HoneyJar fatBera

Security Review

Cantina Managed review by:
Chris Smith, Security Researcher
Cryptara, Security Researcher

March 28, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Honeyjar is building fatBERA: a liquid staking token (LST) designed for users looking to stake their BERA with THJ validators on Berachain.

From Feb 6th to Feb 7th the Cantina team conducted a review of LST on commit hash 941db283. The team identified a total of **13** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 3
- Low Risk: 4
- Gas Optimizations: 4
- Informational: 2

3 Findings

3.1 Medium Risk

3.1.1 Admin can lock funds

Severity: Medium Risk

Context: fatBERA.sol#L206, fatBERA.sol#L252

Description: While there are checks that the owner (admin/reward notifier) doesn't misbehave by setting a reward token to 0 or setting the duration to 0, there are no checks that the owner doesn't use the fatBERA tokens as reward tokens. If they do, these tokens would be lost to the system (see the proof of concept). It would be best to check that the reward token is not `address(0)` and not `address(this)` in the `notifyRewardAmount` function.

Proof of Concept:

```
function test_notifyRewardAmountVaultTokens() public {

    // Alice deposits after failed reward
    vm.prank(alice);
    vault.deposit(10e18, alice);

    // Verify no rewards from before deposit
    assertEq(vault.previewRewards(alice, address(wbera)), 0, "Should have no rewards from before deposit");

    // New reward should work
    vm.startPrank(admin);
    vault.deposit(10e18, admin);
    vault.setRewardsDuration(address(vault), 7 days);
    vault.approve(address(vault), type(uint256).max);
    vm.stopPrank();
    notifyAndWarp(address(vault), 10e18);

    assertApproxEqAbs(vault.previewRewards(alice, address(vault)), 5e18, tolerance, "Should receive new
    ↪  rewards");
    assertApproxEqAbs(vault.previewRewards(admin, address(vault)), 0, tolerance, "Admin should receive new
    ↪  rewards");
    assertEq(vault.balanceOf(admin), 0e18, "Admin should have deposited all 10e18");

    // Record balance before claim
    uint256 balanceBefore = vault.balanceOf(alice);

    // Claim rewards
    vm.prank(alice);
    vault.claimRewards(address(alice));

    // Verify reward received
    assertApproxEqAbs(vault.balanceOf(alice) - balanceBefore, 5e18, tolerance, "Should receive full reward");
    assertEq(vault.previewRewards(alice, address(vault)), 0, "Rewards should be zero after claim");
}
```

HoneyJar: Fixed in commit 3aeaa6c3.

Cantina Managed: Fixed by adding a check that prevents the issue.

3.1.2 Deposits and Mints to Whitelisted Vaults Result in Lost Rewards

Severity: Medium Risk

Context: fatBERA.sol#L323-L343, fatBERA.sol#L352-L359, fatBERA.sol#L368-L376

Description: The contract currently allows minting and depositing directly to a receiver address. Vaults that are whitelisted have an **effective balance of zero**, meaning that any shares deposited or minted to them will not accrue any rewards.

This results in permanently lost rewards for any shares held by such vaults if they are set as the receivers. From a logical perspective, allowing deposits and mints to whitelisted vaults seems problematic, as it creates a scenario where shares exist but do not contribute to the reward calculations. If there is no business reason to support this, it should likely be restricted.

Recommendation: Deposits and mints should be restricted only to receivers that are explicitly non-whitelisted vaults to prevent shares from being rendered ineffective for rewards. Before processing a `deposit` or `mint`, the contract should check if `isWhitelistedVault(receiver)` is `true` if the receiver is a vault. If there is a valid business reason to allow deposits to whitelisted vaults, documentation should clarify why, and the implications should be well understood.

HoneyJar: Fixed in commit bab55d54.

Cantina Managed: Fixed. The code is now checking for `isWhitelistedVault` in `depositNative`, `deposit` and `mint` functions to prevent the un-sync of the `vaultedShares` mapping and the expected way of vault transfer.

