GUARDIAN

Bracket Rd.2

LST Vault

Security Assessment

March 22nd, 2025

Summary

Audit Firm Guardian

Prepared By Roberto Reigada, Zdravko Hristov, Kiki,
Nicholas Chew, 0xCiphky, Vladimir Zotov

Client Firm Bracket

Final Report Date March 22, 2025

Audit Summary

Bracket engaged Guardian to do a 2nd review of their LST management system. From the 19th of February to the 25th of February, a team of 6 auditors reviewed the source code in scope. All findings have been recorded in the following report.

For a detailed understanding of risk severity, source code vulnerability, and potential attack vectors, refer to the complete audit report below.

✅ Verify the authenticity of this report on Guardian's GitHub: https://github.com/guardianaudits

📊 Code coverage & PoC test suite: https://github.com/GuardianOrg/bracketfi-fuzz

Table of Contents

Project Overview

Project Summary

Project Name	Bracket (Rd.2)
Language	Solidity
Codebase	https://github.com/bracket-fi/core-contracts
Commit(s)	Initial commit: 5a39d21032ae6afa7613b18f5cf2de11d0ce34d1 Final commit: c0636a03652e7d6a24a484c0fcf6df73bf4e9987

Audit Summary

Delivery Date	March 22, 2025
Audit Methodology	Static Analysis, Manual Review, Test Suite, Contract Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
● Critical	0	0	0	0	0	0
● High	2	0	0	0	0	2
● Medium	1	0	0	0	0	1
● Low	23	0	0	14	0	9

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: *High*	Impact: *Medium*	Impact: *Low*
Likelihood: *High*	🔴 Critical	🟠 High	🟡 Medium
Likelihood: *Medium*	🟠 High	🟡 Medium	🟢 Low
Likelihood: *Low*	🟡 Medium	🟢 Low	🟢 Low

Impact

High	Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.
Medium	A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.
Low	Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

High	The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.
Medium	An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.
Low	Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Invariants Assessed

During Guardian's review of Bracket contracts, fuzz-testing with Echidna was performed on the protocol's main functionalities. Given the dynamic interactions and the potential for unforeseen edge cases in the protocol, fuzz-testing was imperative to verify the integrity of several system invariants.

Throughout the engagement the following invariants were assessed for a total of 20,000,000+ runs with a prepared Echidna fuzzing suite.

ID	Description	Tested	Passed	Remediation	Run Count
GLOB_01	Contract balance should never be less than total accrued fees	✅	✅	✅	20M+
GLOB_02	If epoch = 0 then totalNonMintedShares = 0	✅	✅	✅	20M+
GLOB_03	If epoch = 0 then totalMintedShares = 0	✅	✅	✅	20M+
GLOB_04	If epoch = 0 then totalShares = 0	✅	✅	✅	20M+
GLOB_05	If epoch = 0 then activeSupply = 0	✅	✅	✅	20M+
GLOB_06	If epoch = 0 then inactiveSupply = totalDeposits	✅	✅	✅	20M+
GLOB_07	Token balance should never be less than inactiveSupply	✅	✅	✅	20M+
GLOB_08	totalShares should equal totalMintedShares + totalNonMintedShares	✅	✅	✅	20M+
GLOB_09	If epoch = 0 then the vault is always unlocked	✅	✅	✅	20M+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
GLOB_10	After startVault() the vault should be immediately unlocked	✅	✅	✅	20M+
GLOB_11	If lastDeposit[user].epoch = epoch, lastDeposit[user].pendingAssets after calling deposit() or withdraw() should be updated	✅	✅	✅	20M+
GLOB_12	Withdrawals must never be executed when the vault is locked	✅	✅	✅	20M+
ERR_01	Unexpected error	✅	✅	❌	20M+

Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	Incorrect Calculation Of totalFees In updateNav	Logical Error	● High	Resolved
H-02	Incorrect Subtraction Of totalFees	Logical Error	● High	Resolved
M-01	Front-Running Risk In startVault Function	Validation	● Medium	Resolved
L-01	Missing Input Validation In deployVault Function	Code Best Practices	● Low	Acknowledged
L-02	Inefficient Removal Logic In removeCollateral	Gas Optimization	● Low	Acknowledged
L-03	Variable Duration Of Locked And Unlocked Periods	Code Best Practices	● Low	Acknowledged
L-04	Incompatibility With Fee-On-Transfer Tokens	Code Best Practices	● Low	Acknowledged
L-05	Potential Reentrancy In claimBrktTvlFees And claimManagerFees	Code Best Practices	● Low	Acknowledged
L-06	Undesirable Delay When Manager Has Insufficient Funds	Logical Error	● Low	Acknowledged
L-07	Zero Share Withdrawals Possible	Code Best Practices	● Low	Resolved
L-08	Unnecessary Check And Casting	Gas Optimization	● Low	Resolved
L-09	Missing Validation In setNextLock	Validation	● Low	Resolved
L-10	_clearDeposit Return Values Never Used	Gas Optimization	● Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
L-11	Mismatch Between Documentation And Code	Code Best Practices	● Low	Resolved
L-12	Vault Lacking Pause Mechanism	Code Best Practices	● Low	Resolved
L-13	Multisig Admin Role Can Be Maliciously Revoked	Logical Error	● Low	Resolved
L-14	Asset Imbalance Through Cross-Token Actions	Validation	● Low	Acknowledged
L-15	Incomplete NatSpec	Code Best Practices	● Low	Resolved
L-16	Wrong Event Emission	Code Best Practices	● Low	Resolved
L-17	Users Can Claim Withdrawals When The Contract Is Paused	Code Best Practices	● Low	Acknowledged
L-18	Slight totalSupply() Discrepancy	Code Best Practices	● Low	Acknowledged
L-19	The nextLock Variable Is Not Properly Updated	Logical Error	● Low	Acknowledged
L-20	Changing Multisig Won't Affect The Vault	Code Best Practices	● Low	Acknowledged
L-21	claimWithdrawal Overestimates Available Balance	Validation	● Low	Acknowledged
L-22	Oracle Does Not Validate Fetched Price	Validation	● Low	Acknowledged
L-23	Inactive Balance Doesn't Include Claimable Withdrawals	Logical Error	● Low	Acknowledged

H-01 | Incorrect Calculation Of totalFees In updateNav

Category	Severity	Location	Status
Logical Error	● High	BracketVault.sol: 133	Resolved

Description

The totalFees variable is intended to represent the total fees affecting the asset flow between the vault and the manager during an NAV update. However, the calculation is inconsistent:

• accruedManagerPerformanceFees and accruedManagerTvlFees are cumulative (including fees from all previous epochs plus the current epoch's managerPerformanceFee and managerTvlFee).

• brktTvlFee is the new fee for the current epoch only, not the cumulative accruedBrktTvlFees.

This inconsistency leads to an incorrect totalFees value. Since totalFees is used in totalDebit = withdrawalAssets + totalFees within _processDepositsWithdrawals, it determines how much the manager must transfer to the vault (or receive from it). Using cumulative fees for manager fees is incorrect.

Expected Behavior: As the fees are now "locked on the vault's balance" the vault should retain all accrued fees and the manager should only need to cover the net asset flow (withdrawalAssets - depositAssets) plus the new fees for the current epoch. The previously accrued fees are already in the vault and can be used to pay withdrawals or claimed fees.

Example:

• Epoch 1: brktTvlFee = 5, so accruedBrktTvlFees = 5.

• Epoch 2: brktTvlFee = 3, so accruedBrktTvlFees = 8. Current code computes totalFees = accruedManagerFees + accruedManagerTvlFees + 3, ignoring the prior 5 in accruedBrktTvlFees. It should use only new fees (managerPerformanceFee + managerTvlFee + brktTvlFee).

Recommendation

Use only the new fees for consistency and logical correctness:

```
uint256 totalFees = managerPerformanceFee + managerTvlFee + brktTvlFee;
```

This ensures the manager transfers funds to cover withdrawals and new fees, while the vault retains previously accrued fees, aligning with the locking mechanism.

Resolution

Bracket Team: The issue was resolved in commit 4d9c92f.

H-02 | Incorrect Subtraction Of totalFees

Category	Severity	Location	Status
Logical Error	● High	BracketVault.sol: 493	Resolved

Description

The _getManagerAvailableBalance function calculates the manager's available balance for transferring funds to the vault when depositAssets < totalDebit. It subtracts totalFees (all accrued fees) from the minimum of the manager's balance and allowance.

