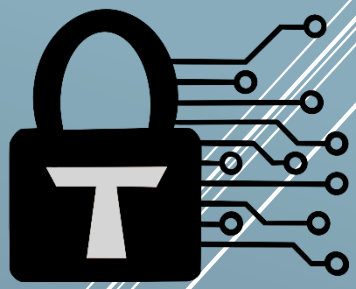
Trust Security



Smart Contract Audit

HoneyJar - LST

06/02/2025

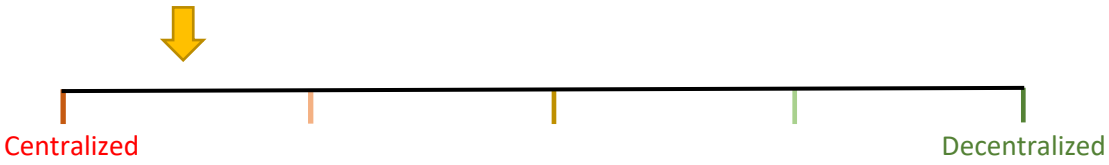
Executive summary

Category	Liquid Staking
Audited file count	1
Lines of Code	198
Auditor	HollaDieWaldfee, SpicyMeatball
Time period	31/01/2025-04/02/2025

Findings

Severity	Total	Fixed	Acknowledged
High	1	1	-
Medium	5	5	-
Low	1	1	-

Centralization score



Signature

Digitally signed by Trust
Date: 2025.02.06 10:51:04 Z00'00'

Document properties

Versioning

Version	Date	Description
0.1	04/02/2025	Client report
0.2	06/02/2025	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

The following files are in scope of the audit:

- src/fatBERA.sol

Repository details

- **Repository URL:** https://github.com/0xHoneyJar/LST
- **Commit hash:** 7523fdcfd1e71273ff5c2edba53cf925fb344322
- **Mitigation hash:** 943f96b82919dec89332afa9507ed0f8fc905137

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Auditor for Trust Security and Renascence Labs and Lead Senior Watson at Sherlock.

SpicyMeatball is a member of the Code4rena Zenith and has reported over 100 bugs in various DeFi protocols.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks.
Documentation	**Moderate**	The code is commented but lacks further documentation.
Best practices	**Good**	Project adheres to industry standards.
Centralization risks	**Severe**	The protocol owner is fully trusted to handle user funds.

.

Findings

High severity findings

TRST-H-1 Transferring shares allows users to steal rewards from the vault

- **Category:** Logical issues
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

The *fatBera* contract distributes rewards to users based on the number of shares they hold. Each time a user's share balance is updated, the protocol calls *_updateRewards()* to accrue rewards and update **userRewardPerSharePaid**, ensuring the same rewards cannot be claimed multiple times:

```
    function _updateRewards(address account, address token) internal {
        RewardData storage data = rewardData[token];
        uint256 accountShares = balanceOf(account);

        if (accountShares > 0) {
            uint256 earnedPerShare = data.rewardPerShareStored -
userRewardPerSharePaid[token][account];
            rewards[token][account] += FixedPointMathLib.mulDiv(
                accountShares,
                earnedPerShare,
                1e18
            );
        }
>>      userRewardPerSharePaid[token][account] = data.rewardPerShareStored;
    }
```

While *_updateRewards()* is correctly called when a user mints or burns shares, it is not called when a user transfers shares to another address. This allows a user to claim the same rewards multiple times by transferring shares to an address with **userRewardPerSharePaid** set to zero.

Recommended mitigation

To prevent this exploit, consider one of the following approaches:

- Call *_updateRewards()* on share transfers to ensure rewards are correctly updated.
- Make minted shares soulbound (non-transferable) to the minter's address.

Team response

Fixed in commit 77c380b.

Mitigation review

Fixed by overriding *ERC20Upgradeable._update()* and calling *_updateRewards()*. Furthermore, a whitelist for trusted vaults has been implemented. When *fatBERA* is transferred to a trusted vault, the user's effective rewards balance remains unchanged such

that rewards do not accrue to the vault. This mechanism introduces a trust assumption that a trusted and whitelisted vault does only allow *fatBERA* withdrawals to the same user address that has made the deposit, and that deposit and withdrawal amounts are equal over the lifetime of a user.

