



# SMART CONTRACT AUDIT REPORT

for

Tetu v2



Prepared By: Xiaomi Huang

PeckShield  
January 18, 2023

## Document Properties

Client	Tetu
Title	Smart Contract Audit Report
Target	Tetu v2
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	January 18, 2023	Xuxian Jiang	Final Release
1.0-rc	January 16, 2023	Luck Hu	Release Candidate

## Contact

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

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

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Tetu v2 . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Revisited Logic in merge() . . . . .	11
3.2	Revisited Max Redeem/Withdraw Amounts in TetuVaultV2 . . . . .	12
3.3	Incorrect missing Amount to Withdraw from Splitter . . . . .	13
3.4	Improved Initial remainingAmount in withdrawToVault() . . . . .	14
3.5	Trust Issue of Admin Keys . . . . .	16
3.6	Suggested Validation of _t in _balanceOfNFT() . . . . .	17
<b>4</b>	<b>Conclusion</b>	<b>19</b>
	<b>References</b>	<b>20</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Tetu v2 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 v2

Tetu is a decentralized yield aggregator committed to providing a next-generation yield aggregator to DeFi investors. Based on Tetu, the audited Tetu v2 introduces some new features, which enables protocol users to participate in governance with `veTETU`, improves the TetuVault with new deposit/withdraw fees, and introduces new `SplitterV2` with auto-rebalance logic adopted to multiple farming strategies, etc. The basic information of the Tetu v2 protocol is as follows:

Table 1.1: Basic Information of The Tetu v2 Protocol

Item	Description
Issuer	Tetu
Website	<a href="https://v2.tetu.io/">https://v2.tetu.io/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 18, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the audit scope only covers the following contracts: `ve/VeTetu.sol`, `vault/ERC4626Upgradeable.sol`, `vault/TetuVaultV2.sol`, `vault/StrategySplitterV2.sol`, `infrastructure/ControllerV2.sol`, `proxy/ControllableV3.sol`, `proxy/ProxyControlled.sol`, `proxy/UpgradeableProxy.sol`.

- <https://github.com/tetu-io/tetu-contracts-v2.git> (5ab0325)

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-v2.git> (fc05eb1)

## 1.2 About PeckShield

PeckShield Inc. [9] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

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

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

Table 1.3: The Full 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

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 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) [7], 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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 v2 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	4	
Low	2	
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 4 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Tetu v2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic in merge()	Business Logic	Fixed
PVE-002	Low	Revisited Max Redeem/Withdraw Amounts in TetuVaultV2	Business Logic	Fixed
PVE-003	Medium	Incorrect missing Amount to Withdraw from Splitter	Business Logic	Fixed
PVE-004	Medium	Improved Initial remainingAmount in withdrawToVault()	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Suggested Validation of _t in _balanceOfNFT()	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 Revisited Logic in merge()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: VeTetu
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

#### Description

In the Tetu v2 protocol, the VeTetu contract implements a Vote-Escrow NFT, which gives users the ability to vote on proposals. Specially, it provides the function for a user to merge his/her NFTs. While reviewing the logic to merge the NFTs, we notice the current end time of the target NFT may not be given correctly.

To elaborate, we show below the code snippet of the `merge()` routine. As the name indicates, it is used to merge two NFTs of the same user. By design, the new end time of the target NFT shall be the bigger one from the end times of both NFTs. However, in the first call to the `_depositFor()` routine, we notice the `lockedEnd` parameter which represents the current end time of the target NFT is set to `end` directly (line 1024). If the end time of the `from` NFT is bigger, the `end` is set to the end time of the `from` NFT, which is not the current end time of the target NFT. If the `lockedEnd` parameter is not given correctly, some state variables may become unexpected in the `_checkpoint()` routine, including the latest check point in the `_pointHistory[]` and the slope change of the old end time in the `slopeChanges[]`.

```

998   function merge(uint _from, uint _to) external nonReentrant {
999       require(attachments[_from] == 0 && voted[_from] == 0, ATTACHED);
1000       require(_from != _to, IDENTICAL_ADDRESS);
1001       require(_idToOwner[_from] == msg.sender && _idToOwner[_to] == msg.sender, NOT_OWNER)
        ;
1002
1003       uint lockedEndFrom = lockedEnd[_from];
1004       uint lockedEndTo = lockedEnd[_to];

