address(0x0)	Low

Details:

Lack of zero-address validation on address parameters may lead to transaction reverts, waste gas, require resubmission of transactions and may even force contract redeployments in certain cases within the protocol:

`BottoAirdrop.sol`

```
50: address payable claimant_,
90: address token_,
```

```
92: address payable recipient_

102: function end(address payable recipient_) public onlyOwner {

BottoGovernance.sol

25: function initialize(address botto_) public initializer {

56: function recover(address token_, address payable recipient_)

BottoGovernanceV2.sol

213: function recover(address _token, address payable _recipient)

BottoLiquidityMining.sol

46: function initialize(address _bottoEth, address _botto) public

initializer {

239: function _applyReward(address account)

263: function rescueTokens(
```

Recommendation:

Consider adding explicit zero-address validation on input parameters of address type.

[B-L-02]	No storage gap for upgradeable contracts might lead to storage slot collision	**Low**

Details:

Without a storage gap, the variable in the contract might be overwritten by the upgraded contract if new variables are added. This could have unintended and serious consequences to the child contracts.

Recommendation:

For upgradeable contracts, it is good to have a storage gap to allow developers to freely add new state variables in the future without compromising the storage compatibility with current deployments. Otherwise, it may be very difficult to write new implementation code.

However, if Client decides to not change it, it would be acceptable, as they might have different plans and mitigation capabilities for following versions of the code.

[B-L-03]	abi.encodePacked() should not be used with dynamic types when passing the result to a hash function such as keccak256()	**Low**

Recommendation:

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123, 0x456)` → `0x123456` → `abi.encodePacked(0x1, 0x23456)`, but `abi.encode(0x123,0x456)` `0x0...1230...456)`

"Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

[GOV-L-04]	Use Ownable2StepUpgradeable instead of OwnableUpgradeable contract	**Low**

Details:

Using the alternative (`Ownable2StepUpgradeable.sol`), which has `transferOwnership` function , is more secure due to 2-stage ownership transfer.

This recommendation refers to the Governance contract (`BottoGovernance.sol`)

[AIR-L-05]	A malicious user can sabotage another user to call claim() function	Low

Details:

In `BottoAirdrop.sol` there is a `claim()` function.

```
function claim(
        bytes32[] memory proof_,
       address payable claimant_,
       uint256 claim_
    ) public nonReentrant {
       require(claimed[claimant_] != true, "Already claimed");
       require(verify(proof_, claimant_, claim_) == true, "Invalid
proof");
       claimed[claimant_] = true;
       if (IERC20(botto).transfer(claimant_, claim_) == true) {
            emit AirdropTransfer(claimant_, claim_);
}

totalClaimed = totalClaimed.add(claim_);

}
```

This function allows the claimant to get Airdrop tokens. Function `claim()` is public and allows anyone to call it.

After a user calls it, his address in the claimed map becomes true.

A malicious user can call the function with another user's address and a miniscule amount of tokens, thereby changing their map to `true`. This can prevent the other user from claiming their tokens, as they are stopped by the first check.

Recommendation:

To avoid this, it is recommended not to allow everyone to call this function.

Change the `claimant_` parameter anywhere in the function to `msg.sender`.

[B-I-01]	Use latest Solidity version	Informational

It is recommended that Solidity is upgraded to the latest available version.

[B-I-02]	It is possible that the totalClaimed state variable returns an incorrect calculation	Informational

Details:

This would occur only if there are any doubts related to the functionality of the $BOTTO token.
In our case, we are confident of its standard and capabilities, which means this issue hardly poses any risks.

[B-I-03]	Different pragma directives are used - inconsistency, make them the same	Informational

Details:

It is recommended that consistent pragma directives are used for every contract and the same Solidity version is implemented everywhere.

[B-I-04]	Use stable pragma statement	Informational

Details:

Using a floating pragma statement `>=0.6.0 <0.8.0` is discouraged as code can compile to different bytecodes with different compiler versions.

