SMART CONTRACT AUDIT REPORT

for

AstridDAO

Prepared By: Xiaomi Huang

PeckShield

May 22, 2022

Document Properties

Client	AstridDAO
Title	Smart Contract Audit Report
Target	AstridDAO
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 22, 2022	Xiaotao Wu	Final Release
1.0-rc1	May 18, 2022	Xiaotao Wu	Release Candidate #1

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 AstridDAO 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 Improved Validation in BAIToken/ATIDToken::permit() 11
3.2 Accommodation of Non-ERC20-Compliant Tokens 12
3.3 Improved Validation On Protocol Parameters 14
3.4 Trust Issue of Admin Keys 15
3.5 Improved Vault Close Logic in VaultManager 17
3.6 Potential Reentrancy Risks In AstridDAO 19
3.7 Incompatibility with Deflationary/Rebasing Tokens 20
3.8 Improved Sanity Checks Of System/Function Parameters 22
3.9 Inconsistent Implementation In ATIDStaking::_insertLockedStake() 23
3.10 Consistent Event Generation of CollateralAddressChanged 24

4 Conclusion 26

References 27

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the AstridDAO 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 can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About AstridDAO

The AstridDAO is a decentralized money market and multi-collateral stablecoin protocol built on Astar and for the Polkadot ecosystem, which allows users to borrow BAI, a stablecoin hard-pegged to USD, against risk assets at 0% interest and minimum collateral ratio. AstridDAO allows users to use the value in their risk assets (including ASTR, BTC, ETH, and DOT) without having to sell them. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The AstridDAO

Item	Description
Issuer	AstridDAO
Website	https://astriddao.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 22, 2022

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

- https://github.com/AstridDao/contracts.git (d72839d)

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

- https://github.com/AstridDAO/contracts/tree/auditing/contracts (c0a5bc3)

1.2 About PeckShield

PeckShield Inc. [12] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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.

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 or analysis does not identify any issue, the

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
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

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) [10], 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.

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 Logics	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 `AstridDAO` protocol implementation. 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	5	■■■■■
Informational	3	■■■
Total	10	

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 that 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 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 3 informational suggestions.

Table 2.1: Key AstridDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation in BAIToken/ATIDToken::permit()	Coding Practices	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-003	Low	Improved Validation On Protocol Parameters	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Improved Vault Close Logic in VaultManager	Business Logic	Resolved
PVE-006	Low	Potential Reentrancy Risks In AstridDAO	Time and State	Resolved
PVE-007	Medium	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Resolved
PVE-008	Informational	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Resolved
PVE-009	Informational	Inconsistent Implementation In ATIDStaking::_insertLockedStake()	Coding Practices	Resolved
PVE-010	Informational	Consistent Event Generation of CollateralAddressChanged	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Validation in BAIToken/ATIDToken::permit()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BAIToken, ATIDToken
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The AstridDAO protocol has two tokens BAIToken and ATIDToken, each supporting the EIP2612 functionality. In particular, the permit() function is introduced to simplify the token transfer process.

To elaborate, we show below this helper routine from the BAIToken contract. This routine ensures that the given owner is indeed the one who signs the approve request. Note that the internal implementation makes use of the ecrecover() precompile for validation. It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., owner, is not equal to address(0). This issue is also applicable to the ATIDToken token contract.

```
    function permit
    (
        address owner,
        address spender,
        uint amount,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
        override
    {
        require(deadline >= block.timestamp, "BAI: expired deadline");
        bytes32 digest = keccak256(abi.encodePacked('\x19\x01',
                         domainSeparator(), keccak256(abi.encode(
```

```
                        _PERMIT_TYPEHASH, owner, spender, amount,
                        _nonces[owner]++, deadline))));
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress == owner, "BAI: invalid signature");
        _approve(owner, spender, amount);
    }
```

Listing 3.1: `BAIToken::permit()`

Recommendation Strengthen the `permit()` routine to ensure the `owner` is not equal to `address(0)`.

