BriVault Competitive Audit

github.com/lucasfhope

November 13, 2025

Contents

Competitive Audit Details

This competitive audit is a CodeHawks First Flight.

The findings described in this report correspond to the following commit hash:

```
1f515387d58149bf494dc4041b6214c2546b3b27
```

Protocol Summary

The 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.

Scope

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

Roles

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

Findings

Severity	Number of valid findings
High	4
Medium	1
Low	1
Total	**6**

[High-1] Multiple deposits for the same user will overwrite `stakedAsset` (SELECTED SUBMISSION)

Description

The protocol allows users to deposit assets multiple times. Each deposit increases the user's vault shares and potential payout if their team wins.

However, when recording the user's `stakedAsset` in the `deposit` function, the protocol overwrites the existing value instead of adding to it. This behavior causes an issue if the user decides to withdraw their assets before the event starts through `cancelParticipation`. In this case, the user will only be refunded the amount from their most recent `deposit` (minus the participation fee), effectively losing their earlier deposits.

This issue does not affect users who call `joinEvent` and participate in the event normally, since their shares are properly accounted for in the vault. The problem specifically affects users who make multiple deposits but later choose to cancel their participation before the event begins.

```
function deposit(uint256 assets, address receiver) public override
    returns (uint256) {
    ...

    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);


    emit deposited (receiver, stakeAsset);

    return participantShares;
}

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);
}
```

Risk

Likelihood:

This occurs whenever a user makes multiple deposits before deciding to cancel their participation, which is a realistic and easily reproducible scenario.

Impact:

The user will lose any assets from deposits before the most recent one, as the protocol only refunds the last recorded deposit amount. This would result in direct financial loss, where earlier deposits would be distributed between the winning team.

Proof of Concept

Add this test to the test suite in `test/briVault.t.sol`.

```
function
    testMultipleDepositsOverwritesStakedAssetsAndWillRefundWrongAmountUponCancelPart
    () public {
```

```
    vm.prank(owner);
    briVault.setCountry(countries);

    uint256 depositAmount = 5e18;
    uint256 base = 10000;
    uint256 feePerDeposit = (depositAmount * participationFeeBsp) / base;
    uint256 depositAmountMinusFee = depositAmount - feePerDeposit;
    uint256 user1AmountBeforeDeposits = mockToken.balanceOf(user1);

    vm.startPrank(user1);
    mockToken.approve(address(briVault), 2*depositAmount);
    briVault.deposit(depositAmount, user1);
    briVault.deposit(depositAmount, user1);
    uint256 amountAfterDeposits = mockToken.balanceOf(user1);
    briVault.cancelParticipation();
    vm.stopPrank();

    // user1 should be refunded both deposits minus fees but instead is refunded only one of the deposits minus both fees
    assert(mockToken.balanceOf(user1) == amountAfterDeposits + depositAmountMinusFee );
    assert(mockToken.balanceOf(user1) == user1AmountBeforeDeposits - depositAmount - feePerDeposit);
    assert(mockToken.balanceOf(user1) < user1AmountBeforeDeposits - 2 * feePerDeposit);
}
```

This test demonstrates that the user only receives a refund for their latest deposit rather than the total of all deposits.

Recommended Mitigation

Accumulate deposits instead of overwriting the previous value in `stakedAsset`.

```
function deposit(uint256 assets, address receiver) public override returns (uint256) {
    ...

    uint256 stakeAsset = assets - fee;

-   stakedAsset[receiver] = stakeAsset;
+   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;
}
```

[High-2] Users that `joinEvent` and then `deposit` later will have shares un accounted for in `totalWinningShares`

Description

The protocol expects users to first call `deposit` and then subsequently call `joinEvent`. This allows users to join a team and have their shares correctly recorded in `userSharesToCountry`. This mapping is later used to calculate `totalWinningShares` for the winning team so that vault assets can be distributed proportionally to winners based on their shares.

However, if a user deposits, joins the event, and then continues to deposit without calling `joinEvent` again, the additional shares from subsequent deposits will not be reflected in `userSharesToCountry`. As a result, when calculating `totalWinningShares`, the protocol will underestimate the true number of shares belonging to users on the winning team.

This allows a malicious user to withdraw more assets than their actual share, while other users that redeem later will be unable to claim their portion because the vault has been drained.

The root cause of this issue is the separation of the deposit and joinEvent logic, which leads to inconsistent accounting of participant shares.

Risk

Likelihood:

A malicious user can exploit this by depositing a minimal amount, calling `joinEvent`, and then making additional deposits. Since there is no restriction on depositing after joining, this situation is likely to occur, especially when users want to increase their stake before the event begins.