Recommendation:

Use a stable pragma statement to get a deterministic bytecode.

[B-I-05]	Use named imports instead of importfile.sol	Informational

Recommendation:

```
`import {contract1 , contract2 from "filename.sol";
```

[B-I-06]	Update external dependency to latest version	Informational

Recommendation:
The latest version is 4.8.2.

package.json:

```
13: "@openzeppelin/contracts": "^3.4.0",
14: "@openzeppelin/contracts-upgradeable": "^3.4.0",
```

[B-I-07]	Solidity compiler optimizations can be problematic	Informational

Details:

The compiler used saves gas, however it is known that it can be problematic in some cases

truffle-config.js

```
compilers: {
    solc: {
     version: "0.7.6",
      settings: {
       optimizer: {
          enabled: true,
          runs: 200,
},
```

Protocol has enabled optional compiler optimizations in Solidity.

Recommendation:

There have been several optimization bugs with security implications, meanwhile optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them.

[B-I-08]	Event is missing indexed fields	Informational

The following events are missing indexed fields:

BottoAirdrop.sol

```
24: event AirdropTransfer(address to, uint256 amount);
25: event RecoveryTransfer(address token, uint256 amount, address
recipient);
```

BottoGovernance.sol

```
19: event Staked(address indexed staker, uint256 amount);
20: event Unstaked(address indexed staker, uint256 amount);
21: event RecoveryTransfer(address token, uint256 amount, address
recipient);
```

BottoGovernanceV2.sol

```
26: event Deposit(uint256 totalRewards, uint256 startTime, uint256
endTime);
27: event Payout(address indexed staker, uint256 reward);
```

BottoLiquidityMining.sol

```
40: event Deposit(uint256 totalRewards, uint256 startTime, uint256
endTime);
41: event Stake(address indexed staker, uint256 bottoEthIn);
```

```
42: event Payout(address indexed staker, uint256 reward);

43: event Withdraw(address indexed staker, uint256 bottoEthOut);
```

BottoRewardDistributor.sol

```
21: event Claim(address indexed claimant, uint256 amount);

22: event Deposit(address depositor, uint256 amount);

23: event RecoveryTransfer(address token, uint256 amount, address
recipient);
```

Recommendation:

Index event fields make the field more quickly accessible to off-chain tools which parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

[B-I-09]	NatSpec incomplete - Documentation related, nice to have for auditors and for ppl wondering whether the project is legit	**Informational**

Details:
NatSpec is an important part of a project's documentation, especially for an outsider wondering whether the project has legitimacy.
It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). In more complex projects such as DeFi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

Recommendation:
Annotate all contracts with NatSpec and add return parameters in NatSpect comments.

Unfinished NatSpec documentation can be found in the following places:

```
BottoAirdrop.sol
70: function verify(
114: function _recover(

BottoGovernanceV2.sol
36: function deposit(
103: function stake(uint256 _botto) public virtual override update
nonReentrant {
121: function _stakeRewards(uint256 _bottoIn, address _account)
private {
162: function payout()
186: function _applyReward(address _account)

BottoLiquidityMining.sol
53: function deposit(
126: function stake(uint256 bottoEthIn) public virtual update
nonReentrant {
159: function withdraw()

187: function payout()
216: function _stake(uint256 bottoEthIn, address account) private {
239: function _applyReward(address account)
263: function rescueTokens(

BottoLiquidityMiningV2.sol
16: function updateEndTime(uint256 _endTime) public virtual update
onlyOwner nonReentrant {
```

[B-I-10]	Use named parameters in mapping types	Informational

Details:

From Solidity 0.8.18 named parameters in mapping types can be used. This will improve the readability of the code.