3.1.3 Whitelisted Vaults Not Receiving Accumulated Rewards

Severity: Medium Risk

Context: fatBERA.sol#L262-L264

Description: When a vault is initially not whitelisted and later becomes whitelisted using `setWhitelistedVault`, any rewards that should have accrued to the vault during the period before whitelisting are lost. The vault does not receive any retroactive rewards for the time it was accumulating shares, but before it was officially marked as whitelisted.

This occurs because rewards are only tracked for effective balances, and whitelisted vaults have an effective balance of `0`, meaning no rewards are recorded for them. However, when a vault is not whitelisted, its shares are considered for rewards, but the rewards are not properly accounted for when transitioning the vault to a whitelisted state.

The provided proof of concept demonstrates that after transferring shares to an unwhitelisted vault and then calling `notifyRewardAmount`, the vault does not accrue rewards. Even after advancing time and whitelisting the vault, it still does not receive rewards for the period before whitelisting.

Proof of Concept:

```
function test_WhitelistingVaultAfterAccumulatingRewards() public {
    address vaultAddress = makeAddr("vault");

    // Alice deposits 100 WBERA
    uint256 depositAmount = 100e18;
    vm.prank(alice);
    vault.deposit(depositAmount, alice);

    // Alice transfers shares to an unwhitelisted vault
    uint256 transferAmount = 50e18;
    vm.prank(alice);
    vault.transfer(vaultAddress, transferAmount);

    // Notify rewards so that the vault accrues some rewards
    uint256 rewardAmount = 20e18;
    vm.prank(admin);
    vault.notifyRewardAmount(address(wbera), rewardAmount);

    // Capture the rewards after the notification and warp
    uint256 rewardsBeforeWhitelistingAndWarp = vault.previewRewards(vaultAddress, address(wbera));
    console2.log("rewardsBeforeWhitelistingAndWarp", rewardsBeforeWhitelistingAndWarp);

    assertEq(rewardsBeforeWhitelistingAndWarp, 0, "Vault should not accrue rewards before whitelisting");

    // Warp time forward to let rewards accumulate
    uint256 warpTime = 3 days;
    vm.warp(block.timestamp + warpTime);

    // Capture rewards before whitelisting
    uint256 rewardsBeforeWhitelisting = vault.previewRewards(vaultAddress, address(wbera));

    console2.log("rewardsBeforeWhitelisting", rewardsBeforeWhitelisting);

    // Whitelist the vault
    vm.prank(admin);
    vault.setWhitelistedVault(vaultAddress, true);

    // Warp more time to check if vault earns new rewards
```

```
    vm.warp(block.timestamp + warpTime);

    // Capture rewards after whitelisting
    uint256 rewardsAfterWhitelisting = vault.previewRewards(vaultAddress, address(wbera));

    console2.log("rewardsAfterWhitelisting", rewardsAfterWhitelisting);

    // Verify that the amount of rewards is at least for the period between the notify and whitelisting
    assertEq(rewardsAfterWhitelisting, rewardsBeforeWhitelisting, "Vault should not accrue new rewards after
    ↪  being whitelisted");
}
```

Recommendation: To ensure vaults do not lose accrued rewards when transitioning from a non-whitelisted state to a whitelisted one, `setWhitelistedVault` should first update rewards for the vault before modifying its whitelisted status. This ensures that any pending rewards are correctly accounted for before setting the vault's effective balance to zero.

Implementing an `_updateRewards(vaultAddress)` call before changing the `isWhitelistedVault` status ensures the vault receives rewards accrued before being whitelisted.

```
function setWhitelistedVault(address vaultAddress, bool status) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (!isWhitelistedVault[vaultAddress]){
        _updateRewards(vaultAddress);
    }
    isWhitelistedVault[vaultAddress] = status;
}
```

HoneyJar: Fixed in commit 6a577737.

Cantina Managed: Fixed as recommended.

