



SMART CONTRACT AUDIT REPORT

for

Tender.fi Protocol



Prepared By: Xiaomi Huang

PeckShield
February 14, 2023

Document Properties

Client	Tender.fi
Title	Smart Contract Audit Report
Target	Tender.fi
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 14, 2023	Xuxian Jiang	Final Release
1.0-rc1	February 5, 2023	Xuxian Jiang	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 Tender.fi	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	Possibly Inaccurate Interest Accrual in CErc20	11
3.2	Potential Front-Running/MEV With Reduced Return	12
3.3	Non-VIP Withdraw Fee Bypass with JIT VIP Status	14
3.4	Trust Issue of Admin Keys	15
3.5	Non ERC20-Compliance Of CToken/CTokenGmx	17
3.6	Interface Inconsistency Between CErc20 And CEther	19
4	Conclusion	21
	References	22

1 | Introduction

Given the opportunity to review the **Tender.fi** design document and related smart contract source code, 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 Tender.fi

Tender.fi is a decentralized open-source protocol for borrowing and lending that is leading the way in innovation. It aims to provide support for autocompounding and collateralization for popular DeFi assets, starting with **GMX** and **GLP**. This is a unique and important aspect of the protocol, as it allows for the collateralization of long-tail assets. **Tender.fi**'s approach to borrowing and lending is what sets it apart from other DeFi protocols. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Tender.fi

Item	Description
Name	Tender.fi
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 14, 2023

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

- <https://github.com/tender-finance/compound-protocol.git> (1ccc01e)

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

- <https://github.com/tender-finance/compound-protocol.git> (d2155bf)

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 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

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

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. 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 `Tender.fi` protocol. 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	3	
Low	2	
Informational	1	
Total	7	

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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Tender.fi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possibly Inaccurate Interest Accrual in CErc20	Business Logic	Fixed
PVE-002	Low	Potential Sandwich/MEV Attacks For Reduced Returns	Time And State	Mitigated
PVE-003	Medium	Non-VIP Withdraw Fee Bypass with JIT VIP Status	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Non ERC20-Compliance Of CToken/C-TokenGmx	Coding Practices	Fixed
PVE-006	Informational	Interface Inconsistency Between CErc20 And CEther	Coding Practices	Fixed

Besides 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 Possibly Inaccurate Interest Accrual in CErc20

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CToken
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Tender.fi protocol provides incentive mechanisms with the support of autocompounding and collateralization for popular DeFi assets, starting with GMX and GLP. While examining the autocompounding feature, we notice the current implementation may result in inaccurate interest accrual.

To elaborate, we show below the `compoundFresh()` function. As the name indicates, this function is added to support GLP-related autocompounding. Note that the `accrualBlockNumber` storage variable records the latest block number for interest accrual. However, this function is exposed even for non-GLP tokens and may be abused to simply update `accrualBlockNumber` without actually collecting any interest!

```
149     function compoundFresh() internal {
150         if(totalSupply == 0) {
151             return;
152         }
153
154         /* Remember the initial block number */
155         uint currentBlockNumber = getBlockNumber();
156         uint accrualBlockNumberPrior = accrualBlockNumber;
157         uint _glpBlockDelta = sub_(currentBlockNumber, accrualBlockNumberPrior);
158
159         if(_glpBlockDelta < autoCompoundBlockThreshold){
160             return;
161         }
162
163         glpBlockDelta = _glpBlockDelta;
```

```

164     prevExchangeRate = exchangeRateStoredInternal();
165
166     // There is a new GLP Reward Router just for minting and burning GLP.
167     /// https://medium.com/@gmX.io/gmx-deployment-updates-nov-2022-16572314874d
168
169     IGmxRewardRouter newRewardRouter = IGmxRewardRouter(0
        xB95DB5B167D75e6d04227CfFFA61069348d271F5);
170
171     glpRewardRouter.handleRewards(true, false, true, true, true, true, false);
172     uint ethBalance = EIP20Interface(WETH).balanceOf(address(this));
173
174     // if this is a GLP cToken, claim the ETH and esGMX rewards and stake the esGMX
        Rewards
175
176     if(ethBalance > 0){
177         uint ethperformanceFee = div_(mul_(ethBalance, performanceFee), 10000);
178         uint ethToCompound = sub_(ethBalance, ethperformanceFee);
179         EIP20Interface(WETH).transfer(admin, ethperformanceFee);
180         newRewardRouter.mintAndStakeGlp(WETH, ethToCompound, 0, 0);
181     }
182
183     accrualBlockNumber = currentBlockNumber;
184 }

```

Listing 3.1: CToken::compoundFresh()

Recommendation Revise the above `compoundFresh()` function to ensure it performs noop for non-GLP tokens.

