SMART CONTRACT AUDIT REPORT

for

Tetu Protocol

Prepared By: Yiqun Chen

PeckShield

September 19, 2021

Document Properties

Client	Tetu
Title	Smart Contract Audit Report
Target	Tetu
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 19, 2021	Xuxian Jiang	Final Release
1.0-rc	September 19, 2021	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Tetu 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 Tetu

Tetu is a decentralized yield aggregator committed to providing a next-generation yield aggregator to DeFi investors. Unlike other yield aggregators, users in Tetu don't have their rewards auto-compounded in the form of their original tokens. Instead, the yield rewards are converted and given directly in the form of xTETU. Furthermore, the protocol's liquidity balancer smart contract strives to maintain a stable price for the Tetu tokens across its liquidity pools.

The basic information of the Tetu protocol is as follows:

Table 1.1: Basic Information of The Tetu Protocol

Item	Description
Issuer	Tetu
Website	https://www.tetu.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 19, 2021

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

- https://github.com/tetu-io/tetu-contracts.git (7c29c46)

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

- https://github.com/tetu-io/tetu-contracts.git (14d6245)

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 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.

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) [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.

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 `Tetu` 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	4	■■■■
Informational	0	
Total	6	

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 and 4 low-severity vulnerabilities.

Table 2.1: Key Tetu Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Gas-Efficient Vault/Strategy Removal	Coding Practices	Fixed
PVE-002	Low	Deduplicate Checks in setDev-Funds() Logic	Business Logic	Fixed
PVE-003	Medium	Possible Costly Vault Share From Improper Initialization	Time and State	Confirmed
PVE-004	Low	Same Controller Enforcement Between Vault & Strategy	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Consistency in Book Keeping Actions	Coding Practices	Fixed

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 Gas-Efficient Vault/Strategy Removal

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Bookkeeper
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

The Tetu protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool. The design is also inspired from the YFI protocol with the separation and support of different vaults and strategies.

To allow for efficient management of supported vaults and strategies, the Tetu protocol has a Bookkeeper contract that maintains current _vaults and _strategies in storage state. Our analysis with their removal logic shows opportunity for improvement. To elaborate, we show below the related routines: removeFromVaults() and removeFromStrategies(). In the following, we use the removeFromVaults() as the example.

```
/// @notice Governance action. Remove given Vault from vaults array
/// @param index Index of vault in the vault array
function removeFromVaults(uint256 index) external onlyControllerOrGovernance {
  require(index < _vaults.length, "wrong index");
  emit RemoveVault(_vaults[index]);

  for (uint256 i = index; i < _vaults.length - 1; i++) {
    _vaults[i] = _vaults[i + 1];
  }
  _vaults.pop();
}

```

```
/// @notice Governance action. Remove given Strategy from strategies array
/// @param index Index of strategy in the strategies array
function removeFromStrategies(uint256 index) external onlyControllerOrGovernance {
  require(index < _strategies.length, "wrong index");
  emit RemoveStrategy(_strategies[index]);

  for (uint256 i = index; i < _strategies.length - 1; i++) {
    _strategies[i] = _strategies[i + 1];
  }
  _strategies.pop();
}
```

Listing 3.1: `Bookkeeper::removeFromVaults()/removeFromStrategies()`

The `removeFromVaults()` logic is rather straightforward in removing the given index from the maintained `_vaults` array. However, instead of repeatedly moving up the rest indexes, it is more gas-efficient to perform a `swap-and-pop` strategy, i.e., swapping the removed index with the last one and then popping the last one. The same strategy is also applicable to the `removeFromStrategies()` function.

Recommendation Improve the above two routines, i.e., `removeFromVaults()` and `removeFromStrategies()`, for reduced gas cost.

Status The issue has been fixed by this PR: `24`.

3.2 Deduplicate Checks in setDevFunds() Logic

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MintHelper`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

As part of the incentive mechanisms, the `Tetu` protocol has allocated the reward to the development team. While analyzing the setup of so-called `Dev Funds`, we notice the current logic can be improved.

To elaborate, we show below the related `setDevFunds()` function. As the name indicates, this function is designed to set up `Dev Funds`. Its business logic is rather straightforward in specifying the fractions for each given fund address and then saving the states in two arrays `devFunds` and `devFundsList`. An invariant is enforced to ensure the sum of given fractions is equal to `FUNDS_RATIO`.

```
/// @notice Set up new Dev Funds. Both arrays should have the same length
/// @param _funds Funds addresses
/// @param _fractions Funds fractions
```

```
function setDevFunds(address[] memory _funds, uint256[] memory _fractions) public
    onlyControllerOrGovernance {
  require(_funds.length != 0, "empty funds");
  require(_funds.length == _fractions.length, "wrong size");
  clearFunds();
  uint256 fractionSum;
  for (uint256 i = 0; i < _funds.length; i++) {
    require(_funds[i] != address(0), "Address should not be 0");
    require(_fractions[i] != 0, "Ratio should not be 0");
    fractionSum = fractionSum.add(_fractions[i]);
    devFunds[_funds[i]] = _fractions[i];
    devFundsList.push(_funds[i]);
  }
  require(fractionSum == FUNDS_RATIO, "wrong sum of fraction");
  emit FundsChanged(_funds, _fractions);
}
```

Listing 3.2: `MintHelper::setDevFunds()`

Our analysis shows the current implementation makes an implicit assumption that the given funds are distinct, which unfortunately is not enforced. With that, we suggest to detect and block the presence of duplicate fund addresses.

