Introduction

A time-boxed security review of the **TheFund** protocol was done by **Joe Dakwa**, with a focus on the security aspects of the application's smart contracts implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About Joe Dakwa

I an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, I does my best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews.

About The Fund

The contract is serving as the underlying entity to interact with one or more third party protocols. It allows users to deposit packages into the Fund and only whitelisted users can do so.

Observations

The contract Fund interacts with third party contract with IERC20 interface via busd . The function Fund.withdrawToken interacts with third party contract with IERC20 interface via token .

Privileged Roles & Actors

The owner is able to do the following:

```
 function setPackagePrice(uint newPrice) external onlyOwner

function setDevFee(uint256 newFee) external onlyOwner

function setRoot(bytes32 _root) external onlyOwner

function setFundAddresses(address _fund, address _dev) external onlyOwner

function withdrawToken(address token) external onlyOwner

function withdrawAll() external onlyOwner

function withdraw(uint256 amount) external onlyOwner
```

The centralisation risks above have been addressed by Certik in a previous audit report.

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

Scope

The following smart contracts were in scope of the audit:

- `The Fund`
- `Open Zeppelin Contracts`
- `Interfaces'
- `Ownable.sol'
- 'Context.sol'
- 'MerkleProof.sol'

Findings Summary

ID	Title	Severity	Status
[H-01]	Owner can change price variable whilst the deposit function is executing which can result in loss of funds for early depositors.	High	TBD
[M-01]	endIndex and startIndex lengths are not checked leading to out of bounds access	Medium	TBD
[M-02]	depositsOfOwnerInRange does not check if the user has any deposits	Medium	TBD

Detailed Findings

[H-01] Owner can change price variable whilst the deposit function is executing which can result in loss of funds for early depositors.

Severity

Impact: High, because early depositors will lose out anytime before fee structure is changed

Likelihood: Medium, because there is responsibility on the owner to set the fee structure, which will happen in real time

Description

In function deposit, users deposit packages into the fund.

Once deposited, the fund stores the package and deposit info in the deposits array.

''' // store deposited package amount on storage deposits[id] = DepositInfo(_msgSender(), amount, busdAmount, block.timestamp); '''

The public price variable of 250 eth, prior to this, is checked against the amount of BUSD sent by the user.

```
    // total BUSD deposited
        uint256 busdAmount = amount * price;
```

However, the vulnerability comes from the fact that before the transaction is completed, the owner can change the price variable.

```
     function setPackagePrice(uint newPrice) external onlyOwner {
        require(newPrice > 0, 'Zero Price');
        price = newPrice;
    }
```

The user will receive a notifcation that their transaction completed with the assumption that the price was fixed at 250 ETH.

But lets say the owner increases the price to 500 ETH, the users initial balance will be lower than the amount of BUSD they sent.

Recommendations

Fetch the current price variable before calculating the total of BUSD deposited.

```
  // Retrieve the current price
    uint256 currentPrice = price;
```

Then, check if the price has changed before the deposit process and revert if so.

```
    // Check if the price has changed during the deposit process
    if (currentPrice != price) {
        revert("Price has changed during deposit");
    }
```

[M-01] endIndex and startIndex lengths are not checked leading to out of bounds access

Severity

Impact: High, because this will result in unexpected values not processed

Likelihood: Low, because it will take some time for both values to possibly exceed the expected values

Description

In depositsOfOwnerInRange, the function loops through the deposits array and returns the deposits for a given user.

However, function does not sufficiently validate the input parameters startIndex and endIndex. Without proper validation, these values can be manipulated, leading to unexpected behavior or errors.

`uint256 length = endIndex - startIndex;`

Recommendations

Consider adding the below, before the division calculation difference check to the function that checks if the startIndex and endIndex are within the bounds of the deposits array. Then you can conduct the division.

`require(startIndex <= endIndex, "Invalid range");`

Then you can add the rest of the below logic to further validate the input parameters.

```
  uint256 userDepositCount = userIDs[owner].length;
    require(endIndex <= userDepositCount, "End index exceeds user's deposit count");

    uint256 length = endIndex - startIndex;
    DepositInfo[] memory userDeposits = new DepositInfo[](length);
```

The function should look something like this:

```
function depositsOfOwnerInRange(address owner, uint startIndex, uint endIndex) external view returns (DepositInfo[] memory) {
    require(startIndex <= endIndex, "Invalid range");

    uint256 userDepositCount = userIDs[owner].length;
    require(endIndex <= userDepositCount, "End index exceeds user's deposit count");

    uint256 length = endIndex - startIndex;
    DepositInfo[] memory userDeposits = new DepositInfo[](length);

    for (uint256 i = startIndex; i < endIndex; i++) {
        // Check if a deposit exists for this ID
        if (i < userDepositCount) {
            userDeposits[i - startIndex] = deposits[userIDs[owner][i]];
        }
    }

    return userDeposits;
}
```

[M-02] depositsOfOwnerInRange does not check if the user has any deposits

Severity

Impact: High, because this will result in the array not checking, due to everytime the function is called

Likelihood: Low, because its only a view function, with a risk of DOS if called excessively.

Description

In `depositsOfOwnerInRange`, the function loops through the deposits array and returns the deposits.

The current implementation of the depositsOfOwnerInRange function does not check whether a deposit exists for a particular ID. As a result, if there are gaps in the deposit IDs (e.g., if some deposits have been deleted), the resulting array may contain uninitialized or empty elements.

The loop used to retrieve deposit information does not check whether a deposit exists for a particular ID. As a result, if there are gaps in the deposit IDs (e.g., if some deposits have been deleted), the resulting array may contain uninitialized or empty elements.

Recommendations

Consider adding a check to the loop that checks if the deposit exists for a particular ID.

```
if (i < userDepositCount) {
        userDeposits[i - startIndex] = deposits[userIDs[owner][i]];
    }
```

This is issue links in with the above [M-01]