Impact:

The accounting for reward distribution will be incorrect, resulting in a low `totalWinningShares` value. Early withdrawers will receive more than they deserve, while later withdrawers may be blocked from claiming because the vault is either empty or holds too low of a balance.

Proof of Concept

Add this test to your test suite in `test/briVault.t.sol`.

```
function testDepositsAfterJoinEventWillIncreaseMailiciousUserPayoutAndLockTokensForOtherW() public {
    vm.prank(owner);
    briVault.setCountry(countries);

    uint256 depositAmount = 5e18;

    vm.startPrank(user1);
    mockToken.approve(address(briVault), 2 * depositAmount);
    briVault.deposit(2 * depositAmount, user1);
    briVault.joinEvent(0);
    vm.stopPrank();

    // malicious user
    vm.startPrank(user2);
    mockToken.approve(address(briVault), 2 * depositAmount);
    briVault.deposit(depositAmount, user2);
    briVault.joinEvent(0);
    briVault.deposit(depositAmount, user2);
    vm.stopPrank();

    // Note: both users have deposited the same amount and joined the same team

    vm.warp(eventEndDate + 1);
    vm.prank(owner);
    briVault.setWinner(0);

    uint256 totalInVault = mockToken.balanceOf(address(briVault));

    // malicious user
    vm.prank(user2);
    briVault.withdraw();
    uint256 user1WithdrawAmount = totalInVault - mockToken.balanceOf(address(briVault));
    console.log(user1WithdrawAmount);

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

This test shows that the malicious user (user2), who should have received 50% of the vault (10e18), actually receives 66.7% (13.33e18). When the other legitimate winner attempts to withdraw, the transaction reverts because the vault no longer holds enough tokens.

Recommended Mitigation

To prevent this issue, integrate the `deposit` and `joinEvent` logic. Doing so ensures that each deposit is tied to a specific team and that all shares are correctly included in team accounting.

```diff
-    function deposit(uint256 assets, address receiver) public override returns (uint256) {
+    function deposit(uint256 assets, address receiver, uint256 countryId) public override returns (uint256) {
         ...
+        joinEvent(countryId);
         return participantShares;
     }

-   function joinEvent(uint256 countryId) public {
+   function joinEvent(uint256 countryId) internal {
         ...
     }
```

It will also be be necessary to update the `joinEvent` logic to account for multiple deposits and attempts to change teams.

[High-3] Users can call `joinEvent` for multiple teams to dilute winner payout and lock tokens in the vault

Description

The protocol is designed so that each user can join only one team. Users who are on the winning team should share the assets in the vault proportionally to their shares.

However, once a user deposits assets into the vault, they can call `joinEvent` multiple times with different `countryId` values. The `countryId` from the last call will determine the team the user is officially assigned to, but each call creates a new entry in `userSharesToCountry` and appends the user's address to `userAddresses`. As a result, `_getWinnerShares` will compute an inflated `totalWinnerShares`, since it counts duplicate entries for the same user.

```
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) {
        revert invalidCountry();
    }

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


    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);
}

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

Risk

Likelihood:

There is currently no restriction preventing a user from calling `joinEvent` multiple times after depositing. This enables a malicious user to manipulate the vault's accounting with only the minimum deposit amount. Even regular users switching teams before the event starts can unintentionally create the same issue.

Impact:

This vulnerability can cause incorrect reward distribution and permanent asset locking. The `totalWinnerShares` will be artificially inflated, reducing the payout for legitimate winners and leaving a significant portion of the vault's assets unclaimable.

Proof of Concept

Add this test to the test suite in `test/briVault.t.sol`.

```
function
    testJoiningMultipleTeamsWillDiluteSharesOfWinnersAndLockTokensInTheVault
    () public {
    vm.prank(owner);
    briVault.setCountry(countries);

    uint256 depositAmount = 10e18;

    vm.startPrank(user1);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user1);
    // Joins and then switches team, but userSharesToCountry[user1][0]
        will persist
    briVault.joinEvent(0);
    briVault.joinEvent(1);
    vm.stopPrank();

    vm.startPrank(user2);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user2);
    briVault.joinEvent(0);
    vm.stopPrank();
    uint256 user2AmountAfterDeposit = mockToken.balanceOf(user2);

    vm.warp(eventEndDate + 1);
    vm.prank(owner);
    briVault.setWinner(0);

    uint256 totalAmountInVaultBeforeWithdraw = mockToken.balanceOf(
        address(briVault));

    console.log(briVault.userToCountry(user1));
    vm.prank(user1);
    vm.expectRevert(BriVault.didNotWin.selector);
    briVault.withdraw();

    vm.prank(user2);
    briVault.withdraw();
    uint256 user2WithdrawAmount = mockToken.balanceOf(user2) -
        user2AmountAfterDeposit;

    // user2 should have withdrawn their deposit + user1's deposit -
```

```
        fees
    // but they will only receive 1/3 of the total prize because user 1 joined multiple teams
    // adding an extra participant the the winning team and being in the userAddress twice
    assertEq(user2WithdrawAmount, totalAmountInVaultBeforeWithdraw / 3);

    // vault will have 2/3 of the total prize locked in it
    assertApproxEqAbs(mockToken.balanceOf(address(briVault)), 2 * (totalAmountInVaultBeforeWithdraw / 3), 2);
}
```

This test shows that a user who joins multiple teams inflates the vault's accounting. Even though user2 is the only legitimate participant on the winning team, they only receive one-third of the vault's assets, while two-thirds remain locked due to duplicated entries from user1.

