shieldify

Terplayer

BeraBTC Vault Token (BVT)

Staking and Distribution

SECURITY REVIEW

Date: 16 September 2025

CONTENTS

1. **About Shieldify Security** 3
2. **Disclaimer** 3
3. **About Terplayer – BeraBTC Vault Token (BVT) Staking and Distribution** 3
4. **Risk classification** 4
   - 4.1 Impact 4
   - 4.2 Likelihood 4
5. **Security Review Summary** 5
   - 5.1 Protocol Summary 5
   - 5.2 Scope 5
6. **Findings Summary** 5
7. **Findings** 6

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Terplayer - BeraBTC Vault Token (BVT) - Staking and Distribution

Decentralized Governance Engine

- Establish a tiered governance framework to enable precise decision-making and control
- Parameter Governance Layer: Adjusts key risk parameters such as collateralization ratios and liquidation penalties (quarterly voting).
- Strategic Decision Layer: Approves/rejects Batoshi Venture investment plans (proposal submission requires a bond deposit).
- Protocol Upgrade Layer: Smart contract iterations require dual-majority approval (>66% of both token holdings and Venture seats).
- Innovative Mechanism: Introduces a "Governance Execution Delay" feature—critical proposals are locked for 72 hours post-approval to prevent flash governance attacks.

Value Capture

Participate in the BVT growth flywheel through multiple channels, providing the ecosystem with greater room for expansion. As the ecosystem grows, participants benefit from both the increasing quantity and value of BVT. Revenue sources include protocol income, BTC appreciation in the treasury, investment returns, PoL (Proof-of-Liquidity) income, and BVT growth earnings.

Governance Rights Securitization

Governance power is derived through the veBVT model: - Voting Power = Staked Amount ×(min(Stake Duration, Max Lock Period) / Max Lock Period + 1) - Delegation Mechanism: Allows users to delegate their voting power to professional governance entities.

Design Logic: - Users select their stake duration (from 1 week to 1 year). - Voting power increases linearly with lock-up duration. - Time decay mechanism: Voting power linearly decreases over time. - At the maximum lock-up period (1 year), users receive full 1:1 voting power.

Economic Impact: By linking governance participation with token lock-up periods, long-term holders are rewarded with surplus governance power and benefits.

Protocol Risk Hedging Tools

BVT holders automatically receive:

- Priority access to liquidation profits.
- Insurance fund claims (in extreme events, BVT holders can apply for compensation proportional to their holdings).
- Exclusive subscription rights to future protocol products.

This structure elevates BVT beyond a traditional governance token, transforming it into a financial instrument that encapsulates protocol risk and rewards, pioneering the "Governance-as-a-Service" (GaaS) paradigm.

Investment Participation Securitization

By staking BVT, users gain decision-making rights and revenue-sharing privileges in Batoshi Venture's investment activities: - Personal Yield Factor = Individual Staked BVT × sqrt(Weighted Average Stake Duration) - Personal Yield Share = (Personal Yield Factor / Personal Yield Factors) × BVT Surplus Returns

Innovative Mechanisms: - Dynamic Staking Weighting: Continuous participation inherits 50% of previous stake duration weighting. - Anti-Collusion Design: Staked amounts from related addresses are aggregated to prevent manipulation of yield weighting. - Liquidity Discount: Early unstaking incurs a 25% yield penalty.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 5 days, with a total of 80 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is very well-written. The audit report contributed one Critical, one High, four Medium and three Low severity issues. They are mainly related to underflow in withdrawal calculations, several commission bypasses and other issues of lower severity.

The Terplayer team has done a great job with the development and has been highly responsive to the Shieldify research team's inquiries.