Recommendation:

In BottoGovernance.sol

From: `19: mapping(address => uint256) public userClaimedRewards;`

To:

```
19: mapping(address user => uint256 rewards) public

userClaimedRewards;
```

[B-I-11]	Claim is best to not be possible after the airdrop period has passed	**Informational**

Recommendation:
To improve user experience in general, it is recommended to disable the `claim` function after the airdrop period has passed. These events typically have a start and end date, which could make claim post-airdrop counterintuitive and confuse some newcomers.

[B-I-12]	Use TransferHelper.safeTransfer to keep the same convention	**Informational**
[B-I-13]	Use TransferHelper.safeTransferFrom to keep the same convention	**Informational**

Details:
In order to be consistent throughout the code, it is recommended to use the TransferHelper function everywhere.

[B-I-14]	Too many dependencies in the package.json	**Informational**

Recommendation:
In order to reduce the amount of dependencies in the package.json, simply copy-paste the TransferHelper library into your project.

[B-I-15]	Emit addresses and amounts for query services	Informational

Details:
It is recommended to emit addresses and amounts for - and not limited to - the following reasons:

- Query services like The Graph
- Reporting
- Replication of historical data and database
- Dates and record keeping

[B-I-16]	Owner deposit function is a bit non-conventional - use `transferFrom`	Informational

Details:
One is supposed to transfer the $BOTTO tokens to the contract, before the owner calls the deposit function for the funds to be there.
This approach works fine but is a bit non-conventional.

Recommendation:
It is recommended to use `transferFrom` in the function so the funds are really deposited on a deposit transaction. It would further decrease logic understanding.

Relating to:

```
function deposit(
        uint256 _totalRewards,
        uint256 _startTime,
        uint256 _endTime
    ) external virtual onlyOwner {
        require(
            startRewardsTime == 0,
            "Governance::deposit: already received deposit"
        );
```

Gas Usage

[B-I-17]	[Gas Optimization] The nonReentrant modifier should occur before all other modifiers	**Informational**

Details:

This is a best-practice to protect against re-entrancy in other modifiers. It can additionally reduce gas costs if this modifier occurs before all others.

This refers to the following:

```
BottoGovernanceV2.sol

103: function stake(uint256 _botto) public virtual override update

nonReentrant {

143: function unstake() public virtual override update nonReentrant

{

162: function payout()

213: function recover(address _token, address payable _recipient)


BottoLiquidityMining.sol

126: function stake(uint256 bottoEthIn) public virtual update

nonReentrant {

159: function withdraw()

187: function payout()

263: function rescueTokens(
```

[B-I-18]	[Gas Optimization] Require statement for staked amount only increases size and gas costs	Informational

Details:

```
require(amount_ > 0, "Invalid amount");
        userStakes[msg.sender] = userStakes[msg.sender].add(amount_);
```

Having the `require` statement for the staked amount of tokens has an unfavorable impact on gas costs.
Important here is to consider whether there will be a concern for people mistakenly sending funds to the contract. If they send a 0 amount, they will pay gas costs for a meaningless transaction, so it is unlikely that this will occur.

Recommendation:

Generally, if they send money by mistake, this will not be detrimental to the logic or business operation.
To mitigate this, the `require` statement can be changed with `if`.
Alternatively, this function can be removed altogether.

[B-I-19]	[Gas Optimization] Cache array length outside of loop	Informational

Details:

```
238:         for (uint256 i = 0; i < tokenIds.length; i++) {
```

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation, with 100 additional extra gas for each iteration except for the first). If it is a memory array, this is an extra mload operation, with 3 additional gas for each iteration except for the first).

Code Efficiency

[B-I-20]	[Code Efficiency] Keep contracts size small by removing the recover function	**Informational**

Details:

```
function recover(
        address token_,
        uint256 amount_,
        address payable recipient_
    ) public onlyOwner {
        require(amount_ > 0, "Invalid amount");
        require(address(botto) != token_, "Recover BOTTO on end");
        _recover(token_, amount_, recipient_);
    }
```

As a good practice the contracts should be as small as possible and avoid unnecessary logic.