Recommended Mitigation

Modify `joinEvent` to check whether the user has already joined a team. If they have, update their existing team instead of creating a new entry and pushing their address again. This ensures proper accounting and prevents duplicates in `userAddresses`.

```
      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) {
              revert invalidCountry();
          }

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

+         uint256 participantShares = balanceOf(msg.sender);
+         string currentCountry = userToCountry[msg.sender];

+         if (bytes(currentCountry).length != 0) {
+             // need to remove shares from previous country
+             delete userSharesToCountry[msg.sender][getCountryIndex(currentCountry)];
+             userSharesToCountry[msg.sender][countryId] = participantShares;
+             userToCountry[msg.sender] = teams[countryId];
+            // dont need to push again to usersAddress or change values
     of numberOfParticipants and totalParticipantShares
+        } else {
+            userToCountry[msg.sender] = teams[countryId];
+            userSharesToCountry[msg.sender][countryId] =
    participantShares;
+            usersAddress.push(msg.sender);
+            numberOfParticipants++;
+            totalParticipantShares += participantShares;
+         }

-        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);
     }
```

[High-4] `deposit` mints participant shares to `msg.sender` instead of `receiver`

Description

The protocol intends for the `receiver` specified in a `deposit` call to receive credit for the staked assets and corresponding participant shares of the vault.

However, the current implementation mints participant shares to `msg.sender` rather than the `receiver`. As a result, if the caller (`msg.sender`) and the `receiver` are different addresses, the shares will be issued to the wrong account.

```
function deposit(uint256 assets, address receiver) public override
    returns (uint256) {
    ..

    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);

     emit deposited (receiver, stakeAsset);

     return participantShares;
}

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

     ...
}
```

Risk

Likelihood:

This issue will occur whenever a user calls `deposit` on behalf of another address (when `receiver` != `msg.sender`). The effect only becomes critical if the participant shares are not transferred to the receiver before they attempt to call `joinEvent`.

Impact:

The `receiver` does not receive the participant shares, but they can still call joinEvent because `stakedAsset[receiver]` is populated. However, their team's shares will not be reflected in `userSharesToCountry`. If the receiver's team later wins, the `totalWinningShares` will be undercounted.

Moreover, since the `receiver` lacks the actual participant shares, they cannot redeem their winnings unless those shares are manually transferred from the original depositor. If this transfer occurs after `joinEvent`, share accounting will remain incorrect, allowing some winning users to withdraw a disproportionate amount of vault assets while other winning users will be prevented from withdrawing.

Proof of Concept

Add this test to the test suite in `test/briVault.t.sol`.

```
function testParticipationSharesGoToSenderRatherThanReceiverInDeposit()
    public {
    vm.prank(owner);
    briVault.setCountry(countries);

    uint256 base = 10000;
    uint256 depositAmount = 5e18;
    uint256 feeAmount = (depositAmount * participationFeeBsp) / base;

    vm.startPrank(user1);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user2);
    vm.expectRevert(BriVault.noDeposit.selector);
    briVault.joinEvent(0);
    vm.stopPrank();

    vm.prank(user2);
    briVault.joinEvent(0);

    vm.startPrank(user3);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user3);
    briVault.joinEvent(0);
    vm.stopPrank();

    vm.warp(eventEndDate + 1);

    vm.prank(owner);
    briVault.setWinner(0);

    uint256 user2BalanceBeforeWithdraw = mockToken.balanceOf(user2);
    vm.prank(user2);
    briVault.withdraw();

    assert(user2BalanceBeforeWithdraw == mockToken.balanceOf(user2));

    vm.expectRevert(BriVault.didNotWin.selector);
    vm.startPrank(user1);
    briVault.withdraw();
    IERC20(briVault).transfer(user2, briVault.balanceOf(user1));
    vm.stopPrank();
    vm.prank(user2);
    briVault.withdraw();

    assert(mockToken.balanceOf(user2) == user2BalanceBeforeWithdraw +
        2*(depositAmount - feeAmount));
}
```

This test demonstrates that when user1 deposits on behalf of user2, the shares are minted to user1 instead of user2. This makes it so that user2 can still join the event but initially cannot withdraw winnings because they do not hold the shares. After user1 transfers the shares to user2, user2 can successfully withdraw, but the vault's internal accounting will be incorrect, leading to wrong share calculations.

