SMART CONTRACT AUDIT REPORT

for

DotDot Protocol

Prepared By: Xiaomi Huang

PeckShield

May 16, 2022

Document Properties

Client	DotDot Finance
Title	Smart Contract Audit Report
Target	DotDot
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 16, 2022	Luck Hu	Final Release
1.0-rc	May 11, 2022	Luck Hu	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4
- 1.1 About DotDot . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Potential Reentrancy Risk in DddLpStaker . . . 11
- 3.2 Accommodation of Non-ERC20-Compliant Tokens . . . 13
- 3.3 Inaccurate EmergencyBailoutInitiated Event Generation . . . 14
- 3.4 Trust Issue of Admin Keys . . . 16
- 3.5 Incompatibility With Deflationary Tokens . . . 18

4 Conclusion 20

References 21

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the DotDot protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About DotDot

DotDot is a protocol to optimize yield, voting power, and liquidity provisioning on Ellipsis. Ellipsis is a decentralized exchange (AMM) where tokens may be swapped using liquidity provided by liquidity providers (LPs) who earn EPX emissions. Those who lock EPX receive vlEPX and earn a higher share of EPX rewards. Moreover, vote locked EPX or vlEPX allow to reward long-term users of a protocol. Those who hold vlEPX earn trading rewards from the protocol as well as voting power to direct EPX emissions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DotDot

Item	Description
Name	DotDot Finance
Website	https://dotdot.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 16, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/dotdot-ellipsis/dotdot-contracts.git (1bd2cd2)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/dotdot-ellipsis/dotdot-contracts.git (d95705d)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems 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).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- 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 1.3: The Full Audit Checklist

Category	Checklist Items
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
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
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 checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis 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.

In particular, we perform the audit according to the following procedure:

- 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.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `DotDot` smart contracts. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Undetermined	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key DotDot Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risk in DddLp-Staker	Time and State	Confirmed
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-003	Low	Inaccurate EmergencyBailoutInitiated Event Generation	Coding Practices	Fixed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Undetermined	Incompatibility With Deflationary Tokens	Business Logic	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 3 for details.

3 | Detailed Results

3.1 Potential Reentrancy Risk in DddLpStaker

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Taking the `DddLpStaker` contract as an example, the `claim()` function (see the code snippet below) is provided for stakers to claim rewards from the contract by externally calling the `rewardToken` contract to transfer rewards to the given `receiver`.

However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (line 308) starts before effecting the update on internal states (lines 312 and 314), hence violating the principle.

```
function claim(address receiver) external {
    _updateReward(msg.sender, receiver, true);
}
```

Listing 3.1: DddLpStaker::claim()

```
301 function _updateReward(address account, address receiver, bool claim) internal {
302     rewardPerTokenStored = rewardPerToken();
303     lastUpdateTime = lastTimeRewardApplicable();
304 
305     uint256 pending = claimable(account);
306     if (pending > 0) {
307         if (claim) {
308             rewardToken.transfer(receiver, pending);
309             emit FeeClaimed(account, receiver, pending);
310             pending = 0;
311         }
312         rewards[account] = pending;
313     }
314     userRewardPerTokenPaid[account] = rewardPerTokenStored;
315 }
```

Listing 3.2: DddLpStaker::_updateReward()

```
function claimable(address account) public view returns (uint256) {
  uint256 delta = rewardPerToken() - userRewardPerTokenPaid[account];
  return userBalances[account].total * delta / 1e18 + rewards[account];
}
```

Listing 3.3: DddLpStaker::claimable()

Specifically, in the case when `rewardToken` is an ERC777 token, a bad actor could hijack a `claim()` call before `rewardToken.transfer()` in line 308 with a callback function. Within the callback function, they could call the `claim()` function to claim rewards from the contract again. Since the `userRewardPerTokenPaid[account]` and the `rewards[account]` are not updated yet, the `claimable()` routine in line 305 could return an unexpected amount of pending rewards. If `rewardToken.transfer()` fails to revert when there's not enough token balance to transfer, the bad actor could exploit the reentrancy bug again and again to drain all the reward tokens in the contract.