Status The issue has been fixed by this commit: `c0a5bc3`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

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 3.2: 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 `_sendCOLGainToUser()` routine in the `ATIDStaking` contract. If the `USDT` token is supported as `IERC20(token)`, the unsafe version of `colToken.transfer(msg.sender, COLGain)` (line 351) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the require statement in line 352 expects a return value)!

```
349     function _sendCOLGainToUser(uint COLGain) internal {
350         emit COLSent(msg.sender, COLGain);
351         bool success = colToken.transfer(msg.sender, COLGain);
352         require(success, "ATIDStaking: Failed to send accumulated COLGain");
353     }
```

Listing 3.3: `ATIDStaking::_sendCOLGainToUser()`

Note a number of routines in the `AstridDAO` protocol can be similarly improved, including `ActivePool::sendCOL()/sendCOLToCollSurplusPool()/sendCOLToDefaultPool()/sendCOLToStabilityPool()`, `BorrowerOperations::_activePoolAddColl()/openVault()/addColl()/adjustVault()`, `CollSurplusPool::claimColl()`, `DefaultPool::sendCOLToActivePool()`, and `StabilityPool::withdrawCOLGainToVault()/_sendCOLGainToDepositor()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()/transferFrom()`.

Status This issue has been resolved as the team confirms that the `AstridDAO` protocol will not support `Non-ERC20-Compliant` tokens.

3.3 Improved Validation On Protocol Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AstridBase
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The AstridDAO protocol is no exception. Specifically, if we examine the AstridBase contract, it has defined a number of protocol-wide risk parameters, such as BAI_GAS_COMPENSATION, MIN_NET_DEBT, PERCENT_DIVISOR, and BORROWING_FEE_FLOOR, etc. In the following, we show the corresponding routines that allow for their changes.

```
    function setBAIGasCompensation(uint newBAIGasCompensation) public onlyOwner {
        BAI_GAS_COMPENSATION = newBAIGasCompensation;
    }
    function setMinNetDebt(uint newMinNetDebt) public onlyOwner {
        MIN_NET_DEBT  = newMinNetDebt;
    }
```

Listing 3.4: AstridBase::setBAIGasCompensation()/setMinNetDebt()

```
    function setPercentageDivisor(uint newPercentageDivisor) public onlyOwner {
        PERCENT_DIVISOR  = newPercentageDivisor;
    }
    function setBorrowingFeeFloor(uint newBorrowingFeeFloor) public onlyOwner {
        BORROWING_FEE_FLOOR = newBorrowingFeeFloor;
    }
```

Listing 3.5: AstridBase::setPercentageDivisor()/setBorrowingFeeFloor()

```
    function setAddresses(
        address _activePool,
        address _defaultPool,
        address _priceFeed
    ) public onlyOwner {
        activePool = IActivePool(_activePool);
        defaultPool = IDefaultPool(_defaultPool);
        priceFeed = IPriceFeed(_priceFeed);
    }

    function setParams(
        uint _MCR,
        uint _CCR,
        uint _BAIGasCompensation,
```

```
        uint _minNetDebt,
        uint _percentageDivisor,
        uint _borrowingFeeFloor,
        address _activePool,
        address _defaultPool,
        address _priceFeed
    ) public onlyOwner {
        MCR = _MCR;
        CCR = _CCR;
        BAI_GAS_COMPENSATION = _BAIGasCompensation;
        MIN_NET_DEBT = _minNetDebt;
        PERCENT_DIVISOR = _percentageDivisor;
        BORROWING_FEE_FLOOR = _borrowingFeeFloor;
        activePool = IActivePool(_activePool);
        defaultPool = IDefaultPool(_defaultPool);
        priceFeed = IPriceFeed(_priceFeed);
    }
```

Listing 3.6: `AstridBase::setAddresses()/setParams()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the `AstridDAO` users may suffer asset losses if the global parameter `CCR` is set to an extremely huge value by an unlikely mis-configuration.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `c0a5bc3`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `AstridDAO` protocol, there are some privileged account, i.e., `owners`. These privileged accounts play critical roles in governing and regulating the system-wide operations (e.g., configure protocol parameters, execute privileged operations, etc.). Our analysis shows that these privileged accounts needs to be scrutinized. In the following, we use the `AstridBase` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    // --- System parameters modification functions ---
    // All below calls require owner.
    function setMCR(uint newMCR) public onlyOwner {
        require(newMCR > _100pct, "MCR cannot < 100%");
        MCR = newMCR;
    }
    function setCCR(uint newCCR) public onlyOwner {
        require(newCCR > _100pct, "CCR cannot < 100%");
        CCR = newCCR;
    }
```

Listing 3.7: `AstridBase::setMCR()/setCCR()`

