Protocol Audit Report

Version 1.0

Bree

December 2, 2025

BriVault Protocol Audit Report

Bree

November 6th - 13th, 2025

Prepared by: Bree Lead Auditors: - Bree

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] Incorrect Minting of Shares to `msg.sender` Instead of `receiver` in `BriVault::deposit` function (ERC4626 Violation + Asset Loss Risk)
  - [H-2] Missing Duplicate Guard in joinEvent causing inflated Winner Shares and Misallocation of Funds
- Medium
  - [M-1] Missing Share Balance Check Allows Users to Stay Registered Without Stake Burn

Protocol Summary

Brivault smart contract implements a tournament betting vault using the ERC4626 tokenized vault standard. It allows users to deposit an ERC20 asset to bet on a team, and at the end of the tournament, winners share the pool based on the value of their deposits.

Disclaimer

Bree makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

This document describes the manual code review of BriVault. The work was undertaken from November 6, 2025 to November 13, 2025.

This repository link was provided view repo

The following script list was included in our scope:

```
- src
   - briTechToken.sol
   - briVault.sol
```

Roles

```
Actors:
owner : Only the owner can set the winner after the event ends.
Users : Users have to send in asset to the contract (deposit +
   participation fee).
        users should not be able to deposit once the event starts.
        Users should only join events only after they have made deposit
            .
```

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	0
Info	0
Total	3

Findings

High

[H-1] Incorrect Minting of Shares to `msg.sender` Instead of `receiver` in `BriVault::deposit` function (ERC4626 Violation + Asset Loss Risk)

Description:

- Normal behavior - Under ERC-4626 specification, when a user calls deposit(assets, receiver), the vault must mint shares directly to the receiver. The caller (msg.sender) provides the tokens, but the receiver is the beneficiary and must receive the vault shares representing ownership of the deposited assets.
- Specific issue - In the current implementation, the briVault::deposit function incorrectly mints shares to msg.sender instead of receiver. As a result, whenever a smart contract deposits on behalf of a user such as a proxy contract the shares are credited to the proxy rather than the intended user. This breaks ERC-4626 compliance and causes misallocation of ownership, potentially locking users out of their assets and enabling privilege abuse or fee capture by unintended parties.

```
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;

         // record staked amount
         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);

         emit deposited(receiver, stakeAsset);

         return participantShares;
     }
```

Risk:

Likelihood: - Any deposit performed through a smart contract, relay, or aggregator automatically triggers the misallocation because msg.sender receives the shares instead of the receiver. - Automated vault interactions, multi-call transactions, and account abstraction systems regularly use proxy contracts, making this behavior likely in standard usage scenarios.

Impact: - Deposited assets do not result in ownership of shares for the intended receiver, preventing withdrawals or participation in rewards and governance. - Malicious or misconfigured intermediaries can capture shares, causing loss of funds or stuck assets for end users.

Proof of Concept:

This PoC reproduces the issue by deploying a contract DepositProxy that calls deposit(…) on behalf of a receiver. The test mints ERC20 tokens to the proxy, makes the proxy approve the vault, performs the deposit, and asserts that the proxy not the receiver received the minted shares.

- The proxy (msg.sender) incorrectly receives all minted shares
- The receiver receives none, despite depositing assets
- Confirms real-world exploitability via proxy/layered calls

```
contract DepositProxy {
    BriVault public vault;

    constructor(BriVault _briVault) {
        vault = _briVault;
    }

    function depositFor(uint256 amount, address receiver) public {
        vault.deposit(amount, receiver);
    }
}

function test_deposit_vulnerability_minting_to_proxy() public {
        DepositProxy proxy = new DepositProxy(briVault);

        //give proxy some tokens
        mockToken.mint(address(proxy), 10 ether);
        //impersonate proxy
        vm.startPrank(address(proxy));
        mockToken.approve(address(briVault), 5 ether);
        //balance before deposit
        uint256 proxyInitialShares = briVault.balanceOf(address(proxy))
            ;
        uint256 receiverInitialShares = briVault.balanceOf(user2);
        console.log("Proxy initial shares:", proxyInitialShares);
        console.log("Receiver initial shares:", receiverInitialShares);

        //call depositFor
        proxy.depositFor(5 ether, user2);
```

```
        vm.stopPrank();

        //balances after
        uint256 proxyFinalShares = briVault.balanceOf(address(proxy));
        uint256 receiverFinalShares = briVault.balanceOf(user2);
        console.log("Proxy final shares:", proxyFinalShares);
        console.log("Receiver final shares:", receiverFinalShares);

        //vulnerability check, receiver has no shares
        assertEq(receiverFinalShares, receiverInitialShares, "Receiver
            should get shares, but did not");
        // proxy incorrectly receives shares
        assertGt(
            proxyFinalShares, proxyInitialShares, "proxy should not
                receive shares, but it did due to vulnerability"
        );
    }
```

Recommended Mitigation: Update the minting logic so shares are minted to the correct address

```
- _mint(msg.sender, participantShares);
+ _mint(receiver, participantShares);
```

[H-2] Missing Duplicate Guard in joinEvent causing inflated Winner Shares and Misallocation of Funds

Description:

- Normal behavior: The joinEvent() function is intended to allow each legitimate participant to join an active event exactly once. The event logic later aggregates all participant shares to determine winner allocations, where each participant's contribution should be counted only once and proportionally to their deposit amount.
- Specific issue: The contract does not enforce uniqueness in event participation. The same user can call joinEvent() multiple times, causing their address to be added repeatedly to the participant list. When winner shares are calculated, all entries including duplicates are counted, allowing a user to artificially inflate their weighting in the final share distribution without depositing additional funds. This causes unfair allocations and potential financial loss.