The recover function can be considered as such logic, as if anyone transfers tokens directly to the contract, the contract will not be broken in any way. They will be locked forever, but this should not be a very high concern. Recover logic increases contract size and complexity.

[AIR-I-21]	[Code Efficiency] Remove getBalanceOf airdrop, as one can simply ask the token itself	**Informational**

Details:

```
function getBalance() public view returns (uint256) {
        return IERC20(botto).balanceOf(address(this));
    }
```

To increase code efficiency, one can simply get the balance of the airdrop by requesting it from the $BOTTO token itself. The `getBalance()` function only increases contract size.

Overall Assessment

Severity	Count	Acknowledged	Addressed
Critical	0	-	TDB
High	0	-	TDB
Medium	2	TDB	TDB
Low	5	TDB	TDB
Informational	16	TDB	TDB
Code Quality	2	TDB	TDB
Gas Optimization	3	TDB	TDB

BottoDAO's team has been quick to evaluate findings.

The second version of this report will reflect acknowledgements and any fixes addressed.

Recommendations

Audita has put forward the following recommendations for Botto contracts:

- Remove `selfdestruct()` function, as it can be detrimental to the code in the long-run.
- Use `call()` instead of `transfer()` in `BottoRewardDistributor.sol`.
- Consider adding explicit zero-address validation on input parameters of address type.
- For upgradeable contracts, it is good to have a storage gap to allow developers to freely add new state variables in the future without compromising the storage compatibility with current deployments.
- Use `abi.encode()` which will pad items to 32 bytes, which will prevent hash collisions.
- Use alternative (`Ownable2StepUpgradeable.sol`) due to its 2-stage ownership transfer in `BottoGovernance.sol`.
- In `BottoAirdrop.sol`, don't allow anyone to call the `claim()` function. Change the `claimant_` parameter anywhere in the function to `msg.sender`.
- Upgrade Solidity to its latest available version.
- Use consistent pragma directives for every contract and implement the same Solidity version everywhere.
- Use stable pragma statement to get a deterministic bytecode.
- Use named imports instead of importfile.sol. `` `import {contract1 , contract2 from "filename.sol";` ``
- Update external dependency to latest version - 4.8.2.
- Disable Solidity compiler optimizations.
- Index event fields. Make the field more quickly accessible to off-chain tools which parse events.

- Annotate all contracts with NatSpec and add return parameters in NatSpect comments.
- Use named parameters in mapping types.
- Disable the `claim` function after the airdrop period has passed.
- For consistency purposes, use the TransferHelper function everywhere in the code.
- In order to reduce the amount of dependencies in the package.json, simply copy-paste the TransferHelper library into your project.
- Emit addresses and amounts for query services.
- Owner deposit function is a bit non-conventional - use `transferFrom`.
- The nonReentrant modifier should occur before all other modifiers
- `Require` statement for staked amount only increases size and gas costs - change it with `if`, or remove it altogether.
- Cache array length outside of loop.
- Keep contracts size small and complexity low, by removing the recover function
- To increase code efficiency, one can simply get the balance of the airdrop by requesting it from the $BOTTO token itself. Remove `getBalance()` function only increases contract size.

Disclaimer

This audit makes no statements or warranties on the security of the code. This report should not be considered a sufficient assessment on the safety of the code, quality status, or any other contract statements. While we have conducted the analysis to our best abilities and produced this report in line with latest industry developments, it is important to not rely on this report only. In order for contracts to be considered as safe as possible, the industry standard requires them to be checked by several independent auditing bodies. Those can be other audit firms or public bounty programs.

The contacts live on a blockchain (a smart contract platform) - Smart contract platforms, their programming languages, and other software components are not immune to vulnerabilities that can be exploited by hackers. As a result, although a smart contract audit can help identify potential security issues, it cannot provide an absolute guarantee of the audited smart contract's security.