```
    function setBAIGasCompensation(uint newBAIGasCompensation) public onlyOwner {
        BAI_GAS_COMPENSATION = newBAIGasCompensation;
    }
    function setMinNetDebt(uint newMinNetDebt) public onlyOwner {
        MIN_NET_DEBT  = newMinNetDebt;
    }
```

Listing 3.8: `AstridBase::setBAIGasCompensation()/setMinNetDebt()`

```
    function setPercentageDivisor(uint newPercentageDivisor) public onlyOwner {
        PERCENT_DIVISOR  = newPercentageDivisor;
    }
    function setBorrowingFeeFloor(uint newBorrowingFeeFloor) public onlyOwner {
        BORROWING_FEE_FLOOR = newBorrowingFeeFloor;
    }
```

Listing 3.9: `AstridBase::setPercentageDivisor()/setBorrowingFeeFloor()`

```
    function setAddresses(
        address _activePool,
        address _defaultPool,
        address _priceFeed
    ) public onlyOwner {
        activePool = IActivePool(_activePool);
        defaultPool = IDefaultPool(_defaultPool);
        priceFeed = IPriceFeed(_priceFeed);
    }

    function setParams(
        uint _MCR,
        uint _CCR,
        uint _BAIGasCompensation,
        uint _minNetDebt,
        uint _percentageDivisor,
        uint _borrowingFeeFloor,
        address _activePool,
```

```
        address _defaultPool,
        address _priceFeed
    ) public onlyOwner {
        MCR = _MCR;
        CCR = _CCR;
        BAI_GAS_COMPENSATION = _BAIGasCompensation;
        MIN_NET_DEBT = _minNetDebt;
        PERCENT_DIVISOR = _percentageDivisor;
        BORROWING_FEE_FLOOR = _borrowingFeeFloor;
        activePool = IActivePool(_activePool);
        defaultPool = IDefaultPool(_defaultPool);
        priceFeed = IPriceFeed(_priceFeed);
    }
```

Listing 3.10: `AstridBase::setAddresses()/setParams()`

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that 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. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account 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. The `AstridDAO` team confirms that there will be a smoother transition from team ownership to `DAO` governance in the future.

3.5 Improved Vault Close Logic in VaultManager

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VaultManager`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

At the core of `AstridDAO` is the `VaultManager` contract which contains the logic to open, adjust and close various vaults. Note each vault is in essence an individual collateralized debt position for borrowing users. While reviewing the current vault-closing logic, we notice the current implementation can be improved.

To elaborate, we show below the related `_closeVault()` routine. The current logic properly releases unused states, including the vault `coll`, `debt`, as well as the associated `rewardSnapshots`. However, it does not release the vault index in the global owners, i.e., `VaultOwners`. The release of `arrayIndex` needs to be performed after the call `_removeVaultOwner()` is completed.

```
    function _closeVault(address _borrower, Status closedStatus) internal {
        assert(closedStatus != Status.nonExistent && closedStatus != Status.active);

        uint VaultOwnersArrayLength = VaultOwners.length;
        _requireMoreThanOneVaultInSystem(VaultOwnersArrayLength);

        Vaults[_borrower].status = closedStatus;
        Vaults[_borrower].coll = 0;
        Vaults[_borrower].debt = 0;

        rewardSnapshots[_borrower].COL = 0;
        rewardSnapshots[_borrower].BAIDebt = 0;

        _removeVaultOwner(_borrower, VaultOwnersArrayLength);
        sortedVaults.remove(_borrower);
    }
```

Listing 3.11: `VaultManager::_closeVault()`

Recommendation Release all unused states once a vault is closed. An example revision is shown below:

```
    function _closeVault(address _borrower, Status closedStatus) internal {
        assert(closedStatus != Status.nonExistent && closedStatus != Status.active);

        uint VaultOwnersArrayLength = VaultOwners.length;
        _requireMoreThanOneVaultInSystem(VaultOwnersArrayLength);

        Vaults[_borrower].status = closedStatus;
        Vaults[_borrower].coll = 0;
        Vaults[_borrower].debt = 0;

        rewardSnapshots[_borrower].COL = 0;
        rewardSnapshots[_borrower].BAIDebt = 0;