However, this is illogical because:

• The accrued fees are locked in the vault's balance, not the manager's. The manager's token.balanceOf(manager) does not include these fees.

• Subtracting totalFees underestimates the manager's ability to transfer funds, potentially causing unnecessary reverts with InsufficientManagerFunds.

The manager's available balance should be the full amount it can transfer to the vault, limited only by its balance and allowance. Since fees are held in the vault and claimed separately (claimManagerFees, claimBrktTvlFees), they do not reduce the manager's available balance.

Example:

• Manager balance = 100, allowance = 100, accruedFees = 10 in the vault.

• Current code: available = min(100, 100) - 10 = 90.

• Actual: Manager can transfer up to 100, as the 10 in fees is in the vault.

Recommendation

Remove the fee subtraction from the _getManagerAvailableBalance function. This accurately reflects the manager's capacity to support the vault.

Resolution

Bracket Team: The issue was resolved in commit a5c9e2f.

M-01 | Front-Running Risk In startVault Function

Category	Severity	Location	Status
Validation	● Medium	BracketVault.sol: 100	Resolved

Description

The startVault function in the BracketVault contract is susceptible to front-running. This function is designed to move the vault from its initial state (Epoch 0) to an active state (Epoch 1), transferring all deposited assets to the manager to kick off operations.

During Epoch 0, users can deposit tokens and withdraw them instantly via the _withdrawEpoch0 function, which does not impose delays or penalties.

An attacker can exploit this setup by depositing a large quantity of tokens right before the startVault transaction, ensuring their deposit is recorded, and then withdrawing those tokens using _withdrawEpoch0 front-running the startVaultcall.

As a result, when startVault executes, it transfers only the remaining deposits, excluding the attacker's withdrawn amount, to the manager. This reduces the capital available to the manager, disrupting the vault's intended starting liquidity.

Recommendation

Consider updating the startVault function to include an additional parameter, minTotalDeposits, which specifies the minimum amount of deposits required to proceed with starting the vault. The function should check the current total deposits against this threshold and revert if the amount is insufficient.

This ensures that the vault only transitions to Epoch 1 when a predefined level of funding is secured, limiting the impact of last second withdrawals.

Resolution

Bracket Team: The issue was resolved in commit 8c1e877.

L-01 | Missing Input Validation In deployVault Function

Category	Severity	Location	Status
Code Best Practices	● Low	VaultFactory.sol: 32	Acknowledged

Description

The deployVault function accepts name, symbol, and withdrawDelay parameters but lacks validation for reasonable ranges or formats. For example, an excessively long name or symbol could cause issues in downstream systems.

Recommendation

Add input validation for name and symbol (e.g., maximum length).

Resolution

Bracket Team: Acknowledged.

L-02 | Inefficient Removal Logic In removeCollateral

Category	Severity	Location	Status
Gas Optimization	● Low	BrktETH.sol: 143	Acknowledged

Description

Inside the `removeCollateral` function, the code always does a "swap and pop" technique to remove the collateral from the array. However, if the item being removed (token) is already the last element in the collaterals array (`index = collaterals.length - 1`), the swap is unnecessary: you can simply `pop()`. The swap in that scenario wastes gas and can be avoided.

Recommendation

Add a condition to check whether index is already the last element. If so, simply `pop()` the last element without performing the swap. Otherwise, do the usual "swap and pop" logic.

Resolution

Bracket Team: Acknowledged.

L-03 | Variable Duration Of Locked And Unlocked Periods

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol	Acknowledged

Description

In the BracketVault contract, each epoch aligns with a nominal 7-day cycle defined by the LOCK_FREQUENCY constant, featuring alternating unlocked and locked periods dictated by the nextLock timestamp.

The unlocked period begins when nextLock is set and lasts until block.timestamp exceeds nextLock, at which point the vault becomes locked as determined by the isVaultLocked function (block.timestamp > nextLock epoch = 0).

The locked period then persists until updateNav is called by an address with the NAV_UPDATER_ROLE, which advances nextLock by 7 days and transitions the vault into the next epoch's unlocked period.

However, the exact durations of these periods within the 7-day cycle are not fixed and depend heavily on when updateNav is executed relative to nextLock.