Recommendation Revise the `setDevFunds()` logic to ensure the given list of fund addresses does not contain any duplicate.

Status The issue has been fixed by this PR: `24`.

3.3 Possible Costly Vault Share From Improper Initialization

- ID: PVE-013
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SmartVault`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

The `Tetu` protocol allows users to deposit supported assets and get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `_deposit()` routine. This routine is used for participating users to deposit the supported assets and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
352 /// @notice Mint shares and transfer underlying from user to the vault
353 ///         New shares = (invested amount * total supply) / underlyingBalanceWithInvestment()
354 function _deposit(uint256 amount, address sender, address beneficiary) internal updateRewards(sender) {
355   require(amount > 0, "zero amount");
356   require(beneficiary != address(0), "zero beneficiary");
357
358   uint256 toMint = totalSupply() == 0
359   ? amount
360   : amount.mul(totalSupply()).div(underlyingBalanceWithInvestment());
361   _mint(beneficiary, toMint);
362
363   IERC20Upgradeable(underlying()).safeTransferFrom(sender, address(this), amount);
364
365   // only statistic, no funds affected
366   IBookkeeper(IController(controller()).bookkeeper())
367   .registerUserAction(beneficiary, toMint, true);
368
369   emit Deposit(beneficiary, amount);
370 }
```

Listing 3.3: SmartVault::_deposit()

Specifically, when the pool is being initialized (line 358), the share value directly takes the value of `amount` (line 359), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `toMint = amount = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been confirmed. And the team decides to mitigate this issue by properly following a guarded launch process.

3.4 Same Controller Enforcement Between Vault & Strategy

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmartVault`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

In the `Tetu` protocol, there is a one-to-one mapping between a `vault` and its `strategy`. To properly link a `vault` with its `strategy`, it is natural for the two to operate on the same underlying asset and share the same `controller`. For example, a smart vault allows for `USDC`-based deposits and withdraws. The associated `strategy` naturally has `USDC` as the underlying asset. If these two have different underlying assets, the link should not be successful. With that, the current abstract contract for `Synthetix`-based strategies can be improved to enforce the same underlying asset between the `vault` and the associated `strategy`.

If we examine the `setStrategy()` routine in the `SmartVault` contract, this routine allows for dynamic binding of the `vault` with a new `strategy` (line 588). A successful binding needs to satisfy a number of requirements. One specific example is shown as follows: `require(IStrategy(_strategy).underlying() == address(underlying()))`. Apparently, this requirement guarantees the consistency of the underlying asset between the `vault` and its associated `strategy`.

```
575 /// @notice Check the strategy time lock, withdraw all to the vault and change the strategy
576 ///         Should be called via controller
577 function setStrategy(address _strategy) external override onlyController {
578   require(_strategy != address(0), "zero strat");
579   require(IStrategy(_strategy).underlying() == address(underlying()), "wrong underlying");
580   require(IStrategy(_strategy).vault() == address(this), "wrong strat vault");
581
582   emit StrategyChanged(_strategy, strategy());
583   if (_strategy != strategy()) {
584     if (strategy() != address(0)) {// if the original strategy (no underscore) is defined
585       IERC20Upgradeable(underlying()).safeApprove(address(strategy()), 0);
586       IStrategy(strategy()).withdrawAllToVault();
587     }
588     _setStrategy(_strategy);
589     IERC20Upgradeable(underlying()).safeApprove(address(strategy()), 0);
590     IERC20Upgradeable(underlying()).safeApprove(address(strategy()), type(uint256).max);
591     IController(controller()).addStrategy(_strategy);
592   }
```

```
}
```

Listing 3.4: Controller :: setStrategy ()

In addition, we further recommend to add another requirement to ensure they share the same controller, i.e., `require(IStrategy(_strategy).vault()== address(this))`. By doing so, we can ensure that a new `strategy` linkage with an ill-provided argument with an unmatched controller will be timely detected and prevented to cause unintended consequences. With that, we suggest to maintain an invariant by ensuring the consistency of the same controller when a new `strategy` is being linked.

Recommendation Ensure the consistency of the common controller between the `vault` and its associated `strategy`. An example revision is shown below.