```

```

1005     uint end = lockedEndFrom >= lockedEndTo ? lockedEndFrom : lockedEndTo;
1006     uint oldDerivedAmount = lockedDerivedAmount[_from];
1007
1008     uint length = tokens.length;
1009     for (uint i; i < length; i++) {
1010         address stakingToken = tokens[i];
1011         uint _lockedAmountFrom = lockedAmounts[_from][stakingToken];
1012         if (_lockedAmountFrom == 0) {
1013             continue;
1014         }
1015         lockedAmounts[_from][stakingToken] = 0;
1016
1017         _depositFor(DepositInfo({
1018             stakingToken : stakingToken,
1019             tokenId : _to,
1020             value : _lockedAmountFrom,
1021             unlockTime : end,
1022             lockedAmount : lockedAmounts[_to][stakingToken],
1023             lockedDerivedAmount : lockedDerivedAmount[_to],
1024             lockedEnd : end,
1025             depositType : DepositType.MERGE_TYPE
1026         }));
1027
1028         emit Merged(stakingToken, msg.sender, _from, _to);
1029     }
1030     ...
1031 }

```

Listing 3.1: VeTetu::merge()

**Recommendation** Revisit the logic in the above `merge()` routine and set the `lockedEnd` parameter to the current end time of the target NFT in the first call to the `_depositFor()` routine.

**Status** The issue has been fixed by this commit: `fc05eb1`.

## 3.2 Revisited Max Redeem/Withdraw Amounts in TetuVaultV2

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: TetuVaultV2
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

The `TetuVaultV2` contract is a customized `ERC4626` vault with some new features added. For example, it introduces the deposit/withdraw fees. While reviewing the logic to retrieve the maximum withdraw

amount for a user, we notice it doesn't properly take the withdraw fee into consideration.

To elaborate, we show below the code snippet of the `maxWithdraw()` routine. As defined in EIP-4626, the `maxWithdraw()` is expected to return the maximum amount of the underlying asset that can be withdrawn from the owner balance in the vault, through a withdraw call. However, we notice the `maxWithdraw()` routine doesn't take the withdraw fee into consideration, which shall be subtracted from the calculated asset amount per the owner balance. As a result, it returns an unexpected maximum withdraw amount that contains the withdraw fee, which shall go to the insurance contract.

```
321 function maxWithdraw(address owner) public view override returns (uint) {
322     return Math.min(maxWithdrawAssets, convertToAssets(balanceOf(owner)));
323 }
```

Listing 3.2: `maxWithdraw()`

**Recommendation** Revise the `maxWithdraw()` logic to subtract the withdraw fee from the maximum withdraw amount.

**Status** The issue has been fixed by this commit: [6d9ef6d](#).

### 3.3 Incorrect missing Amount to Withdraw from Splitter

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: TetuVaultV2
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

#### Description

In the Tetu v2 protocol, the TetuVaultV2 contract is a customized ERC4626 vault. Users deposits will be buffered in the vault, and after the defined buffer is filled, the remaining assets are invested to the splitter. User can withdraw directly from the vault if it has enough assets to cover the buffer and the withdrawal amount. Or the missing part will be withdrawn from the splitter to the vault. While examining the calculation of the missing amount, we notice the missing amount calculation needs to be improved.