Medium severity findings

TRST-M-1 *withdrawPrincipal()* decrements depositPrincipal which breaks deposit limit

- **Category:** Accounting errors
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

In *withdrawPrincipal()*, the admin is allowed to withdraw user deposits and the **depositPrincipal** variable tracks how much user deposits are available to be withdrawn. As a side-effect of decrementing **depositPrincipal**, the deposit limit is effectively reset, and users can again deposit WBERA until **depositPrincipal > maxDeposits**.

The root cause is the dual use of **depositPrincipal** which can only be used to either track available deposits to be withdrawn by the admin or overall user deposits to enforce the deposit limit.

Recommended mitigation

It is recommended to not use **depositPrincipal** to enforce deposit limits, and instead to use the *totalSupply()* which has a 1:1 relationship with the deposited WBERA.

Team response

Fixed in commit 215c6ef.

Mitigation review

Verified, the recommendation has been implemented.

TRST-M-2 *notifyRewardAmount()* is vulnerable to sandwiching once withdrawals are enabled

- **Category:** Sandwiching issues
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

The *notifyRewardAmount()* function pays out rewards immediately. Once withdrawals are enabled, users can deposit before rewards are paid out, and withdraw immediately after rewards are paid out, earning the same rewards as users that have deposited for the whole time. Any entity that proposes blocks on Berachain will be able to sandwich the call to *notifyRewardAmount()* by doing the following:

1. Transaction 1 borrows WBERA from a lending protocol and deposits it into *fatBERA*.
2. Transaction 2 contains the call to *notifyRewardAmount()*.
3. Transaction 3 claims rewards, withdraws WBERA and repays the borrowed WBERA.

A less sophisticated attack is simply to observe when *notifyRewardAmount()* tends to get called and to deposit around that time.

Withdrawals are not enabled initially, but clearly paying out rewards immediately will allow different strategies for sophisticated actors to extract value from regular users of the protocol.

Recommended mitigation

Implement a design where rewards are paid out over some duration to reward users for the time that they have deposited into the protocol.

A simple but sufficient implementation of this can be found in the Curve *MultiRewards* contract.

Team response

Fixed in commit d058572.

Mitigation review

Verified, the recommendation has been implemented.

TRST-M-3 Reward calculations with 18 decimals precision are prone to rounding loss

- **Category:** Rounding issues
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

The rewards per share are accumulated in the **rewardPerShareStored** variable which has a precision of 18 decimals.

```
data.rewardPerShareStored += FixedPointMathLib.fullMulDiv(
    rewardAmount,
    1e18,
    totalSharesCurrent
);
```

Any token can be used as a reward token. Reward tokens probably have 18 decimals, but this is not enforced. Also, **totalSharesCurrent** can be capped by using a deposit limit, but as the protocol grows, it should be possible to raise this limit.

Therefore, it is possible to have configurations with a token like WBTC as reward token and **totalSharesCurrent** on the order of **1e6*1e18 = 1e24** or higher (this depends on the BERA price). In such a scenario, assuming **totalSharesCurrent = 2e24**, if around $1k of WBTC (**1e6**) is paid out in rewards, the calculation can round to zero (**1e6*1e18/2e24 = 0**). Even before rounding results in zero rewards, the rounding error can become unacceptable.