3.2 Low Risk

3.2.1 Misleading Values on Preview Functions When Withdrawals Are Disabled

Severity: Low Risk

Context: *(No context files were provided by the reviewers)*

Description: The contract overrides `maxWithdraw` and `maxRedeem` to always return `0`, effectively disabling withdrawals and redemptions. However, `previewWithdraw` and `previewRedeem` are not overridden and will still return non-zero values based on share calculations. This can create integration issues for external contracts or frontends that rely on `previewWithdraw` and `previewRedeem` to determine whether withdrawals are possible.

A user or contract querying `previewWithdraw` or `previewRedeem` may see a non-zero value and assume that withdrawals are allowed, only to have transactions fail because `maxWithdraw` and `maxRedeem` enforce a `0` limit. This creates an inconsistent user experience and could lead to unintended behavior in automated integrations.

Recommendation: To ensure consistency with the withdrawal disabling mechanism, `previewWithdraw` and `previewRedeem` should be overridden to return `0` while withdrawals remain disabled. This ensures that any contract or interface querying these functions receives an accurate representation of withdrawal availability. When withdrawals are eventually enabled, the functions can be updated to return proper values again.

HoneyJar: Fixed in commit b033322d.

Cantina Managed: Fixed, the `previewWithdraw` and `previewRedeem` are overridden to return 0 when the contract is paused. The pausing state does only affect the `withdraw` and `redeem` functions.

3.2.2 `setMaxDeposits` Can Be Front-Run and Prevent Proper Deposit Management

Severity: Low Risk

Context: fatBERA.sol#L179-L182

Description: The function `setMaxDeposits` allows the admin to set a new maximum deposit cap, but it permits `newMax` to be exactly equal to `depositPrincipal`. This means that if an admin or governance

entity attempts to set `maxDeposits` to match the current principal, it effectively pauses further deposits. However, because deposits can be made by users at any time, there is a risk that someone front-runs the admin's transaction by making a deposit just before `setMaxDeposits` is executed. This would cause the new `maxDeposits` value to be lower than `depositPrincipal`, triggering a revert and preventing the intended deposit cap adjustment.

If the goal is to allow the admin to stop deposits by setting `maxDeposits` to `depositPrincipal`, this logic should be handled in a way that ensures the transaction does not fail due to front-running deposits.

Recommendation: An approach would be to automatically set `maxDeposits` to `depositPrincipal` if `newMax` is provided as a lower value, ensuring that deposits are halted instead of reverting the admin's transaction. Either approach would allow governance to adjust deposit caps reliably without the risk of front-running disruptions.

HoneyJar: Fixed in commit ab9a3acf.

Cantina Managed: Fixed, the function does now check if `newMax` is less than `depositPrincipal`, `maxDeposits` is set to `depositPrincipal` to halt deposits.

3.2.3 Principal Withdrawal Can Cause Underflow and Withdrawal Failures

Severity: Low Risk

Context: fatBERA.sol#L190-L196

Description: The function `withdrawPrincipal` allows the admin to withdraw principal assets from the contract, reducing the value of `depositPrincipal`. However, once withdrawals and redeems are enabled, this functionality can introduce significant issues. If users attempt to withdraw their deposits while the contract's deposit principal has been reduced through admin withdrawals, an underflow may occur when trying to deduct the withdrawal amount from `depositPrincipal`. This would cause the transaction to revert.

Even before an underflow occurs, another failure mode is possible: the contract may simply not have enough of the underlying asset available to transfer to withdrawing users. Since withdrawals rely on `safeTransfer`, any missing balance in the contract due to prior principal withdrawals would cause withdrawal transactions to fail. Users expecting to be able to redeem their shares may find that liquidity is missing, leading to a non-functional withdrawal mechanism.

Recommendation: The contract should ensure that `withdrawPrincipal` does not allow the admin to withdraw funds that are needed for future withdrawal requests. Implementing a minimum reserve check would prevent principal withdrawals from depleting the assets required to satisfy user withdrawals. A function to compute the maximum safe principal withdrawal based on current deposits and expected withdrawal demand would also mitigate this issue. If `withdrawPrincipal` is only meant for exceptional circumstances, additional restrictions should be put in place to clarify its intended use and prevent liquidity depletion.