```
/// @notice Check the strategy time lock, withdraw all to the vault and change the
    strategy
///         Should be called via controller
function setStrategy(address _strategy) external override onlyController {
  require(_strategy != address(0), "zero strat");
  require(IStrategy(_strategy).underlying() == address(underlying()), "wrong
      underlying");
  require(IStrategy(_strategy).vault() == address(this), "wrong strat vault");
  require(IStrategy(_strategy).vault() == address(this), "wrong strat vault");

  emit StrategyChanged(_strategy, strategy());
  if (_strategy != strategy()) {
    if (strategy() != address(0)) {// if the original strategy (no underscore) is
        defined
      IERC20Upgradeable(underlying()).safeApprove(address(strategy()), 0);
      IStrategy(strategy()).withdrawAllToVault();
    }
    _setStrategy(_strategy);
    IERC20Upgradeable(underlying()).safeApprove(address(strategy()), 0);
    IERC20Upgradeable(underlying()).safeApprove(address(strategy()), type(uint256).max
        );
    IController(controller()).addStrategy(_strategy);
  }
}
```

Listing 3.5: Revised Controller :: setStrategy ()

Status The issue has been fixed by ensuring the linked vault and strategy have the same controller.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Controller`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Tetu` protocol, there is a special administrative contract, i.e., `Controller`. This `Controller` contract plays a critical role in governing and regulating the system-wide operations (e.g., `vault` /`strategy` addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among `vault`, `controller`, and `strategy`. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
/// @notice Only Governance can do it. Transfer token from FundKeeper to controller
/// @param _fund FundKeeper address
/// @param _token Token address
/// @param _amount Token amount
function fundKeeperTokenMove(address _fund, address _token, uint256 _amount) external
onlyGovernance timeLock(
  keccak256(abi.encode(IAnnouncer.TimeLockOpCodes.FundTokenMove, _fund, _token,
      _amount)),
  IAnnouncer.TimeLockOpCodes.FundTokenMove,
  false,
  address(0)
) {
  IFundKeeper(_fund).withdrawToController(_token, _amount);
  emit FundKeeperTokenMoved(_fund, _token, _amount);
}
```

Listing 3.6: `Controller::fundKeeperTokenMove()`

Note that it could be worrisome if the privileged `governance` account behind the `Controller` contract is a plain EOA account. A revised 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.

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 The issue has been confirmed by the team. The team clarifies that the admin key is `Gnosis Safe multi-sig (3/4)` contract with public well-known persons.

3.6 Consistency in Book Keeping Actions

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmartVault`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

The `Tetu` protocol provides the support to collect certain statistics about the protocol runtime dynamics. In particular, the `Bookkeeper` contract allows for the registration of user actions, e.g., `deposit` or `withdraw`. While examining the statistics collection, we notice the related bookkeeping actions can be improved for consistency.

In particular, we show below the registered user actions behind `deposit` and `withdraw`. The `withdraw` logic gracefully, with a `try-catch` enclosure (lines 325 – 327), handles the case when the bookkeeping call to the `Bookkeeper` contract may be failed. However, the `deposit` logic does not have the same treatment. For consistency, we suggest to employ the same `try-catch` to gracefully accommodate the unlikely, but possible `Bookkeeper` failures.

```
/// @notice Burn shares, withdraw underlying from strategy
///         and send back to the user the underlying asset
function _withdraw(uint256 numberOfShares) internal updateRewards(msg.sender) {
  require(totalSupply() > 0, "no shares");
  require(numberOfShares > 0, "zero amount");

  userLastWithdrawTs[msg.sender] = block.timestamp;

  uint256 totalSupply = totalSupply();
  _burn(msg.sender, numberOfShares);

  // only statistic, no funds affected
  try IBookkeeper(IController(controller()).bookkeeper())
  .registerUserAction(msg.sender, numberOfShares, false) {
  } catch {}
  ...
}

/// @notice Mint shares and transfer underlying from user to the vault
///         New shares = (invested amount * total supply) /
    underlyingBalanceWithInvestment()
```

```
function _deposit(uint256 amount, address sender, address beneficiary) internal updateRewards(sender) {
  require(amount > 0, "zero amount");
  require(beneficiary != address(0), "zero beneficiary");

  uint256 toMint = totalSupply() == 0
  ? amount
  : amount.mul(totalSupply()).div(underlyingBalanceWithInvestment());
  _mint(beneficiary, toMint);

  IERC20Upgradeable(underlying()).safeTransferFrom(sender, address(this), amount);

  // only statistic, no funds affected
  IBookkeeper(IController(controller()).bookkeeper())
  .registerUserAction(beneficiary, toMint, true);

  emit Deposit(beneficiary, amount);
}
```

Listing 3.7: CommonMaster::set()

Recommendation Gracefully handle the situation when the call to the `Bookkeeper` contract fails.

Status The issue has been fixed by this PR: `24`.

4 Conclusion

In this audit, we have analyzed the `Tetu` protocol design and implementation. `Tetu` is a decentralized yield aggregator that allows users to deposit into a decentralized liquidity platform and earn rewards in return. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 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.