        _removeVaultOwner(_borrower, VaultOwnersArrayLength);
        sortedVaults.remove(_borrower);
        Vaults[_borrower].arrayIndex = 0;
    }
```

Listing 3.12: `VaultManager::_closeVault()`

Status The issue has been fixed by this commit: `c0a5bc3`.

3.6 Potential Reentrancy Risks In AstridDAO

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple contracts
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The BorrowerOperations contract of AstridDAO provides an external addColl() function for users to add collateral to a vault. While reviewing the current BorrowerOperations contract, we notice there is a potential reentrancy risk in current implementation.

To elaborate, we show below the code snippet of the addColl() routine in BorrowerOperations. The execution logic is rather straightforward: it firstly transfers the COLToken from the msg.sender to the BorrowerOperations contract, and then performs a collateral top-up for the msg.sender. If the COLToken faithfully implements the ERC777-like standard, then the addColl() routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when transfer() or transferFrom() actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering tokensToSend() and tokensReceived() hooks. Consequently, any transfer() or transferFrom() of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in COLToken.transferFrom() (line 215) before the actual transfer of the underlying assets occurs. So far, we also do not know how an attacker can exploit this issue to earn profit. After internal discussion, we consider it is necessary to bring this issue up to the team. Though the implementation of the addColl() function is well designed, we may intend to use the ReentrancyGuard::nonReentrant modifier to protect the addColl() function at the whole protocol level.

```
212     // Send collateral to a vault
213     function addColl(uint _amount, address _upperHint, address _lowerHint) external
            override {
214         // NOTE(Astrid): The vault owner should grant allowance to BorrowerOperations
                beforehand.
215         bool success = COLToken.transferFrom(msg.sender, address(this), _amount);
216         require(success, "BorrowerOperations: failed to add collateral to vault.");
```

```
        _adjustVault(msg.sender, /*_collChange=*/_amount, /*_isCollIncrease=*/(_amount >
             0), /*_BAIChange=*/0, /*_isDebtIncrease=*/false, _upperHint, _lowerHint, 0)
            ;
    }
```

Listing 3.13: BorrowerOperations::addColl()

Note a number of routines in the AstridDAO protocol can be similarly improved, including BorrowerOperations ::_activePoolAddColl()/openVault()/adjustVault(), and StabilityPool::withdrawCOLGainToVault()/_sendCOLGainToDepo ().

Recommendation Apply the non-reentrancy protection in the above-mentioned routine.

Status The issue has been fixed by this commit: c0a5bc3.

3.7 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.6, the BorrowerOperations contract provides an external addColl() function for users to add collateral to a vault. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the BorrowerOperations contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the contract's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the addColl() routine that is used to transfer COLToken to the BorrowerOperations contract.

```
    // Send collateral to a vault
    function addColl(uint _amount, address _upperHint, address _lowerHint) external
        override {
        // NOTE(Astrid): The vault owner should grant allowance to BorrowerOperations
            beforehand.
        bool success = COLToken.transferFrom(msg.sender, address(this), _amount);
        require(success, "BorrowerOperations: failed to add collateral to vault.");

        _adjustVault(msg.sender, /*_collChange=*/_amount, /*_isCollIncrease=*/(_amount >
             0), /*_BAIChange=*/0, /*_isDebtIncrease=*/false, _upperHint, _lowerHint, 0)
            ;
```

```
    }
```

Listing 3.14: `BorrowerOperations::addColl()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `swapExactAmountIn()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Velo FCX` for trading. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Note a number of routines in the `AstridDAO` protocol shares the same issue, including `ActivePool::sendCOL()/sendCOLToCollSurplusPool()/sendCOLToDefaultPool()/sendCOLToStabilityPool()`, `BorrowerOperations::_activePoolAddColl()/openVault()/adjustVault()`, `CollSurplusPool::claimColl()`, `DefaultPool::sendCOLToActivePool()`, and `StabilityPool::withdrawCOLGainToVault()/_sendCOLGainToDepositor()`.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been resolved as the team confirms that currently the `AstridDAO` protocol will not support `deflationary/rebasing` tokens.

3.8 Improved Sanity Checks Of System/Function Parameters

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LockupContract`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

In the `LockupContract` contract, the `withdrawATID()` function is used to withdraw a certain amount of `ATID` from the contract to the `beneficiary`. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the implementation of the `withdrawATID()` function. Specifically, the current implementation fails to check the given argument in `amount`. As a result, a user could `withdrawATID(0)` and transfer zero tokens, which is a waste of gas.