• If updateNav is called as soon as block.timestamp surpasses nextLock (e.g., at nextLock + 1 second), the unlocked period lasts nearly the full 7 days (e.g., 6 days, 23 hours, 59 minutes, and 59 seconds), and the locked period is minimal (e.g., 1 second), though it cannot be zero due to the onlyLocked modifier requiring block.timestamp > nextLock.

• If updateNav is delayed by 3 days after nextLock, the unlocked period shortens to approximately 4 days, and the locked period extends to 3 days.

• If updateNav is never called, the vault remains locked indefinitely beyond nextLock, halting critical operations such as withdrawals (restricted by the onlyUnlocked modifier) and delaying the processing of queued deposits.

This variability in period lengths introduces significant unpredictability into user interactions, such as deposit and withdrawal timing, and vault management, as the duration of access to key functions depends on the timing of updateNav calls by the NAV_UPDATER_ROLE.

Recommendation

Consider introducing a stricter scheduling mechanism for epoch transitions by enforcing a maximum locked period duration.

Resolution

Bracket Team: Acknowledged.

L-04 | Incompatibility With Fee-On-Transfer Tokens

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol	Acknowledged

Description

The BracketVault contract does not explicitly support fee-on-transfer tokens, which are ERC20 tokens that deduct a fee from the transferred amount during operations like transfer or transferFrom, resulting in the recipient receiving less than the specified amount.

In the current implementation, the contract uses SafeERC20.safeTransferFrom and SafeERC20.safeTransfer for asset movements in functions such as deposit, withdraw, startVault, updateNav and claimWithdrawal.

These operations assume that the full specified amount is transferred to the intended recipient (e.g., the vault, manager, or user). However, with fee-on-transfer tokens, the actual amount received is reduced by the fee (e.g., if a 1% fee is applied to a 100-token transfer, only 99 tokens arrive).

The contract does not account for this discrepancy, as it records the full assets amount in state variables like totalDeposits, totalQueuedDeposits or lastDeposit[user].assets without verifying the actual received balance.

For example, in the deposit function, token.safeTransferFrom(msg.sender, address(this), assets) is called and assets is directly added to totalDeposits or totalQueuedDeposits, but if a fee reduces the received amount, the vault's internal accounting overstates the assets held, leading to inconsistencies.

Recommendation

Merely informative, avoid using Fee-On-Transfer tokens.

Resolution

Bracket Team: Acknowledged.

L-05 | Potential Reentrancy In claimBrktTvlFees And claimManagerFees

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol: 174	Acknowledged

Description

The claimBrktTvlFees and claimManagerFees functions transfers tokens to msg.sender before updating the contract's internal accounting.

If a token with on-transfer hooks was used, it could invoke a callback (reentrancy) that call the respective function again, leading to multiple payouts before the state is reset.

In the current order, the manager could potentially re-enter and claim fees repeatedly if the token's transfer call trigger external code.

While this is unlikely with standard ERC20 tokens, it still poses a theoretical risk given the absence of reentrancy protections.

Recommendation

Consider applying the the Check-Effects-Interactions (CEI) pattern or add the nonReentrant modifier to the claimBrktTvlFees and claimManagerFees functions.

Resolution

Bracket Team: Acknowledged.

L-06 | Undesirable Delay When Manager Has Insufficient Funds

Category	Severity	Location	Status
Logical Error	● Low	BracketVault.sol: 469	Acknowledged

Description

The updateNav function is critical for updating the NAV and advancing the epoch. However, it currently reverts when withdrawals exceed deposits and the manager lacks sufficient funds.

Insufficient funds can occur if assets are time-locked in a yield strategy or being bridged across chains. However, this should not prevent updateNav from executing.

Failing to call updateNav can delay the start of a new epoch and potentially exceed nextLock, causing overlaps with subsequent epochs. This could disrupt accounting and lead to delays in deposits and withdrawals.

Since a mechanism for partial withdrawals already exists, updateNav should be allowed to proceed even if the manager temporarily lacks sufficient funds.

Recommendation

Modify updateNav to proceed even when the manager has insufficient funds to transfer to the vault

Resolution

Bracket Team: Acknowledged.