HoneyJar: Withdrawing will most likely call a withdraw on the validator itself instead of transfer from the fatBERA contract. The current withdraw functions were there for testing. I should have just had them do nothing to not confuse you guys. We will be completely re writing the withdraw and redeem functions when they are enabled we just dont know the withdraw patterns yet. Withdraw principal will be called frequently by our validator operator so that they can deposit it into one of the validators we are running.

Cantina Managed: Acknowledged.

3.2.4 `_updateReward` calculation could lead to reward inflation allowing a user to steal rewards

Severity: Low Risk

Context: fatBERA.sol#L562-L566, fatBERA.sol#L545-L548

Description: Because the calculation to figure out how much `additional` reward to assign to the token's `rewardPerShareStored` assumes `totalSupply` is 1e18 or greater, any values below `1e18` will cause the `additional` reward to inflate. This `rewardPershareStored` is then used to calculate the `userRewardPerSharePaid` when `_updateRewards` is invoked. If a user has enough of the `totalSupply` they can manipulate it in a single block, this could lead to them inflating their rewards. It appears that this cannot currently happen, but is a serious consideration when enabling `withdraw` and `redeem`.

Likely the solution to this is to implement some sort of minimum dust for the deposits and not allow someone to withdraw/redeem if they are going to leave dust amounts behind. Another option is to have the owner "burn" some amount of initial wBERA when initializing the contract to prevent `totalSupply` from dropping below a certain amount.

Proof of Concept: The following proof of concept shows the basic math inflation:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.23;

import {Test} from "forge-std/Test.sol";
import {FixedPointMathLib} from "solady/utils/FixedPointMathLib.sol";
import {console2} from "forge-std/console2.sol";

contract RewardMathTest is Test {
    function setUp() public {}

    function test_rewardCalculations() public {
        // Common values for both scenarios
        uint256 elapsed = 1 days;         // 1 day in seconds
        uint256 rewardRate = 1e18;        // 1 token per second
        uint256 scale = 1e36;             // Standard scaling factor

        // Scenario 1: Normal total supply (10e18)
        uint256 totalSupply1 = 10e18;     // 10 tokens
        uint256 additional1 = FixedPointMathLib.fullMulDiv(
            elapsed * rewardRate,         // 86400 * 1e18
            scale,                        // 1e36
            totalSupply1                  // 10e18
        );

        // Scenario 2: Minimal total supply (1 wei)
        uint256 totalSupply2 = 1;         // 1 wei
        uint256 additional2 = FixedPointMathLib.fullMulDiv(
            elapsed * rewardRate,         // 86400 * 1e18
            scale,                        // 1e36
            totalSupply2                  // 1
        );

        console2.log("Scenario 1 (10e18 total supply):");
        console2.log("Additional rewards per share:", additional1);

        console2.log("\nScenario 2 (1 wei total supply):");
        console2.log("Additional rewards per share:", additional2);

        console2.log("\nRatio between scenarios (2/1):", additional2 / additional1);
    }
}
```

HoneyJar: We will fix all withdraw and redeem issues when we allow withdrawing and redemption. Also our deployment script already deposits 1 WBERA immediately which we will burn to help mitigate.

Cantina Managed: Yes, with the deposit of 1 WBERA, this resolves the issue currently. The `proxy` could theoretically remove this in the future triggering the problem. However, since withdraw/redeem are not enabled, ensuring the auditing team that reviews that implementation checks this issue should be sufficient.

3.3 Gas Optimization

3.3.1 `claimRewards(address)` contains duplicate code and could be simplified

Severity: Gas Optimization

Context: fatBERA.sol#L384-L409

Description/Recommendation: Since `claimRewards(address)` does the same thing as `claimRewards(address,address)` just for every token and the token specific function is already `public`, duplicate code could be removed by refactoring:

```
function claimRewards(address token, address receiver) public nonReentrant {
    _updateRewards(msg.sender, token);

    uint256 reward = rewards[token][msg.sender];
    if (reward > 0) {
        rewards[token][msg.sender] = 0;
        IERC20(token).safeTransfer(receiver, reward);
    }
}