Status The issue has been fixed by this commit: [2ba64e2](#).

3.2 Potential Front-Running/MEV With Reduced Return

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CTokenGmx
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

As mentioned earlier, the `Tender.fi` protocol uses an enhanced yield model, where additional yield comes from the integration with `GMX`. Specifically, when borrowers provide `GLP` as collateral, `Tender.fi` can obtain additional yields (e.g., `esGMX`). Therefore, there is a constant need of swapping these yield tokens to another. However, our analysis shows the current conversion does not enforce meaningful slippage control.

```

74     function swapExactInputSingle(uint256 amountIn) internal returns (uint256 amountOut)
75     {
76         // Approve the router to spend WETH.
77         TransferHelper.safeApprove(WETH, address(swapRouter), amountIn);
78
79         // Naively set amountOutMinimum to 0. In production, use an oracle or other data
80         // source to choose a safer value for amountOutMinimum.
81         // We also set the sqrtPriceLimitx96 to be 0 to ensure we swap our exact input
82         // amount.
83         ISwapRouter.ExactInputSingleParams memory params =
84             ISwapRouter.ExactInputSingleParams({
85                 tokenIn: WETH,
86                 tokenOut: gmxToken,
87                 fee: poolFee,
88                 recipient: msg.sender,
89                 deadline: block.timestamp,
90                 amountIn: amountIn,
91                 amountOutMinimum: 0,
92                 sqrtPriceLimitX96: 0
93             });
94
95         // The call to 'exactInputSingle' executes the swap.
96         amountOut = swapRouter.exactInputSingle(params);
97     }

```

Listing 3.2: CTokenGmx::swapExactInputSingle()

To elaborate, we show above one example routine `swapExactInputSingle()`. We notice the conversion is routed to an external `UniswapV3Router` in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the TWAP or time-weighted average price of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been mitigated in the following commits: 65bce8d and 06b80d6.

3.3 Non-VIP Withdraw Fee Bypass with JIT VIP Status

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: Comptroller
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Tender.fi protocol provides incentive mechanisms that offer different fees based on the holding amount of TND balance. While examining the current logic in computing the associated fee, we notice the so-called non-VIP fee charge may be bypassed.

To elaborate, we show below the related code snippet to determine the membership-related fee factors. It comes to our attention the holding TND balance is queried from the `tndAddress` token, which allows a window opportunity to have a just-in-time (JIT) balance for VIP membership benefits. In particular, a protocol user may flashloan the required balance (`tokenBalanceVipThreshold`) right before the withdraw operation and return the same amount immediately after the operation. By doing so, the user can simply enjoy the membership benefits without actual cost!

```

938     function getIsAccountVip(address _account)
939     public
940     view
941     override
942     returns (bool)
943     {
944         if (vipNft != address(0)) {
945             if (IERC721(vipNft).balanceOf(_account) > 0) {
946                 return true;
947             }
948         }
949
950         if (whitelistedUser[_account]) {
951             return true;
952         }
953
954         if (compAddress != address(0) && tokenBalanceVipThreshold > 0) {
955             if (EIP20Interface(compAddress).balanceOf(_account) >=
956                 tokenBalanceVipThreshold && EIP20Interface(tndAddress).balanceOf(_account)
957                 >= tokenBalanceVipThreshold) {
958                 return true;
959             }
960         }
961
962         return false;

```

961

}

Listing 3.3: Comptroller::getIsAccountVip()

Recommendation Revisit the membership fee design to ensure it cannot be bypassed. Note other operations (e.g., redeem and borrow) share the same issue.