Recommended mitigation

Increase precision from 18 decimals to 36 decimals. This provides sufficient precision even in an adverse scenario (e.g., **1e5 * 1e36 / 1e30 = 1e11**). In addition, *_updateRewards()* and *previewRewards()* need to use *fullMulDiv()* to avoid an intermediate overflow.

```
diff --git a/src/fatBERA.sol b/src/fatBERA.sol
index 6e35412..804d0f7 100644
--- a/src/fatBERA.sol
```

```
+++ b/src/fatBERA.sol
@@ -145,7 +145,7 @@ contract fatBERA is
         if (totalSharesCurrent > 0) {
             data.rewardPerShareStored += FixedPointMathLib.fullMulDiv(
                 rewardAmount,
-                1e18,
+                1e36,
                 totalSharesCurrent
             );
         }
@@ -289,10 +289,10 @@ contract fatBERA is
         uint256 earnedPerShare = data.rewardPerShareStored -
userRewardPerSharePaid[token][account];
         uint256 accountShares = balanceOf(account);

-        return rewards[token][account] + FixedPointMathLib.mulDiv(
+        return rewards[token][account] + FixedPointMathLib.fullMulDiv(
             accountShares,
             earnedPerShare,
-            1e18
+            1e36
         );
     }

@@ -316,10 +316,10 @@ contract fatBERA is

         if (accountShares > 0) {
             uint256 earnedPerShare = data.rewardPerShareStored -
userRewardPerSharePaid[token][account];
-            rewards[token][account] += FixedPointMathLib.mulDiv(
+            rewards[token][account] += FixedPointMathLib.fullMulDiv(
                 accountShares,
                 earnedPerShare,
-                1e18
+                1e36
             );
         }
         userRewardPerSharePaid[token][account] = data.rewardPerShareStored;
```

Team response

Fixed in commit [af58298](af58298) and [c44aa62](c44aa62).

Mitigation review

The recommended mitigation has been implemented. Furthermore, the mitigation for TRST-M-2 has introduced another source for rounding errors (when calculating **rewardRate**), which has also been addressed.

TRST-M-4 Some ERC-4626 view functions return incorrect values

- **Category:** EIP compliance issues
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

fatBera extends OpenZeppelin's *ERC4626Upgradeable* contract with custom logic. However, some ERC-4626 functions need to be overridden to reflect accurate values.

- *maxDeposit()* and *maxMint()* currently return **type(uint256).max**, which is incorrect since the vault imposes a limit on how many assets can be deposited:

```
    function deposit(uint256 assets, address receiver) public virtual override
returns (uint256) {
        if (depositPrincipal + assets > maxDeposits) revert ExceedsMaxDeposits();
```

- Similarly, *maxWithdraw()* and *maxRedeem()* fail to account for vault restrictions. Withdrawals are not possible when the vault is paused, and the owner can stake assets in external protocols, further limiting available withdrawals.

Recommended mitigation

The native ERC-4626 functions should be overridden to reflect the actual state of the protocol.

Additionally, once *maxDeposit()* is correctly implemented, redundant limit checks in *fatBera.deposit()* and *fatBera.mint()* can be removed, as they are already enforced by the parent ERC4626 contract. However, *depositNative()* should explicitly use *maxDeposit()* to maintain proper validation.

It is of high importance to make the code fully compliant with any declared specification as integrating contracts are free to assume the properties defined in the spec hold. As a result, unexpected and severe damage could occur.

Team response

Fixed in commit ee49c24.

Mitigation review

Verified, the recommendation has been implemented. When the upgrade to enable withdrawals is performed, *maxWithdraw()* and *maxRedeem()* need to be modified since they currently return zero.

TRST-M-5 *fatBERA* contract is not upgradeable

- **Category:** Upgradeability issues
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

It is important that *fatBERA* is upgradeable to implement withdrawals from the consensus layer once Berachain enables it. However, while *fatBERA* is deployed behind a *ERC1967Proxy*, it does not inherit *UUPSUpgradeable*, and so it does not contain any logic to perform an upgrade.

As a result, the contract can never be upgraded and transitioning to a new implementation would require that the admin withdraws all principal and migrates to a new contract.

Recommended mitigation

Inherit *UUPSUpgradeable* and override *_authorizeUpgrade()*.

```
diff --git a/src/fatBERA.sol b/src/fatBERA.sol
index 6e35412..d48987b 100644
--- a/src/fatBERA.sol
+++ b/src/fatBERA.sol
@@ -8,6 +8,7 @@ import {PausableUpgradeable} from "@openzeppelin/contracts-
upgradeable/utils/Pau
 import {FixedPointMathLib} from "solady/utils/FixedPointMathLib.sol";
 import {ReentrancyGuardUpgradeable} from "@openzeppelin/contracts-
upgradeable/utils/ReentrancyGuardUpgradeable.sol";
 import {AccessControlUpgradeable} from "@openzeppelin/contracts-
upgradeable/access/AccessControlUpgradeable.sol";
+import {UUPSUpgradeable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";

 /*##############################################################
                            INTERFACES
@@ -21,7 +22,8 @@ contract fatBERA is
     ERC4626Upgradeable,
     PausableUpgradeable,
     ReentrancyGuardUpgradeable,
-    AccessControlUpgradeable
+    AccessControlUpgradeable,
+    UUPSUpgradeable
 {
     using SafeERC20 for IERC20;
     using FixedPointMathLib for uint256;
@@ -99,6 +101,8 @@ contract fatBERA is
         _unpause();
     }

+    function _authorizeUpgrade(address newImplementation) internal override
onlyRole(DEFAULT_ADMIN_ROLE) {}
```