```
    // Withdraw a certain amount of ATID from this contract to the beneficiary.
    function withdrawATID(uint amount) external {
        require(canWithdraw(amount), "LockupContract: requested amount cannot be
            withdrawed");

        IATIDToken atidTokenCached = atidToken;
        // Also subject to initial locked time.
        require(atidTokenCached.transfer(beneficiary, amount), "LockupContract: cannot
            withdraw ATID");
        claimedAmount += amount;

        emit LockupContractWithdrawn(amount);
    }
```

Listing 3.15: `LockupContract::withdrawATID()`

Recommendation Validate the input arguments by ensuring `amount > 0` in the above `withdrawATID()` function.

Status The issue has been fixed by this commit: `c0a5bc3`.

3.9 Inconsistent Implementation In ATIDStaking::_insertLockedStake()

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ATIDStaking
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

In the AstridDAO protocol, the ATIDStaking contract allows users to deposit their ATID tokens and earn the borrowing and redemption fees in BAI and colToken. While reviewing the implementation of the _insertLockedStake() routine in this contract, we notice that there exists certain inconsistency that can be resolved.

To elaborate, we show below the code snippet of the _insertLockedStake function. It comes to our attention that the msg.sender is used as the key for updating the mapping state variable nextLockedStakeIDMap (lines 128 – 132). But for the updating of other mapping state variables, the _stakerAddress is used as the key instead. These mapping state variables store the staking state of the stakers.

```
126     function _insertLockedStake(address _stakerAddress, uint _ATIDamount, uint
            _stakeWeight, uint _lockedUntil) internal returns (uint newLockedStakeID) {
127         // Get (or init) next ID and increment.
128         if (nextLockedStakeIDMap[msg.sender] == 0) {
129             nextLockedStakeIDMap[msg.sender] = 1;
130         }
131         uint nextLockedStateID = nextLockedStakeIDMap[msg.sender];
132         nextLockedStakeIDMap[msg.sender]++;
133 
134         // Create and insert the new stakes into the map.
135         LockedStake memory newLockedStake = LockedStake({
136             active: true,
137 
138             ID: nextLockedStateID,
139             prevID: tailLockedStakeIDMap[_stakerAddress],  // Can be 0.
140             nextID: 0,  // New tail.
141 
142             amount: _ATIDamount,
143             lockedUntil: _lockedUntil,
144             stakeWeight: _stakeWeight
145         });
146         lockedStakeMap[_stakerAddress][newLockedStake.ID] = newLockedStake;
147 
148         ...
```

```
    }
```

Listing 3.16: ATIDStaking::_insertLockedStake()

Recommendation Use the _stakerAddress as the key for updating the mapping state variables which keep track of the staking states.

Status The issue has been fixed by this commit: c0a5bc3.

3.10 Consistent Event Generation of CollateralAddressChanged

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- 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 CollSurplusPool contract as an example. This contract has a privileged external function to configure the current contract addresses after their deployment. While examining the events that reflect their update, we notice there is a lack of emitting the related event to reflect the COLToken update. It comes to our attention that the same routine in BorrowerOperations has properly emitted the respective event COLTokenAddressChanged.

```
    function setAddresses(
        address _borrowerOperationsAddress,
        address _vaultManagerAddress,
        address _activePoolAddress,
        address _collateralTokenAddress
    )
        external
        override
        onlyOwner
    {
        checkContract(_borrowerOperationsAddress);
        checkContract(_vaultManagerAddress);
        checkContract(_activePoolAddress);
```

```
        borrowerOperationsAddress = _borrowerOperationsAddress;
        vaultManagerAddress = _vaultManagerAddress;
        activePoolAddress = _activePoolAddress;
        COLToken = IERC20(_collateralTokenAddress);

        emit BorrowerOperationsAddressChanged(_borrowerOperationsAddress);
        emit VaultManagerAddressChanged(_vaultManagerAddress);
        emit ActivePoolAddressChanged(_activePoolAddress);

        // _renounceOwnership();
    }
```

Listing 3.17: `CollSurplusPool::setAddresses()`

Recommendation Properly emit respective events when a new `collateralToken` becomes effective. This affects a number of contracts, including `ActivePool`, `CollSurplusPool`, and `DefaultPool`.

Status The issue has been fixed by this commit: `c0a5bc3`.

4 Conclusion

In this audit, we have analyzed the AstridDAO design and implementation. The AstridDAO is a decentralized money market and multi-collateral stablecoin protocol built on Astar and for the Polkadot ecosystem, which allows users to borrow BAI, a stablecoin hard-pegged to USD, against risk assets at 0% interest and minimum collateral ratio. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

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

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

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