Recommended Mitigation

Mint participant shares to the `receiver` address in `deposit`.

```
    function deposit(uint256 assets, address receiver) public override
       returns (uint256) {
        ..

        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(receiver, participantShares);

        emit deposited (receiver, stakeAsset);

        return participantShares;
    }
```

[Medium-1] Potential Denial of Service due to an unbounded loop in `setWinner`

Description

When the `eventEndDate` has passed, the owner is expected to call `setWinner`, which calculates the total number of shares for the winning team.

However, `_getWinnerShares`, which is called within `setWinner`, iterates over the entire `userAddresses` array to compute `totalWinnerShares`. This loop is unbounded, meaning that gas usage grows linearly with the number of users. As the array size increases, the cost of executing setWinner also increases.

If `userAddresses` becomes too large, the gas cost may exceed the block gas limit, causing the transaction to revert and preventing the owner from finalizing the event.

```
function _getWinnerShares () internal returns (uint256) {

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

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) {
        revert invalidCountry();
    }

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

    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);
}
```

Risk

Likelihood: Medium

Every call to joinEvent appends the user's address to `userAddresses`, even if the same user calls `joinEvent` multiple times. As a result, the array can grow very large with a sufficient amount of users.

Impact: High

The owner may be unable to execute `setWinner` due to excessive gas costs, effectively freezing all vault assets and preventing users from claiming rewards. This results in a complete denial of service for the protocol's core functionality.

Proof of Concept

Add this test to your test suite in `test/briVault.t.sol`.

```
function testSetWinnerDOSWhenThereAreManyUsers() public {
    vm.prank(owner);
    briVault.setCountry(countries);

    uint256 depositAmount = 0.005 ether;
    vm.startPrank(user1);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user1);
    for (uint256 i = 0; i < 30_000; i++) { briVault.joinEvent(i %
        countries.length); }
    vm.stopPrank();

    vm.warp(eventEndDate + 1);

    bytes memory data = abi.encodeWithSelector(BriVault.setWinner.
        selector, 0);
    vm.prank(owner);
    uint256 g0 = gasleft();
    (bool ok, ) = address(briVault).call{gas: 30_000_000}(data);
    uint256 used = g0 - gasleft();

    assert(!ok);
    console.log("setWinner gas used for 30k entries in userAddresses:",
         used);
    }
```

This test shows that a malicious user can inflate userAddresses to 30,000 entries. At this size, setWinner consumes over 30 million gas, which is around Ethereum's block gas limit, causing it to revert or become very expensive for the owner.

Recommended Mitigation

Use a mapping to track total team shares instead of iterating through an array of user addresses.

```

-   mapping(address => mapping(uint256 => uint256)) public
    userSharesToCountry;
+   mapping(uint256 => uint256) public countryIdToShares;

    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) {
            revert invalidCountry();
        }

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

        userToCountry[msg.sender] = teams[countryId];

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

-       usersAddress.push(msg.sender);

-       numberOfParticipants++;
-       totalParticipantShares += participantShares;

        emit joinedEvent(msg.sender, countryId);
    }

-   function _getWinnerShares () internal returns (uint256) {
+   function _getWinnerShares (uint256 countryIndex) internal returns (
   uint256) {

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

+       totalWinnerShares = countryIdToShares[countryIndex];

        return totalWinnerShares;
    }
```

This change removes the need for an unbounded loop and ensures that the gas cost of setWinner remains constant, regardless of the number of participants.

Additional changes would need to be made to prevent multiple calls to join a team by the same user and account for multiple deposits.

[Low-1] `deposit` emits the `receiver` as the depositor

Description

The `deposited` event is intended to emit the address that deposited into the vault along with the amount deposited.

However, the `deposit` function currently emits the `receiver` address as the depositor instead of `msg.sender`.

```
@>  event deposited (address indexed _depositor, uint256 _value);

    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);

@>      emit deposited (receiver, stakeAsset);

        return participantShares;
    }
```

Risk

Likelihood: Medium

This occurs whenever a user calls `deposit` with `receiver` set to a different address.

Impact: Low

The emitted event data will be incorrect, potentially causing off-chain indexers to attribute deposits to the wrong address.

Recommended Mitigation

Emit `msg.sender` instead of `receiver` in the `deposited` event.

```
function deposit(uint256 assets, address receiver) public override
    returns (uint256) {
     ...

-    emit deposited (receiver, stakeAsset);
+    emit deposited (msg.sender, stakeAsset);

     return participantShares;
}
```