/**
 * @notice Claims accrued rewards for all reward tokens.
 * @param receiver The address receiving the claimed rewards.
 * @dev Iterates through all reward tokens, updates rewards, and transfers available rewards.
 */
function claimRewards(address receiver) public nonReentrant {
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        claimRewards(rewardTokens[i], receiver);
    }
}
```

HoneyJar: Fixed in commit 5678cdc6. Had to remove the `nonReentrant` modifier from the all reward tokens one.

Cantina Managed: Fix verified.

3.3.2 Inconsistent errors

Severity: Gas Optimization

Context: fatBERA.sol#L223, fatBERA.sol#L251-L252

Description/Recommendation: For consistency, you should use custom errors instead of `require(..., "string error")` in the lines highlighted by the context of the present issue.

3.3.3 Unnecessary storage reads in `_lastTimeRewardApplicable`

Severity: Gas Optimization

Context: fatBERA.sol#L558-L575

Description/Recommendation: Since `_lastTimeRewardApplicable` is only used in `_updateReward` and they both read `rewardData[token]`, it would make more sense to either remove this function and inline the timestamp check or pass the `periodFinish` so the function becomes:

```
function _lastTimeRewardApplicable(uint256 periodFinish) internal view returns (uint256) {
    return block.timestamp < periodFinish ? block.timestamp : periodFinish;
}
```

HoneyJar: Fixed in commit e9f1b1dc.

Cantina Managed: Fix verified.

3.3.4 Unnecessary amount checks

Severity: Gas Optimization

Context: fatBERA.sol#L191, fatBERA.sol#L205

Description/Recommendation: Since the parameters are `uint256` it is not possible to pass a negative number. The affected lines can check equivalency to 0 instead.

HoneyJar: Fixed in commit 1a68aca6.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Separation of Pause and Unpause Permissions

Severity: Informational

Context: fatBERA.sol#L134-L144

Description: The contract currently requires the `DEFAULT_ADMIN_ROLE` to call both `pause` and `unpause`. While this approach centralizes control, it is often preferable to separate these permissions to avoid potential misuse or delays in emergency situations. In most best-practice implementations, a distinct **Pauser role** is assigned the ability to **pause** the contract quickly in case of unexpected issues or security threats, while only a higher-privileged **Admin role** retains the ability to **unpause** and resume normal operations.

By using a single admin role for both actions, the contract introduces a potential governance risk where only high-privilege accounts can respond to urgent issues, potentially delaying mitigation. Conversely, if a compromised admin account has the ability to pause and unpause at will, it could be used to manipulate contract operations maliciously.

Recommendation: Introduce a dedicated `PAUSER_ROLE` that is authorized only for pausing the contract, while retaining the `DEFAULT_ADMIN_ROLE` for unpausing. This separation ensures that emergency responders can react quickly to halt operations without granting them the ability to resume them arbitrarily. Additionally, if governance mechanisms are involved, pausing should be executable with minimal overhead while unpausing may require a more thorough review process.

HoneyJar: Fixed in commit 674df9ac.

Cantina Managed: Fixed, a new `PAUSER_ROLE` role was added to the `pause` function, the unpause is left to the admin only.

3.4.2 Implemented, but not ready functions

Severity: Informational

Context: fatBERA.sol#L411-L452

Description: The HoneyJar team stated that:

> withdrawal and redemption will not be enabled for months and only after an upgrade and re audit of those functions.

However, these functions appear to have code that would run if the contract was unpaused. This is not the case, though, even if the contract is unpaused, the `maxWithdraw` amount is set to `0` so it would still be impossible to withdraw or redeem if the contract is unpaused.

Recommendation: It seems it would be better to just override `withdraw` and `redeem` with noop functions until they are ready to be implemented. Further, testing can be improved by `vm.expectRevert("...Error expected...")` to ensure the reversions you are testing are the ones expected.

HoneyJar: An update to `maxWithdraw` and `maxRedeem` will also be included in the future upgrade as we aren't sure what those are going to be yet it depends on how validator withdrawals work.

Cantina Managed: Acknowledged.