Similar violations can be found in the `DddLpStaker::deposit()/withdraw()` and `DddIncentiveDistributor::depositIncentive()`, etc.

Recommendation Apply the `checks-effects-interactions` design pattern or add the reentrancy guard modifier.

Status This issue has been confirmed by the team. And the team clarifies that
1. The reward token contract is known one which is not ERC777, nor has malicious logic.
2. If a malicious incentive token is involved, there is no economic incentive from this issue and it could not harm other aspects of the protocol. So we do not feel it warrants mitigation.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `withdraw()` routine in the `EllipsisProxy` contract. If the `ZRX` token is supported as `_token`, the unsafe version of `IERC20(_token).transfer(_receiver, _amount)` (line 205) may proceed without a revert for transfer failure. Because it returns `false` for failure in the `ZRX` token contract's `transfer()/transferFrom()` implementation.

```
201    function withdraw(address _receiver, address _token, uint256 _amount) external
           returns (uint256) {
202        require(msg.sender == lpDepositor);
203        require(emergencyBailout[msg.sender][_token] == address(0), "Emergency bailout")
               ;
204        uint256 reward = lpStaker.withdraw(_token, _amount, true);
205        IERC20(_token).transfer(_receiver, _amount);
206        return reward;
207    }
```

Listing 3.5: EllipsisProxy :: withdraw()

Similar violations can be found in `EllipsisProxy::getReward()` and `DddLpStaker::deposit()/withdraw()/_updateReward()`, etc.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()/transferFrom()`.

Status This issue has been confirmed by the team. The team clarifies that only `Ellipsis LP` tokens will be used in the above mentioned functions, which all use standard ERC20 return values.

3.3 Inaccurate EmergencyBailoutInitiated Event Generation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EllipsisProxy`
- Category: Coding Practices [6]
- 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 `EllipsisProxy::emergencyWithdraw()` routine as an example. This routine is designed to trigger an emergency withdrawal for the given `_token`, deploy a bailout contract and transfer the `_token` balance to the bailout contract. While examining the event that reflect this emergency withdrawal operation, we notice the emitted `EmergencyBailoutInitiated` event (line 284) contains incorrect information. Specifically, the event (as shown in below code snippet) is defined with a number of parameters: the first parameter `token` encodes the address of the token to be emergency withdrawn; and the second parameter `lpDepositor` indicates the address of the `LpDepositor` contract which provides interfaces for depositors to deposit/withdraw token; while the last indicates the deployed bailout contract where the token balance is transferred to. The emitted event contains an incorrect order for the second and third parameters.

```
event EmergencyBailoutInitiated(
    address token,
    address lpDepositor,
    address bailout
);
```

Listing 3.6: EllipsisProxy.sol

```
262     function emergencyWithdraw(address _token) external {
263         require(msg.sender == emergencyAdmin);
264         require(emergencyBailout[lpDepositor][_token] == address(0), "Already initiated"
            );
265 
266         bytes20 targetBytes = bytes20(bailoutImplementation);
267         address bailout;
268         assembly {
269             let clone := mload(0x40)
270             mstore(clone, 0
                x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
271             mstore(add(clone, 0x14), targetBytes)
272             mstore(add(clone, 0x28), 0
                x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
273             bailout := create(0, clone, 0x37)
274         }
275         emergencyBailout[lpDepositor][_token] = bailout;
276 
277         IEmergencyBailout(bailout).initialize(_token, lpDepositor);
278         lpStaker.emergencyWithdraw(_token);
279 
280         uint256 amount = IERC20(_token).balanceOf(address(this));
281         require(amount > 0, "Bailout on empty pool");
282 
283         IERC20(_token).safeTransfer(bailout, amount);
284         emit EmergencyBailoutInitiated(_token, bailout, lpDepositor);
```

```
    }
```

Listing 3.7: EllipsisProxy :: emergencyWithdraw()