5.1 Protocol Summary

Project Name	**Terplayer – BVT Staking and Distribution**
Repository	berabtc-vault-token
Type of Project	Vault Token, Staking, Distribution
Audit Timeline	5 days
Review Commit Hash	c68f412b3c7dfd99d3f6302a42bdf772ededb2a3
Fixes Review Commit Hash	3f5794019f643000f2b6d39e757eddb8aac6ef97

5.2 Scope

The following smart contracts were in the scope of the security review:

File	**nSLOC**
src/BondDealer.sol	160
src/BvtRewardVault.sol	172
Total	**332**

6. Findings Summary

The following issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **2**
- **Medium** issues: **4**
- **Low** issues: **3**

ID	Title	Severity	Status
[C-01]	Withdrawal Calculation Causes Underflow, Locking All User Funds	Critical	Fixed
[H-01]	Users with Level 5 Account May Bypass the Applied 12% Commissions	High	Acknowledged
[M-01]	64-hop Traversal in `getHigherLevelParents()` Enables Commission Bypass Via Long Chains	Medium	Acknowledged
[M-02]	Missing Self-Flag in `delegatedStakeUsers` Causes Duplicate Entries and Withdrawal Failures	Medium	Fixed
[M-03]	Protocol Prioritises Lower Level Users, Hence Robbing the Higher Level Ones of Commissions	Medium	Fixed
[M-04]	The System Might Skip Searching for Level 5 Users Under Certain Conditions	Medium	Acknowledged
[L-01]	Misleading `setLevel()` Governance Check with `onlyOwner`	Low	Fixed
[L-02]	Absence of Storage Gaps Will Lead to Storage Collision in the Upgradable Contracts	Low	Fixed
[L-03]	Level 5 Users Will Be Obligated to Pay Commissions to Lower Level Users Under Certain Conditions	Low	Acknowledged

7. Findings

[C-01] Withdrawal Calculation Causes Underflow, Locking All User Funds

Severity

Critical Risk

Description

The withdrawal function includes the user in their own delegation list and uses ceiling division for all calculations. This causes `totalDelegatedAmount` to exceed the requested withdrawal amount, resulting in an underflow when calculating `remainingAmount = amount - totalDelegatedAmount`, which renders all withdrawals unsuccessful.

Location of Affected Code

File: src/BvtRewardVault.sol#L155

```
function withdraw(uint256 amount) external nonReentrant {
  // code

  // Calculate and withdraw from delegated stakes
  for (uint256 i = 0; i < users.length; i++) {
      address user = users[i];
      uint256 delegatedAmount = delegatedStakes[msg.sender][user];
      if (delegatedAmount > 0) {
          uint256 withdrawAmount = (delegatedAmount * amount + stakes[msg
              .sender] - 1)  / stakes[msg.sender];
          if (withdrawAmount > 0) {
              totalDelegatedAmount += withdrawAmount;
              _delegateWithdraw(msg.sender, user, withdrawAmount);
          }
      }
  }
  // Calculate remaining amount to withdraw from user's own stake
  uint256 remainingAmount = amount - totalDelegatedAmount;
  if (remainingAmount > 0) {
      _delegateWithdraw(msg.sender, msg.sender, remainingAmount);
  }
  // code
}
```

Impact

All withdrawals fail due to underflow, permanently locking user funds.

Recommendation

Exclude the user from their delegation list and handle their portion separately, and also, consider using floor division with remainder assignment to prevent over-calculation.

Team Response

Fixed.

[H-01] Users with Level 5 Account May Bypass the Applied 12% Commissions

Severity

High Risk

Description

When having a Level 5 account, a user is obligated to pay a 12% commission rate to the `recipientComission` address. Now, let's imagine the following scenario:

1. The user has such a Level 5 account, which is connected only to other Level 5 accounts or to accounts with a level equal to or lower than that of the child and makes a new account specifying the Level 5 account as his parent

2. The user deposits all of his assets from the newly created account, hence paying the 12% commission rate to his own Level 5 account, resulting in him practically paying only 2% commissions (if `fiveLevelRecipientComission` address is different from `address(0)`)
3. As a result of this, the Level 5 user robs the protocol of its commissions and generates 12% more assets than he should