To elaborate, we show below the `_processWithdrawFromSplitter()` routine. As the name indicates, it is used to calculate for the withdrawal amount from the splitter and move assets to the vault. Firstly, the routine uses current buffer amount to calculate the desired withdrawal amount from the splitter. By design, it shall use the new buffer amount which is the total asset amount subtracting the desired withdrawal amount of the user. What's more, it doesn't subtract the available asset

amount in the vault from the missing amount. As a result, more assets than expected are withdrawn from the splitter.

```

374 function _processWithdrawFromSplitter(
375     uint assetsNeed,
376     uint shares,
377     uint totalSupply_,
378     uint _buffer,
379     ISplitter _splitter,
380     uint assetsInVault
381 ) internal {
382     // withdraw everything from the splitter to accurately check the share value
383     if (shares == totalSupply_) {
384         _splitter.withdrawAllToVault();
385     } else {
386         uint assetsInSplitter = _splitter.totalAssets();
387         // we should always have buffer amount inside the vault
388         uint missing = (assetsInSplitter + assetsInVault)
389             * _buffer / BUFFER_DENOMINATOR
390             + assetsNeed;
391         missing = Math.min(missing, assetsInSplitter);
392         // if zero should be resolved on splitter side
393         _splitter.withdrawToVault(missing);
394     }
395 }

```

Listing 3.3: TetuVaultV2::\_processWithdrawFromSplitter()

**Recommendation** Revise current execution logic of `_processWithdrawFromSplitter()` to withdraw the exact desired amount of assets from the splitter.

**Status** The issue has been fixed by this commit: [6d9ef6d](#).

### 3.4 Improved Initial remainingAmount in withdrawToVault()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: StrategySplitterV2
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

#### Description

As described in Section 3.3, when the vault doesn't have enough assets to cover the withdrawal amount and the buffer, it will withdraw the missing part from the splitter. Similarly, if it doesn't have enough available assets in the splitter, it will withdraw the remaining assets from the strategies

one by one starting from the lower APR until the target amount is reached. While reviewing the withdraw logic in the `StrategySplitterV2` contract, we notice it may withdraw more assets than expected from the strategies.

To elaborate, we show below the code snippet of the `withdrawToVault()` routine, which is called from the vault to withdraw assets from the splitter. It takes the target amount from the input parameter `amount`. Specially, if current available balance in the splitter is smaller than the target amount (line 467), it withdraws the remaining amount from the strategies (lines 476–480). However, it comes to our attention that, the initial `remainingAmount` is set to the target amount without subtracting current balance of the splitter (line 466). As a result, it will withdraw more assets from the strategies than expected. Our analysis shows that, the initial `remainingAmount` shall be set to `amount - balance`.

```

461 function withdrawToVault(uint256 amount) external override {
462     _onlyVault();
463
464     address _asset = asset;
465     uint balance = IERC20(_asset).balanceOf(address(this));
466     uint remainingAmount = amount;
467     if (balance < amount) {
468         uint length = strategies.length;
469         for (uint i = length; i > 0; i--) {
470             IStrategyV2 strategy = IStrategyV2(strategies[i - 1]);
471
472             uint strategyBalance = strategy.totalAssets();
473             uint balanceBefore = strategyBalance + balance;
474
475             // withdraw from strategy
476             if (strategyBalance <= remainingAmount) {
477                 strategy.withdrawAllToSplitter();
478             } else {
479                 strategy.withdrawToSplitter(remainingAmount);
480             }
481             emit WithdrawFromStrategy(address(strategy));
482
483             uint currentBalance = IERC20(_asset).balanceOf(address(this));
484             // assume that we can not decrease splitter balance during withdraw process
485             uint withdrew = currentBalance - balance;
486             balance = currentBalance;
487
488             remainingAmount = withdrew <= remainingAmount ? remainingAmount - withdrew : 0;
489             ...
490         }
491     }
492     ...
493 }

```

Listing 3.4: `StrategySplitterV2 :: withdrawToVault()`

**Recommendation** Set the initial `remainingAmount` to `amount - balance` in case there are not enough assets available in the splitter.

**Status** The issue has been fixed by this commit: 6d9ef6d.

## 3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

### Description

In the Tetu v2 protocol, there is a privilege account, i.e., Governance that plays a critical role in governing and regulating the system-wide operations (e.g., add staking token). In the following, we use the `VeTetu` contract as an example and show the representative functions potentially affected by the privileges of the Governance account.

