SMART CONTRACT AUDIT REPORT

for

DegoTokenV2

Prepared By: Patrick Lou

PeckShield
March 13, 2022

Document Properties

Client	DEGO Finance
Title	Smart Contract Audit Report
Target	DegoTokenV2
Version	1.0-rc
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Luo
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author	Description
1.0-rc	March 13, 2022	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the DegoTokenV2 contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to either security or performance. This document outlines our audit results.

1.1 About DegoTokenV2

DegoTokenV2 is an ERC20-compliant protocol token on the Ethereum blockchain, which is designed to be deflationary with a decreasing supply through burns. It is a governance token for the Dego platform which can be used to vote for mechanism changes, incentivize user participation, purchase new NFTs, and contribute to a dividend pool, etc. The basic information of the audited token contract is as follows:

Table 1.1: Basic Information Of DegoTokenV2

Item	Description
Issuer	DEGO Finance
Website	https://dego.finance/
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	March 13, 2022

In the following, we show the etherscan link to the contract address with the verified source code used in this audit:

- https://etherscan.io/address/0x3Da932456D082CBa208FEB0B096d49b202Bf89c8#code

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead of Transfer
	Costly Loop
	(Unsafe) Use of Untrusted Libraries
	(Unsafe) Use of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `DegoTokenV2` contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
Informational	1	■
Total	3	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found, and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, and 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key DegoTokenV2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Burn() Event Generation For Token Burn	Coding Practices	Confirmed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-003	Informational	Suggested Usage Of constant For _burnPool	Coding Practices	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 4 for details.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited DegoTokenV2. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	✓
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	✓
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	—

4 Detailed Results

4.1 Suggested Burn() Event Generation For Token Burn

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DegoTokenV2
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the DegoTokenV2 contract as an example. While examining the events that reflect the _totalSupply dynamics, we notice there is a lack of emitting an event to reflect the _totalSupply being reduced (line 1074) in the _transfer() routine where some portion of the transferred tokens are burned as fee. What is more, it comes to our attention that a Burn() event (line 934) is defined in the DegoTokenV2 contract, but it is never used anywhere. Given this, we suggest to emit the Burn() event in the _transfer() routine when the _totalSupply is reduced for the tokens burned as fee.

```
930   //events
931   event eveSetRate(uint256 burn_rate, uint256 reward_rate);
932   event eveRewardPool(address rewardPool);
933   event Mint(address indexed from, address indexed to, uint256 value);
934   event Burn(address indexed sender, uint256 indexed value);
935 }
```

Listing 4.1: DegoTokenV2

```
  function _transfer(address from, address to, uint256 value)
  internal override whenNotPaused
  {
      require(!blackAccountMap[from], "can't transfer");
      uint256 sendAmount = value;
      uint256 burnFee = (value.mul(_burnRate)).div(_rateBase);
      if (burnFee > 0) {
          //to burn
          super._transfer(from, _burnPool, burnFee);
          _totalSupply = _totalSupply.sub(burnFee);
          sendAmount = sendAmount.sub(burnFee);
          _totalBurnToken = _totalBurnToken.add(burnFee);
      }

      uint256 rewardFee = (value.mul(_rewardRate)).div(_rateBase);
      if (rewardFee > 0) {
         //to reward
          super._transfer(from, _rewardPool, rewardFee);
          sendAmount = sendAmount.sub(rewardFee);
          _totalRewardToken = _totalRewardToken.add(rewardFee);
      }
      super._transfer(from, to, sendAmount);
  }
}
```

Listing 4.2: `DegoTokenV2::_transfer()`

Recommendation Properly emit the `Burn()` event to timely reflect the state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been confirmed.

4.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `DegoTokenV2`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `DegoTokenV2` contract, there are privileged accounts (including `owner` and `_minters`) that play critical roles in governing and regulating the token-related operations. Our analysis shows that the `owner` and the only one member of `_minters` are currently configured as the same address: `0x72a7e0764a06697d8755048ccec37a37106e4798`, which is a proxy to a multi-sig `GnosisSafe` account.

To elaborate, we show below some sensitive operations that are related to `owner`. Specifically, it has the authority to set the black list, set the reward pool, and set various fee rates, etc.

```
    function addBlackAccount(address _blackAccount) external onlyOwner {
        require(!blackAccountMap[_blackAccount], "has in black list");
        blackAccountMap[_blackAccount] = true;
        emit AddBlackAccount(_blackAccount);
    }

    function delBlackAccount(address _blackAccount) external onlyOwner {
        require(blackAccountMap[_blackAccount], "not in black list");

        blackAccountMap[_blackAccount] = false;
        emit DelBlackAccount(_blackAccount);
    }

    /**
    * @dev for govern value
    */
    function setRate(uint256 burn_rate, uint256 reward_rate) external
        onlyOwner
    {
        require(_maxGovernValueRate >= burn_rate && burn_rate >= _minGovernValueRate,"
            invalid burn rate");
        require(_maxGovernValueRate >= reward_rate && reward_rate >= _minGovernValueRate
            ,"invalid reward rate");

        _burnRate = burn_rate;
        _rewardRate = reward_rate;

        emit eveSetRate(burn_rate, reward_rate);
    }

    /**
    * @dev for set reward
    */
    function setRewardPool(address rewardPool) external
        onlyOwner
    {
        require(rewardPool != address(0x0));

        _rewardPool = rewardPool;

        emit eveRewardPool(_rewardPool);
    }
```

Listing 4.3: DegoTokenV2

What is more, the `_minters` have the authority to mint new tokens.

```
    function mint(address account, uint256 amount) external
    {
        require(account != address(0), "ERC20: mint to the zero address");
```

```
        require(_minters[msg.sender], "!minter");

        uint256 curMintSupply = totalSupply().add(_totalBurnToken);
        uint256 newMintSupply = curMintSupply.add(amount);
        require(newMintSupply <= _maxSupply,"supply is max!");

        _mint(account, amount);
        emit Mint(address(0), account, amount);
    }
```

Listing 4.4: DegoTokenV2::mint()

It would be worrisome if the `owner` or the `_minters` are plain EOA accounts. The current multi-sig account greatly alleviates this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

Recommendation Promptly transfer the privileges of the `owner` and the `_minters` of `DegoTokenV2` to the intended governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated by setting a multi-sig account as the privileged account.

4.3 Suggested Usage Of constant For _burnPool

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DegoTokenV2`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

`DegoTokenV2` is an `ERC20`-compliant protocol token, which is designed to be deflationary with a decreasing supply through burns during token transfer. The tokens to be burned are transferred to the `_burnPool` which is set to a constant address: `0x6666666666666666666666666666666666666666`. It comes to our attention that the `_burnPool` variable is never changed. Based on this, it is strongly suggested to re-define `_burnPool` as constant for reduced gas cost.

```
    // burn pool!
    address public _burnPool = 0x6666666666666666666666666666666666666666;
```

Listing 4.5: DegoTokenV2

Recommendation Define the `_burnPool` variable as constant for reduced gas cost.

Status The issue has been confirmed.

5 Conclusion

In this security audit, we have examined the design and implementation of the DegoTokenV2 contract. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified three issues of varying severities. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.