Team response

Fixed in commit [9173e6f](#) and [c8b6578](#).

Mitigation review

Verified, the recommendation has been implemented.

Low severity findings

TRST-L-1 *notifyRewardAmount()* incorrectly increments totalRewards when totalSupply is zero

- **Category:** Accounting errors
- **Source:** fatBERA.sol
- **Status:** Fixed

Description

When **totalSupply()==0**, *notifyRewardAmount()* does not increment **rewardPerShareStored** to avoid division by zero. However, **totalRewards** is still incremented even though no rewards will be paid out. Since **totalRewards** is not used internally, this does not affect internal accounting, but it can lead to issues in integrations that read this variable.

Recommended mitigation

If **totalShares = 0**, it is recommended to revert without making any state changes.

```
diff --git a/src/fatBERA.sol b/src/fatBERA.sol
index 6e35412..0ef4b7b 100644
--- a/src/fatBERA.sol
+++ b/src/fatBERA.sol
@@ -35,6 +35,7 @@ contract fatBERA is
     error InvalidMaxDeposits();
     error ExceedsAvailableRewards();
     error InvalidToken();
+    error ZeroShares();
     /*##############################################################
                             STRUCTS
     ##############################################################*/
@@ -129,6 +130,8 @@ contract fatBERA is
     function notifyRewardAmount(address token, uint256 rewardAmount) external
onlyRole(REWARD_NOTIFIER_ROLE) {
         if (rewardAmount <= 0) revert ZeroRewards();
         if (token == address(0)) revert InvalidToken();
+        uint256 totalSharesCurrent = totalSupply();
+        if (totalSharesCurrent == 0) revert ZeroShares();

         IERC20 rewardToken = IERC20(token);
         if (rewardAmount > rewardToken.balanceOf(address(this))) revert
ExceedsAvailableRewards();
@@ -140,15 +143,12 @@ contract fatBERA is
         }

         RewardData storage data = rewardData[token];
-        uint256 totalSharesCurrent = totalSupply();

-        if (totalSharesCurrent > 0) {
-            data.rewardPerShareStored += FixedPointMathLib.fullMulDiv(
-                rewardAmount,
-                1e18,
-                totalSharesCurrent
-            );
-        }
+        data.rewardPerShareStored += FixedPointMathLib.fullMulDiv(
+            rewardAmount,
+            1e18,
+            totalSharesCurrent
+        );
         data.totalRewards += rewardAmount;
         emit RewardAdded(token, rewardAmount);
```

```
    }
```

Team response

Fixed in commit ffab9ab and a67cc42.

Mitigation review

Fixed by making an initial deposit of 1 WBERA which will accrue all rewards when all other users have withdrawn.

Additional recommendations

TRST-R-1 The *receive()* function should be removed

It is not necessary for the *fatBERA* contract to hold native BERA tokens, so the *receive()* function should be removed to avoid anyone sending native tokens by accident.

```
diff --git a/src/fatBERA.sol b/src/fatBERA.sol
index 6e35412..d833817 100644
--- a/src/fatBERA.sol
+++ b/src/fatBERA.sol
@@ -324,8 +324,4 @@ contract fatBERA is
         }
         userRewardPerSharePaid[token][account] = data.rewardPerShareStored;
     }
-
-    /*##############################################################
-    ##############################################################*/
-    receive() external payable {}
 }
```