Location of Affected Code

File: src/BvtRewardVault.sol#L96

```
function deposit(uint256 amount) external nonReentrant {
  // code
  for (uint256 i = 0; i < commissionParents.length; i++) {
      UserInfo memory parentInfo = commissionParents[i];
      if (parentInfo.user == address(0)) {
          continue;
      }
      uint256 commission = bondDealerContract.getCommission(
          uint256(parentInfo.level)
      );

      uint256 commissionAmount = (amount *
          (commission - totalHigherLevelCommissionAmountRate)) / 10000;
      if (commissionAmount > 0) {
          if (!isUserInDelegatedStakeList[msg.sender][parentInfo.user]) {
              isUserInDelegatedStakeList[msg.sender][
                  parentInfo.user
              ] = true;
              delegatedStakeUsers[msg.sender].push(parentInfo.user);
          }
          _delegateStake(msg.sender, parentInfo.user, commissionAmount);
          userAmount -= commissionAmount;
          totalHigherLevelCommissionAmountRate = commission;
      }
  }
```

```
    if (recipientComission != address(0)) {
        uint256 totalCommissionAmountRate = bondDealerContract
            .getTotalCommission();
        if (
            totalCommissionAmountRate >
               totalHigherLevelCommissionAmountRate
        ) {
            uint256 recipientCommissionAmount = (amount *
                (totalCommissionAmountRate -
                    totalHigherLevelCommissionAmountRate)) /
                MAX_COMMISSION_RATE;
            if (
                !isUserInDelegatedStakeList[msg.sender][recipientComission]
            ) {
                isUserInDelegatedStakeList[msg.sender][
                    recipientComission
                ] = true;
                delegatedStakeUsers[msg.sender].push(recipientComission);
            }
            _delegateStake(
                msg.sender,
                recipientComission,
                recipientCommissionAmount
            );
            userAmount -= recipientCommissionAmount;
        }
    }
    // code
}
```

Impact

User steals the 12% commissions from the protocol

Recommendation

Don't allow new accounts to directly attach to Level 5 accounts, as this is a possible outcome of the situation. Allow them to attach to accounts with 1-2 levels higher than their own.

Team Response

Acknowledged.

[M-01] 64-hop Traversal in `getHigherLevelParents()` Enables Commission Bypass Via Long Chains

Severity

Medium Risk

Description

The `getHigherLevelParents()` traverses up to 64 parent hops and only returns the nearest parent for each strictly higher level. An attacker can create a chain of 64 accounts with the same/lower levels, ensuring no strictly higher-level parent is found within the hop limit. When `BvtRewardVault.deposit()` uses this list, `totalHigherLevelCommissionAmountRate` remains 0 and the entire capped commission is routed to `recipientComission`, bypassing intended parent payouts.

Location of Affected Code

File: src/BondDealer.sol#L250

```
function _getHigherLevelParents(address user) internal view returns (
    UserInfo[] memory) {
  // code
  // Traverse up the parent chain
  for (uint256 hop = 0; hop < 64 && cur != address(0); hop++) {
      UserInfo memory parent = userInfos[cur];
      uint256 parentLevel = uint256(parent.level);

      // Only consider parents with higher level than current user
      if (parentLevel > currentLevel && !found[parentLevel]) {
          higherLevelParents[parentLevel] = parent;
          found[parentLevel] = true;
          count++;
          // If we found all possible higher levels, we can stop
          if (count == 6 - currentLevel - 1) break;
      }
      cur = parent.parent;
  }
  // code
}
```

Impact

- Systematic bypass of parent commission distribution.
- Breaks incentive design for higher-level parents since funds are diverted to `recipientComission` instead.
- No direct loss of funds, but materially intended revenue sharing.

Recommendation

Consider removing or increasing the hop limit.

Team Response

Acknowledged.

[M-02] Missing Self-Flag in `delegatedStakeUsers` Causes Duplicate Entries and Withdrawal Failures

Severity

Medium Risk

Description

When adding `msg.sender` to `delegatedStakeUsers[msg.sender]` on first deposit, the code doesn't set `isUserInDelegatedStakeList[msg.sender][msg.sender] = true`. If the same address appears again through other paths (as `recipientComission`, `fiveLevelRecipientComission`, or `parent`), it gets added as a duplicate entry, causing withdrawal calculations to process the same user multiple times and, almost certainly, revert due to underflow.