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

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

```
8               revert invalidCountry();
9           }
10          // audit consistent event start check deposit has a different
                check
11          if (block.timestamp > eventStartDate) {
12              revert eventStarted();
13          }
14
15          userToCountry[msg.sender] = teams[countryId];
16
17          uint256 participantShares = balanceOf(msg.sender);
18          userSharesToCountry[msg.sender][countryId] = participantShares;
19
20          //audit does it check for duplicate users? but it may be
                intended to allow multiple entries
21  @>      usersAddress.push(msg.sender);
22
23          numberOfParticipants++;
24          totalParticipantShares += participantShares;
25
26          emit joinedEvent(msg.sender, countryId);
27      }
```

Risk:

Likelihood: - Duplicate entries occur whenever the contract is deployed in its current state, because joinEvent() does not restrict how many times a participant can join. - Attackers are fully incentivized to exploit this since it requires zero extra cost (multiple calls cost minimal gas and no additional tokens), making abuse extremely accessible and predictable.

Impact: - A malicious participant can manipulate reward distribution, receiving disproportionately high winner shares compared to their actual deposited amount. - Honest users experience direct financial loss, as their rightful share is diluted resulting in incorrect payout calculations, unfair settlements, and potentially severe trust/reputation damage to the platform or project.

Proof of Concept:

- Test scenario with 2 users betting on same winning team
- User1 exploits vulnerability by calling joinEvent() twice
- Tournament concludes, winner is set
- Internal _getWinnerShares() reveals inflated total due to duplicates
- Expected shares: user1Shares + user2Shares
- Actual shares: (2 user1Shares) + user2Shares
- 100% inflation for User1's contribution
- Direct proof of broken payout calculations

```
1  function test_joinEvent_duplicateUserInflatesWinnerShares() public {
```

```
        vm.warp(eventStartDate - 1);

        //user 1 deposits and joins event
        vm.startPrank(user1);
        mockToken.approve(address(briVault), 5 ether);
        uint256 user1shares = briVault.deposit(5 ether, user1);
        briVault.joinEvent(10);
        //user 1 joins again
        briVault.joinEvent(10);
        vm.stopPrank();

        //user 2 deposits and joins event
        vm.startPrank(user2);
        mockToken.approve(address(briVault), 5 ether);
        uint256 user2shares = briVault.deposit(5 ether, user2);
        briVault.joinEvent(10);
        vm.stopPrank();

        vm.warp(eventEndDate + 1);
        vm.startPrank(owner);

        briVault.setWinner(10);
        vm.stopPrank();

        uint256 totalWinnerShares = briVault.getWinnerSharesForTest();

        console.log("User1 shares:", user1shares);
        console.log("User2 shares:", user2shares);
        console.log("Total winner shares (with duplicate):",
            totalWinnerShares);
        console.log("Expected minimum winner shares (without duplicate)
            :", (user1shares) + user2shares);
        console.log("Expected maximum winner shares (with duplicate):",
             (user1shares * 2) + user2shares);

        //proof that duplicate entry inflated winner shares
        assertGt(
            totalWinnerShares,
            (user1shares * 2) + user2shares,
            "Total winner shares should account for duplicate entries"
        );
    }
```

To demonstrate the vulnerability, a test function getWinnerSharesForTest() was added to expose the internal _getWinnerShares() calculation. This allows direct observation of how duplicate entries inflate the winner share total.

```
function getWinnerSharesForTest() external returns (uint256) {
        return _getWinnerShares();
    }
```

Recommended Mitigation:

The root cause of this issue is the lack of a mechanism preventing duplicate event participation. The fix should focus on ensuring each user can only join once per event, unless multiple entries are explicitly part of the design.

```
+ mapping(address => bool) private hasJoined;


function joinEvent(uint256 countryIndex) external {
+    require(!hasJoined[msg.sender], "Already joined");
    // existing logic
    hasJoined[msg.sender] = true;
}
```

Medium

[M-1] Missing Share Balance Check Allows Users to Stay Registered Without Stake Burn

Description:

- Normal behavior: Users can deposit assets and join an event, where their stake is tracked via stakedAsset, their shares are minted, and their participation is recorded in usersAddress and related mappings. When a user calls cancelParticipation(), their staked assets should be refunded, their shares burned, and they should be fully removed from the event's participant tracking.
- Issue: The cancelParticipation() function only refunds assets and burns shares but does not remove the user from participant arrays or mappings (usersAddress, userToCountry, userSharesToCountry, totalParticipantShares). This allows users to remain registered as active participants even after canceling, causing their shares to incorrectly influence winner calculations and event logic.

```
function cancelParticipation() public {
        if (block.timestamp >= eventStartDate) {
            revert eventStarted();
        }

        uint256 refundAmount = stakedAsset[msg.sender];

    @>    // update total shares and participant tracking

        stakedAsset[msg.sender] = 0;

        uint256 shares = balanceOf(msg.sender);

```

```
14          _burn(msg.sender, shares);
15
16          IERC20(asset()).safeTransfer(msg.sender, refundAmount);
17      }
```

Risk:

Likelihood: - Every time a user deposits and joins an event, then calls cancelParticipation() before the event starts, the participant remains in the usersAddress array and related mappings - Any event with multiple participants has a high chance of accumulating “ghost participants” if multiple users cancel, because the function does not clean up logical participation state.

Impact: - Users who have canceled can still affect winner calculations, potentially skewing rewards or outcomes unfairly. - The event’s totalParticipantShares becomes inaccurate, undermining integrity of share-based logic and any dependent financial distributions.

Proof of Concept:

1. User deposits tokens and joins an event successfully (joinEvent() emits joinedEvent).
2. The user then cancels their participation (cancelParticipation()), triggering refund of their deposit.
3. After cancelation:
    - stakedAsset(user) returns 0 (as expected).
    - totalParticipantShares remains unchanged (unexpected).
    - usersAddress still contains the user address.

Below is PoC test that confirms ghost participants after they cancel participation:

```
1  function test_cancelParticipation_leaves_ghost_participant() public {
2          // user1 deposits and joins event
3          vm.startPrank(user1);
4          mockToken.approve(address(briVault), 5 ether);
5          briVault.deposit(5 ether, user1);
6          briVault.joinEvent(20);
7          vm.stopPrank();
8
9          // user2 deposits and joins event
10         vm.startPrank(user2);
11         mockToken.approve(address(briVault), 3 ether);
12         briVault.deposit(3 ether, user2);
13         briVault.joinEvent(30);
14         vm.stopPrank();
15
16         // record total shares BEFORE cancel
17         uint256 beforeTotalShares = briVault.totalParticipantShares();
18         assertGt(beforeTotalShares, 0);
19         console.log("Total participant shares before cancel:",
               beforeTotalShares);
```

```
20
21          // ensure user1 is present in usersAddress
22          bool presentInitially = false;
23          for (uint256 i = 0; i < 10; ++i) {
24              try briVault.usersAddress(i) returns (address a) {
25                  if (a == address(0)) break;
26                  if (a == user1) {
27                      presentInitially = true;
28                      break;
29                  }
30              } catch {
31                  break;
32              }
33          }
34          assertTrue(presentInitially, "user1 should be in usersAddress
                after deposit");
35
36          // user1 cancels participation
37          vm.startPrank(user1);
38          briVault.cancelParticipation();
39          vm.stopPrank();
40
41          // check stakedAsset is zeroed
42          assertEq(briVault.stakedAsset(user1), 0 ether, "stakedAsset
                should be zero after cancel");
43
44          // check totalParticipantShares decremented
45          uint256 afterTotalShares = briVault.totalParticipantShares();
46          console.log("Total participant shares after cancel:",
                afterTotalShares);
47          assertEq(afterTotalShares, beforeTotalShares, "
                totalParticipantShares should have been decremented but was
                not");
48
49          // check user1 still appears in usersAddress (ghost participant
                )
50          bool foundAfterCancel = false;
51          for (uint256 i = 0; i < 10; ++i) {
52              try briVault.usersAddress(i) returns (address a) {
53                  if (a == user1) {
54                      foundAfterCancel = true;
55                      break;
56                  }
57              } catch {
58                  break;
59              }
60          }
61          assertTrue(foundAfterCancel, "user1 still present in
                usersAddress after cancel -> ghost participant");
62      }
```

Recommended Mitigation:

- Remove user from participant list upon cancelation: When cancelParticipation() executes, the user's address should be removed from the usersAddress array or marked as inactive in a mapping such as isParticipant[user] = false.
- Adjust total shares accurately: Decrease totalParticipantShares by the user's current shares before zeroing them.

```
function cancelParticipation() public {
    if (block.timestamp >= eventStartDate) {
        revert eventStarted();
    }

    uint256 refundAmount = stakedAsset[msg.sender];
    uint256 shares = balanceOf(msg.sender);

    // Update total shares and participant tracking
+    totalParticipantShares -= shares;
+    isParticipant[msg.sender] = false;



    stakedAsset[msg.sender] = 0;
    _burn(msg.sender, shares);
    IERC20(asset()).safeTransfer(msg.sender, refundAmount);
}
Updates
```