L-07 | Zero Share Withdrawals Possible

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol: 239	Resolved

Description

If a small `assets` value is passed into the `withdraw` function while nav is greater than `1e18`, the calculated number of shares to be burned may round down to zero.

As a result, `claimWithdraw` can still be called with zero shares, and the transaction will succeed. Although no direct risk to funds has been identified, this behavior is unexpected and should be prevented.

Recommendation

In the `withdraw` function, ensure that if the calculated shares amount is zero, the transaction reverts to prevent unintended withdrawals.

Resolution

Bracket Team: The issue was resolved in commit d54f629.

L-08 | Unnecessary Check And Casting

Category	Severity	Location	Status
Gas Optimization	● Low	BracketVault.sol: 484	Resolved

Description

In the latest commit, accruedManagerPerformanceFees was changed from an int256 to uint256. Therefore, it's no longer necessary to check that it is > 0 and re-cast it to a uint256.

Recommendation

Update the accruedManagerPerformanceFees function removing the redundant check and type cast.

Resolution

Bracket Team: The issue was resolved in commit a5c9e2f.

L-09 | Missing Validation In setNextLock

Category	Severity	Location	Status
Validation	● Low	BracketVault.sol: 184	Resolved

Description

The function `setNextLock` should check that the new `nextLock` is not less than the current timestamp. An erroneous update could disrupt vault accounting.

Recommendation

Validate that `_nextLock > block.timestamp`.

Resolution

Bracket Team: The issue was resolved in commit e8fa0d3.

L-10 | _clearDeposit Return Values Never Used

Category	Severity	Location	Status
Gas Optimization	● Low	BracketVault.sol: 373	Resolved

Description

The internal function `_clearDeposit` returns `bool, Deposit memory newLastDeposit`. However, these values are never used.

Recommendation

Consider removing the return values if there is no need for them.

Resolution

Bracket Team: The issue was resolved in commit dbe079c.

L-11 | Mismatch Between Documentation And Code

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol: 221	Resolved

Description

The protocol documentation states that “In the case of a fund closing, every user will be processed automatically for a full withdrawal.” However, the current code only allows users to process their own withdrawals.

Consequently, during a fund closing, the protocol will not be able to execute the intended automatic full withdrawal process for all users.

Recommendation

If the documentation is outdated, update it to reflect the actual behaviour of the code. Otherwise, modify the code to implement the automatic full withdrawal functionality as described in the documentation.

Resolution

Bracket Team: Resolved.

L-12 | Vault Lacking Pause Mechanism

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol	Resolved

Description

After a strategy vault is shut down, users may still unknowingly deposit into it. Since the vault is no longer active, their funds remain stuck until an admin manually calls `updateNav` to trigger withdrawals.

Additionally, without a mechanism to halt operations, users could continue interacting with the vault during an exploit

Recommendation

Consider adding a pausing mechanism and implement the admin functions `pause` and `unpause`.

Resolution

Bracket Team: The issue was resolved in commit a5b662a.

L-13 | Multisig Admin Role Can Be Maliciously Revoked

Category	Severity	Location	Status
Logical Error	● Low	BracketVault.sol: 86	Resolved

Description

In the `initialize` function, the contract grants `DEFAULT_ADMIN_ROLE` to the `multisig` parameter. However, there is no validation preventing `DEFAULT_ADMIN_ROLE` from being included in the `roles` array during initialization.

This means the vault creator could include `DEFAULT_ADMIN_ROLE` in the `roles` array and assign it to themselves or another address.

Since `DEFAULT_ADMIN_ROLE` is its own admin (as per `OpenZeppelin`'s `AccessControl`), any address with this role can revoke it from other addresses. This creates a vulnerability where:

1. Vault creator includes `DEFAULT_ADMIN_ROLE` in `roles` array during initialization

2. Both multisig and creator now have `DEFAULT_ADMIN_ROLE`

3. Creator can call `revokeRole(DEFAULT_ADMIN_ROLE, multisig)` to remove multisig's admin access

4. Creator now has sole control of admin functions

Recommendation

Add validation in the `initialize` function to prevent `DEFAULT_ADMIN_ROLE` from being included in the `roles` array.

Resolution

Bracket Team: The issue was resolved in commit d584be9.