Location of Affected Code

File: src/BvtRewardVault.sol#L96

```
function deposit(uint256 amount) external nonReentrant {
  // code
  if (delegatedStakeUsers[msg.sender].length == 0) {
      delegatedStakeUsers[msg.sender].push(msg.sender);
  }
  // code
}
```

Impact

- Duplicate entries in `delegatedStakeUsers` array cause withdrawal calculations to process the same user multiple times.
- Can cause `remainingAmount = amount - totalDelegatedAmount` to underflow and revert, locking user funds.
- Affects users where commission recipients are also parents or in self-parenting scenarios.

Recommendation

Set the membership flag when adding self in `isUserInDelegatedStakeList[msg.sender]` to true.

Team Response

Fixed.

[M-03] Protocol Prioritises Lower Level Users, Hence Robbing the Higher Level Ones of Commissions

Severity

Medium Risk

Description

Let's investigate the flow of the following scenario: `L0 => L5 => L1`

The L0 user wants to directly connect with the specific L5 user and pay the 12% commission to him. The problem in this example will be that the L5 account parent is an L1 account, which, as of this moment, will be accounted for by the protocol and both the parents will receive as follows: \codex{L1 == 4%} \codex{L5 == 8%}

Which will not be what the `L0` user desires. On top of that, the `L0` user will be obligated to pay more gas because of the loop in the `BondDealer::_getHigherLevelParents()` function.

Location of Affected Code

File: src/BondDealer.sol#L250

```
function _getHigherLevelParents(address user) internal view returns (
    UserInfo[] memory) {
  // code
  // Traverse up the parent chain
  for (uint256 hop = 0; hop < 64 && cur != address(0); hop++) {
      UserInfo memory parent = userInfos[cur];
      uint256 parentLevel = uint256(parent.level);

      // Only consider parents with higher level than current user
      if (parentLevel > currentLevel && !found[parentLevel]) {
          higherLevelParents[parentLevel] = parent;
          found[parentLevel] = true;
          count++;
          // If we found all possible higher levels, we can stop
          //FIXME: This can stop when level 5 is found
          if (count == 6 - currentLevel - 1) break;
      }
      cur = parent.parent;
  }
  // code
}
```

Impact

Users are obligated to pay more gas and the Level 5 users are essentially robbed of their commission rates

Recommendation

Consider fixing the code like this:

```
for (uint256 hop = 0; hop < 64 && cur != address(0); hop++) {
    UserInfo memory parent = userInfos[cur];
    uint256 parentLevel = uint256(parent.level);

// Only consider parents with a higher level than the current user
    if (parentLevel > currentLevel && !found[parentLevel]) {
        higherLevelParents[parentLevel] = parent;
        found[parentLevel] = true;
        count++;
// If we found all possible higher levels, we can stop
//FIXME: This can stop when level 5 is found
        if (count == 6 - currentLevel - 1) break;
    }
+   currentLevel =uint256( parent.level);
+   if (currentLevel == 6) break;
    cur = parent.parent;
}
```

This way, the user will be able to start from a specific level without going down the ladder and paying commission to dealers he doesn't want to.

Team Response

Fixed.

[M-04] The System Might Skip Searching for Level 5 Users Under Certain Conditions

Severity

Medium Risk

Description

Let's consider the following scenario of a parent tree: `L0 => L2 =>L3 =>L4 => L5`

The L0 user chooses to start from the L2 dealer and everything is going smoothly until the `for` loop in the BondDealerV1::_getHigherLevelParents()' function. This scenario will go as follows: 1. `L0 =>L2` parenting connection will increase the `count` to 1 and the `currentLevel` variable to 2 2. `L2 => L3` parenting connection will increase the `count` to 2 and the `currentLevel` variable to 3 3. The problem arises when we go from Level 3 to Level 4. As the `count` variable will be increased to 3, the following equation will break from the loop and skip the search for the Level 5 user: `if (count == 6 - currentLevel || currentLevel == uint256(BondLevel.LEVEL_5)) break;`

As can be seen here, `currentLevel == 3` and `count == 3`, which will break from the for loop before the `L5` user is acknowledged.