Status The issue has been fixed by this commit: 06b80d6.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the Tender.fi protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

1217     function _setGlpAddresses(IStakedGlp stakedGLP_, IGmxRewardRouter glpRewardRouter_,
1218         address glpManager_, address gmxToken_, address stakedGmxTracker_, address
1219         sbfGMX_) override public returns (uint) {
1220         // Check caller is admin
1221         if (msg.sender != admin) {
1222             revert SetStakedGlpAddressOwnerCheck();
1223         }
1224         stakedGLP = stakedGLP_;
1225         glpRewardRouter = glpRewardRouter_;
1226         glpManager = glpManager_;
1227         sbfGMX = sbfGMX_;
1228         gmxToken = gmxToken_;
1229         stakedGmxTracker = IRewardTracker(stakedGmxTracker_);
1230         return NO_ERROR;
1231     }
1232
1233     function _setAutocompoundRewards(bool autocompound_) override public returns (uint)
1234     {
1235         // Check caller is admin
1236         if (msg.sender != admin) {
1237             revert SetAutoCompoundOwnerCheck();
1238         }
1239     }

```

```
1235     }
1236     EIP20Interface(WETH).approve(glpManager, type(uint256).max);
1237     autocompound = autocompound_;
1238     return NO_ERROR;
1239 }
1240
1241 function _setAutoCompoundBlockThreshold(uint autoCompoundBlockThreshold_) override
1242     public returns (uint) {
1243     // Check caller is admin
1244     if (msg.sender != admin) {
1245         revert SetAutoCompoundOwnerCheck();
1246     }
1247     autoCompoundBlockThreshold = autoCompoundBlockThreshold_;
1248     return NO_ERROR;
1249 }
```

Listing 3.4: Example Setters in the CToken Contract

Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange 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 that current contracts have the support of being deployed behind a proxy. And 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 with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.5 Non ERC20-Compliance Of CToken/CTokenGmx

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CToken, CTokenGmx
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

Each asset supported by the `Tender.fi` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `CTokens`, users can earn interest through the `CToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `CTokens` as collateral. In the following, we examine the ERC20 compliance of these `CTokens`.

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	✓

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 part 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 3.2: 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 <i>address(0x0)</i> when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the CToken contract. Specifically, the current `transfer()` function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

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-adopted ERC20 spec-

ification. 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 <code>transfer()/transferFrom()</code> calls	—
Rebasing	The <code>balanceOf()</code> 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	✓

Recommendation Revise the `CToken` implementation to ensure its ERC20-compliance.

Status The issue has been fixed by this commit: `9bc9edf`.

3.6 Interface Inconsistency Between `CErc20` And `CEther`

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

Each asset supported by the `Tender.fi` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And `CTokens` are the primary means of interacting with the protocol when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `CTokens`: `CErc20` and

CEther. Both types expose the ERC20 interface and they wrap an underlying ERC20 asset and Ether, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `repayBorrow()` function as an example, the `CErc20` type returns an error code while the `CEther` type simply reverts upon any failure. The similar inconsistency is also present in other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

```

109  /**
110   * @notice Sender repays their own borrow
111   * @param repayAmount The amount to repay, or -1 for the full outstanding amount
112   * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
113   */
114   function repayBorrow(uint repayAmount) override external returns (uint) {
115       repayBorrowInternal(repayAmount);
116       return NO_ERROR;
117   }

```

Listing 3.5: `CErc20::repayBorrow()`

```

84  /**
85   * @notice Sender repays their own borrow
86   * @dev Reverts upon any failure
87   */
88   function repayBorrow() external payable {
89       repayBorrowInternal(msg.value);
90   }

```

Listing 3.6: `CEther::repayBorrow()`

It is also worth mentioning that the similar issue is also applicable to other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

Recommendation Ensure the consistency between these two types: `CErc20` and `CEther`.

Status The issue has been fixed by the following commits: 9747ed5 and 9bc9edf.

4 | Conclusion

In this audit, we have analyzed the `Tender.fi` design and implementation. The system presents a decentralized open-source protocol for borrowing and lending that is leading the way in innovation. It aims to provide support for autocompounding and collateralization for popular DeFi assets, starting with `GMX` and `GLP`. This is a unique and important aspect of the protocol, as it allows for the collateralization of long-tail assets. `Tender.fi`'s approach to borrowing and lending is what sets it apart from other DeFi protocols. 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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [3] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.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>.