Recommendation Properly emit the `EmergencyBailoutInitiated` event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status This issue has been fixed in the commit: 9d69dad.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `DotDot` protocol, there are certain privileged accounts (`owner/minters, etc.`) that play a critical role in governing and regulating the system-wide operations. Specially, the `owner` is privileged to set protocol parameters and the `minters` are privileged to mint `DDD` token, etc. Our analysis shows that the privileged accounts needs to be scrutinized. In the following, we examine the privileged accounts and their related privileged accesses in current contracts.

```
    function setMinters(address[] calldata _minters) external onlyOwner {
        for (uint256 i = 0; i < _minters.length; i++) {
            minters[_minters[i]] = true;
        }

        emit MintersSet(_minters);
        renounceOwnership();
    }

    function mint(address _to, uint256 _value) external returns (bool) {
        require(minters[msg.sender], "Not a minter");
        balanceOf[_to] += _value;
        totalSupply += _value;
        emit Transfer(address(0), _to, _value);
        return true;
    }
```

Listing 3.8: `DddToken.sol`

```
function setAddresses(
    ITokenLocker _dddLocker,
```

```
    address _fixedVoteLpToken,
    IEllipsisProxy _proxy
) external onlyOwner {
    dddLocker = _dddLocker;
    fixedVoteLpToken = _fixedVoteLpToken;
    proxy = _proxy;

    renounceOwnership();
}
```

Listing 3.9: `DotDotVoting::setAddresses()`

```
    function emergencyWithdraw(address _token) external {
        require(msg.sender == emergencyAdmin);
        require(emergencyBailout[lpDepositor][_token] == address(0), "Already initiated"
            );

        bytes20 targetBytes = bytes20(bailoutImplementation);
        address bailout;
        assembly {
            let clone := mload(0x40)
            mstore(clone, 0
                x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
            mstore(add(clone, 0x14), targetBytes)
            mstore(add(clone, 0x28), 0
                x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
            bailout := create(0, clone, 0x37)
        }
        emergencyBailout[lpDepositor][_token] = bailout;

        IEmergencyBailout(bailout).initialize(_token, lpDepositor);
        lpStaker.emergencyWithdraw(_token);

        uint256 amount = IERC20(_token).balanceOf(address(this));
        require(amount > 0, "Bailout on empty pool");

        IERC20(_token).safeTransfer(bailout, amount);
        emit EmergencyBailoutInitiated(_token, bailout, lpDepositor);
    }
```

Listing 3.10: `EllipsisProxy::emergencyWithdraw()`

Note that if the privileged accounts are plain EOA accounts, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still 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 as mitigation.

Recommendation Promptly transfer the privileged accounts to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, 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 confirmed by the team. And the team has confirmed that most privileged functions are followed by a `renounceOwnership()`, which are one-time-use stuff called during deployment. Others will be well handled via a multi-sig account.

3.5 Incompatibility With Deflationary Tokens

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `DotDot` protocol, the `LpDepositor` contract is designed to take users' assets (`LP` tokens) and deliver rewards (`EPX/DDD` tokens) depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
function deposit(address _user, address _token, uint256 _amount) external {
  IERC20(_token).safeTransferFrom(msg.sender, address(proxy), _amount);

  uint256 balance = userBalances[_user][_token];
  uint256 total = totalBalances[_token];

  uint256 reward = proxy.deposit(_token, _amount);
  _updateIntegrals(_user, _token, balance, total, reward);

  userBalances[_user][_token] = balance + _amount;
  totalBalances[_token] = total + _amount;

  address depositToken = depositTokens[_token];
  if (depositToken == address(0)) {
      depositToken = _deployDepositToken(_token);
      depositTokens[_token] = depositToken;
  }
  IDepositToken(depositToken).mint(_user, _amount);
```

```
    emit Deposit(msg.sender, _user, _token, _amount);
}
```

Listing 3.11: LpDepositor::deposit())

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as deposit(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in safeTransfer() or safeTransferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the safeTransfer() or safeTransferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into DotDot for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Note the same issue also exists in DddLpStaker::deposit().

Recommendation Check the balance before and after the safeTransfer() or safeTransferFrom() call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed. And the team clarifies that these functions only handle Ellipsis LP tokens which are not deflationary.

4 | Conclusion

In this audit, we have analyzed the DotDot protocol design and implementation. DotDot is a protocol to optimize yield, voting power, and liquidity provisioning for Ellipsis on BNB Smart Chain. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.