Location of Affected Code

File: src/BondDealer.sol#L275

```
function _getHigherLevelParents(address user) internal view returns (
    UserInfo[] memory) {
     UserInfo memory currentUser = userInfos[user];
     require(currentUser.user != address(0), "BondDealer: User does not
        exist");
     uint256 currentLevel = uint256(currentUser.level);
     UserInfo[] memory higherLevelParents = new UserInfo[](6);
     bool[6] memory found; // Track which levels have been found
     uint256 count = 0;
     address cur = currentUser.parent;
     // Traverse up the parent chain
     for (uint256 hop = 0; hop < 64 && cur != address(0); hop++) {
         UserInfo memory parent = userInfos[cur];
         uint256 parentLevel = uint256(parent.level);
         // Only consider parents with higher level than current user
         if (!found[parentLevel]) {
             if (parentLevel > currentLevel || (hop == 0 && parentLevel >
                0)) {
                 higherLevelParents[parentLevel] = parent;
                 found[parentLevel] = true;
                 count++;
                 // If we found all possible higher levels, we can stop
@>               if (count == 6 - currentLevel || currentLevel == uint256(
    BondLevel.LEVEL_5)) break;
             }
         }
         if (parentLevel > currentLevel) currentLevel = parentLevel;
         cur = parent.parent;
     }
     // Create result array with only the found higher level parents
     UserInfo[] memory result = new UserInfo[](count);
     if (count > 0) {
         uint256 resultIndex = 0;
         for (uint256 i = 0; i < 6; i++) {
             if (found[i]) {
                 result[resultIndex] = higherLevelParents[i];
                 resultIndex++;
             }
         }
     }
     return result;
}
```

Impact

Level 5 users may not be iterated through under those conditions

Recommendation

Save the `currentLevel` from the original user and add 1 to it. This way, the `6- (currentLevel +1)` equation will give you all of the available spots to fill the `higherLevelParents` array. If at some point a level is bypassed, for example `L0=>L2`, remove 1 from this equation, as there will be 1 available spot less (because L1 is skipped and, respectively, 2 levels are skipped, remove 2 and so on).

This will look like this:

```
function _getHigherLevelParents(address user) internal view returns (
    UserInfo[] memory) {
        UserInfo memory currentUser = userInfos[user];
        require(currentUser.user != address(0), "BondDealer: User does
            not exist");
        uint256 currentLevel = uint256(currentUser.level);
+       int256  ad = currentLevel + 1;
        UserInfo[] memory higherLevelParents = new UserInfo[](6);
        bool[6] memory found; // Track which levels have been found
        uint256 count = 0;
        address cur = currentUser.parent;
// Traverse up the parent chain
        for (uint256 hop = 0; hop < 64 && cur != address(0); hop++) {
            UserInfo memory parent = userInfos[cur];
            uint256 parentLevel = uint256(parent.level);
// Only consider parents with higher level than current user
            if (!found[parentLevel]) {
                if (parentLevel > currentLevel || (hop == 0 &&
                    parentLevel > 0)) {
                    higherLevelParents[parentLevel] = parent;
                    found[parentLevel] = true;
                    count++;
// If we found all possible higher levels, we can stop
-                   if (count == 6 - currentLevel || currentLevel ==
    uint256(BondLevel.LEVEL_5)) break;
+                   if (count == 6 - ad || currentLevel == uint256(
    BondLevel.LEVEL_5)) break;
                }
            }
```

```
-             if (parentLevel > currentLevel) currentLevel = parentLevel;
+             if (parentLevel > currentLevel){
+              uint256 ad1 = parentLevel -  currentLevel;
+              if( ad1 > 1 ) ad += ad1 - 1;
+              currentLevel = parentLevel;
+             }
              cur = parent.parent;
         }
// Create result array with only the found higher-level parents
        UserInfo[] memory result = new UserInfo[](count);
        if (count > 0) {
            uint256 resultIndex = 0;
            for (uint256 i = 0; i < 6; i++) {
                if (found[i]) {
                    result[resultIndex] = higherLevelParents[i];
                    resultIndex++;
                }
            }
        }
        return result;
}
```

Team Response

Acknowledged.

[L-01] Misleading `setLevel()` Governance Check with `onlyOwner`

Severity

Low Risk

Description

The `setLevel()` uses `onlyOwner` and also enforces `msg.sender == levelOwner`. Given `initialize(address _levelOwner, address _initialOwner)` can set different addresses, the effective authority is the `levelOwner`. The `onlyOwner` modifier is misleading and may cause governance friction or accidental lockout expectations.