TRST-R-2 Limit number of reward tokens

By calling *notifyRewardAmount()*, the reward notifier can add an arbitrary number of reward tokens. It is recommended to revert if the length of the **rewardTokens** array exceeds a reasonable limit like **10** or **20**.

```
diff --git a/src/fatBERA.sol b/src/fatBERA.sol
index 6e35412..fee51c2 100644
--- a/src/fatBERA.sol
+++ b/src/fatBERA.sol
@@ -35,6 +35,7 @@ contract fatBERA is
     error InvalidMaxDeposits();
     error ExceedsAvailableRewards();
     error InvalidToken();
+    error ExceedsMaxRewardsLength();
     /*##############################################################
                             STRUCTS
     ##############################################################*/
@@ -63,6 +64,8 @@ contract fatBERA is
     // Define role constants
     bytes32 public constant REWARD_NOTIFIER_ROLE = keccak256("REWARD_NOTIFIER_ROLE");

+    uint256 public constant MAX_REWARD_TOKENS = 10;
+
     /*##############################################################
                             CONSTRUCTOR
     ##############################################################*/
@@ -136,6 +139,7 @@ contract fatBERA is
         // Add to reward tokens list if new
         if (!isRewardToken[token]) {
             rewardTokens.push(token);
+            if (rewardTokens.length > MAX_REWARD_TOKENS) revert
ExceedsMaxRewardsLength();
             isRewardToken[token] = true;
         }
```

Importantly, this recommendation does not remove the trust assumptions for the reward notifier and just serves as a sanity check.

It should also be considered whether adding reward tokens can be limited to the admin, so that the reward notifier can only interact with trusted tokens. This would be a step beyond limiting the number of reward tokens.

TRST-R-3 *notifyRewardAmount()* does not take into account pending rewards

When *notifyRewardAmount()* is called, it is checked that **rewardAmount** is less or equal to the contract's balance. However, this check does not consider that some of the balance can be in the contract to pay out pending rewards.

Since the reward notifier is fully trusted, this is not a security issue, but it is still recommended to provide a meaningful mechanism to prevent double accounting. A simple way to achieve this is to transfer **rewardAmount** from the reward notifier.

```
diff --git a/src/fatBERA.sol b/src/fatBERA.sol
index 6e35412..8af9e50 100644
--- a/src/fatBERA.sol
+++ b/src/fatBERA.sol
@@ -131,7 +131,7 @@ contract fatBERA is
         if (token == address(0)) revert InvalidToken();

         IERC20 rewardToken = IERC20(token);
-        if (rewardAmount > rewardToken.balanceOf(address(this))) revert
ExceedsAvailableRewards();
+        rewardToken.safeTransferFrom(msg.sender, address(this), rewardAmount);

         // Add to reward tokens list if new
         if (!isRewardToken[token]) {
```

The same caveat as in TRST-R-2 applies here that this change does not remove the trust assumptions for the reward notifier. However, this fix does reduce the trust assumptions significantly because it prevents the reward notifier from stealing user deposits.

Centralization risks

TRST-CR-1 Admin is fully trusted

The admin is fully trusted since it can call *withdrawPrincipal()* to withdraw and stake user deposits. Beyond that, the *fatBERA* contract is upgradeable to arbitrary implementations.

TRST-CR-2 Reward notifier is fully trusted

The reward notifier role is granted by the admin and is allowed to call *notifyRewardAmount()*. Reward notifiers are therefore trusted to handle rewards and, in the worst case, can steal those rewards by not notifying the *fatBERA* contract. Beyond that, they can only cause availability impact by overflowing calculations that could be resolved with an upgrade.

TRST-CR-2 Whitelisted vaults must provide one to one mapping of user deposits and withdrawals

The mitigation for TRST-H-1 has introduced a trust assumption for whitelisted vaults. Users that have deposited *fatBERA* must only be allowed to withdraw the *fatBERA* to their own address, and the amount that can be withdrawn by a user must equal the amount that has been deposited by the user. Failure to comply with this can lead to internal accounting errors in *fatBERA*, prevent users from withdrawing *fatBERA* from whitelisted vaults and loss of rewards.

Systemic risks

TRST-SR-1 Staking risk

WBERA that is deposited by users is withdrawn by the admin and staked in the Berachain consensus layer. Thus, the deposited funds are exposed to all risks involved in running validators. These risks go beyond the centralization risk of trusting the protocol admin and the entities that run the validators, and include technical risks, such as new exploits in the consensus layer, or network downtime that could lead to slashing.