Specifically, the privileged functions in the `VeTetu` contract allow for the Governance to set the whitelist for `veTETU` token transfer, add new staking token, and set the weight of the staking token, etc.

```

233 function whitelistTransferFor(address value) external {
234     require(isGovernance(msg.sender), NOT_GOVERNANCE);
235     require(value != address(0), WRONG_INPUT);
236     uint timeLock = govActionTimeLock[TimeLockType.WHITELIST_TRANSFER];
237     require(timeLock != 0 && timeLock < block.timestamp, TIME_LOCK);
238
239     isWhitelistedTransfer[value] = true;
240     govActionTimeLock[TimeLockType.WHITELIST_TRANSFER] = 0;
241
242     emit TransferWhitelisted(value);
243 }
244
245 function addToken(address token, uint weight) external {
246     require(isGovernance(msg.sender), NOT_GOVERNANCE);
247     uint timeLock = govActionTimeLock[TimeLockType.ADD_TOKEN];
248     require(timeLock != 0 && timeLock < block.timestamp, TIME_LOCK);
249
250     _addToken(token, weight);
251     govActionTimeLock[TimeLockType.ADD_TOKEN] = 0;
252 }
253
254 function _addToken(address token, uint weight) internal {
255     require(token != address(0) && weight != 0, WRONG_INPUT);

```



```

256     _requireERC20(token);
257
258     uint length = tokens.length;
259     for (uint i; i < length; ++i) {
260         require(token != tokens[i], DUPLICATE);
261     }
262
263     tokens.push(token);
264     tokenWeights[token] = weight;
265     isValidToken[token] = true;
266
267     emit StakingTokenAdded(token, weight);
268 }

```

Listing 3.5: Example Privileged Operations in the `VeTetu` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the protocol users. It is worrisome if the privileged accounts are plain EOA accounts. 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.

**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 mitigated as the team clarifies that the Governance is Gnosis Safe multi-sig (3/5) contract with public well-known signers.

### 3.6 Suggested Validation of `_t` in `_balanceOfNFT()`

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `VeTetu`
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

#### Description

In the `Tetu v2` protocol, the `VeTetu` contract implements a `Vote-Escrow NFT` where it provides the interfaces to retrieve the voting power of a `NFT`. The voting power is calculated per the last check point of the `NFT`. While reviewing the logic of the `_balanceOfNFTat()` routine, we notice there is a lack of proper validation for the input timestamp.

To elaborate, we show below the code snippet of the `_balanceOfNFTAt()` routine. As the name indicates, it is used to retrieve the voting power of a NFT at the given timestamp `_t`. The voting power is calculated per the (bias, slope) of the last check point and the time elapsed since the last check point (line 1215). By design, the given timestamp shall be after the time of the last check point. In case the given timestamp is a history timestamp before the last check point, it shall revert the call. Based on this, it is suggested to add a validation for the given timestamp `_t`.

```

1209  function _balanceOfNFT(uint _tokenId, uint _t) internal view returns (uint) {
1210      uint _epoch = userPointEpoch[_tokenId];
1211      if (_epoch == 0) {
1212          return 0;
1213      } else {
1214          Point memory lastPoint = _userPointHistory[_tokenId][_epoch];
1215          lastPoint.bias -= lastPoint.slope * int128(int256(_t) - int256(lastPoint.ts));
1216          if (lastPoint.bias < 0) {
1217              lastPoint.bias = 0;
1218          }
1219          return uint(int256(lastPoint.bias));
1220      }
1221  }

```

Listing 3.6: VeTetu::\_balanceOfNFT()

**Recommendation** Add a proper validation for the given timestamp to ensure it is before the time of the last check point for the NFT.

**Status** The issue has been fixed by this commit: [6d9ef6d](#).

## 4 | Conclusion

In this audit, we have analyzed the Tetu v2 protocol design and implementation. Tetu is a decentralized yield aggregator committed to providing a next-generation yield aggregator to DeFi investors. Based on Tetu, the audited Tetu v2 introduces some new features, which enables protocol users to participate in governance with veTETU, improves the TetuVault with new deposit/withdraw fees, and introduces new SplitterV2 with auto-rebalance logic adopted to multiple farming strategies, etc. 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-628: Function Call with Incorrectly Specified Arguments. <https://cwe.mitre.org/data/definitions/628.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.