Location of Affected Code

File: src/BondDealer.sol#L63

```
function setLevel(address[] calldata users, uint256[] calldata levels)
    external onlyOwner {
  require(msg.sender == levelOwner,"BondDealer: Invalid level owner");
  // code
}
```

Impact

Confusing authority model since it implies owner control while enforcing levelOwner, and possibly governance friction.

Recommendation

Consider making `setLevel()` explicitly `levelOwner` -only and removing `onlyOwner`. Also, consider implementing a `setLevelOwner()` function.

Team Response

Fixed.

[L-02] Absence of Storage Gaps Will Lead to Storage Collision in the Upgradable Contracts

Severity

Low Risk

Description

When contracts upgrade their logic, a storage collision may occur if any other storage variables are added to support the new logic. That's where storage gaps and namespaced storage come into play. When put and used correctly, they protect the upgraded system of storage collision.

This is not the case with both `BvtRewardVault` and `BondDealer` contracts, which are upgradable but don't support either storage gaps or namespaced storage.

Location of Affected Code

File: src/BvtRewardVault.sol

File: src/BondDealer.sol

Impact

When the contracts are upgraded and new logic with new storage variables is added, a storage collision will occur, resulting in an unusable protocol.

Recommendation

Either add storage gaps or namespaced storage.

Team Response

Fixed.

[L-03] Level 5 Users Will Be Obligated to Pay Commissions to Lower Level Users Under Certain Conditions

Severity

Low Risk

Description

There is a slight chance of a Level 5 user having a Level 0 parent, which will lead to the Level 5 user paying commissions to lower-level users. Right now, the `BondDealer::_getHigherLevelParents` function performs the following check:

```
@> if (parentLevel > currentLevel || (hop == 0 && parentLevel > 0)) {
```

This will lead to skipping the break line and will actually compute a `higherLevelParents` array for the highest level possible, leading to the Level 5 user paying commissions to lower-level users instead of the protocol.

Location of Affected Code

File: src/BondDealer.sol#L270

```
function _getHigherLevelParents(address user) internal view returns (
    UserInfo[] memory) {
     UserInfo memory currentUser = userInfos[user];
     require(currentUser.user != address(0), "BondDealer: User does not
         exist");
     uint256 currentLevel = uint256(currentUser.level);
     UserInfo[] memory higherLevelParents = new UserInfo[](6);
     bool[6] memory found; // Track which levels have been found
     uint256 count = 0;
     address cur = currentUser.parent;
     // Traverse up the parent chain
     for (uint256 hop = 0; hop < 64 && cur != address(0); hop++) {
         UserInfo memory parent = userInfos[cur];
         uint256 parentLevel = uint256(parent.level);
         // Only consider parents with higher level than current user
```

```
        if (!found[parentLevel]) {
@>              if (parentLevel > currentLevel || (hop == 0 &&
    parentLevel > 0)) {
                higherLevelParents[parentLevel] = parent;
                found[parentLevel] = true;
                count++;
                // If we found all possible higher levels, we can stop
                if (count == 6 - currentLevel || currentLevel == uint256(
                    BondLevel.LEVEL_5)) break;
            }
        }
        if (parentLevel > currentLevel) currentLevel = parentLevel;
        cur = parent.parent;
    }
    // Create result array with only the found higher-level parents
    UserInfo[] memory result = new UserInfo[](count);
    if (count > 0) {
        uint256 resultIndex = 0;
        for (uint256 i = 0; i < 6; i++) {
            if (found[i]) {
                result[resultIndex] = higherLevelParents[i];
                resultIndex++;
            }
        }
    }
    return result;
}
```

Impact

Level 5 users may end up paying commissions to other users instead of paying them to the protocol

Recommendation

Don't check against `parentLevel` being 0 because it practically does nothing. Even though a Level 0 user finds his way into the result array, it won't do anything because no stake is going to be allocated to him. Another way of fixing this is to just return an empty array if `currentLevel == uint256(BondLevel.LEVEL_5)` before the for loop has even started.

Team Response

Acknowledged.

shieldify

Thank you!