L-14 | Asset Imbalance Through Cross-Token Actions

Category	Severity	Location	Status
Validation	● Low	BrktETH.sol: 83	Acknowledged

Description

The burn function allows users to withdraw a different collateral token than the one they initially deposited, which can lead to asset imbalance issues in the protocol. Here's how this can impact the system:

1. A user deposits Token A into the protocol using the mint function

2. Next, they call burn to withdraw Token B instead of Token A

3. This cross-token withdrawal can:

• Create imbalances in individual token reserves

• Potentially prevent swapCollateral operations due to insufficient balances

• Impact yield generation if certain assets become depleted and the current ratio between assets is far off from the optimal ratio that is typically maintained.

While the 5-day withdrawal delay (WITHDRAWAL_DELAY) provides some protection by giving time for rebalancing, the core issue remains that users can systematically drain specific collateral tokens, affecting the protocol's ability to maintain balanced reserves and optimal yield generation.

The impact is particularly concerning because:

• It affects the protocol's yield generation capabilities

• It can create scenarios where other users cannot withdraw their preferred tokens

• The swapCollateral function may fail due to insufficient balances

Recommendation

Consider implementing one of these solutions:

1. Require users to withdraw the same token type they deposited by tracking individual deposits

2. Add a fee mechanism for cross-token withdrawals to discourage imbalancing behavior

At minimum, document this behavior in the protocol specifications so users understand the associated risks.

Resolution

Bracket Team: Acknowledged.

L-15 | Incomplete NatSpec

Category	Severity	Location	Status
Code Best Practices	● Low	IBracketVault.sol	Resolved

Description

The `NatSpec` for the `IBracketVault.Deposit` struct lacks description of the `queuedAssets` parameter.

Recommendation

Describe the `queuedAssets` parameter as well.

Resolution

Bracket Team: The issue was resolved in commit fc72b46.

L-16 | Wrong Event Emission

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol	Resolved

Description

The data used for the `NavUpdated` event in `updateNav()` is the data for the updated epoch, but the `epoch` param is passed after it has been increased. This will lead to the event being emitted with `epoch n + 1` and the nav data for `epoch n`.

Recommendation

Consider correcting the `NavUpdated event.`

Resolution

Bracket Team: The issue was resolved in commit e24e55f.

L-17 | Users Can Claim Withdrawals When The Contract Is Paused

Category	Severity	Location	Status
Code Best Practices	● Low	BrktETH.sol	Acknowledged

Description

`BrktETH.claimWithdrawal()` can be called when the contract is paused.

Recommendation

Be aware of this behavior.

Resolution

Bracket Team: Acknowledged.

L-18 | Slight totalSupply() Discrepancy

Category	Severity	Location	Status
Code Best Practices	● Low	BracketVault.sol	Acknowledged

Description

totalPendingShares in BracketVault are increased with the total deposited assets converted to shares in _processDepositsWithdrawals. They are then reduced by the amount of shares being minted in _clearDeposit().

Because of rounding issues, the sum of the shares minted to each user may not be equal to the sum of the shares added in _processDepositsWithdrawals.

Because of that, BracketVault.totalNonMintedShares() will return slightly inflated value, therefore RebalancingToken.totalShares(), RebalancingToken.activeSupply() and RebalancingToken.totalSupply() as well.

Recommendation

Document this discrepancy so integrators can avoid issues related to it.

Resolution

Bracket Team: Acknowledged.

L-19 | The nextLock Variable Is Not Properly Updated

Category	Severity	Location	Status
Logical Error	● Low	BracketVault.sol	Acknowledged

Description

Whenever `updateNav()` is called, `LOCK_FREQUENCY` is being added to `nextLock` instead of setting `nextLock` to `block.timestamp + LOCK_FREQUENCY`.

This will result in new epoch being shorter than they should be, depending on the delay between the expiry of the previous lock and the update of the nav.

The update of the nav can be delayed due to several different factors - offchain calculation is not so straightforward, network congestion, the `updateNav()` reverting because of `InsufficientManagerFunds()`, etc…

Let's take a look at a simplified example:

• `nextLock` = 100

• `LOCK_DURATION` = 100

• `updateNav()` is called at 150

• `nextLock = 100 + 100 = 200`

• The new epoch will be locked after `LOCK_DURATION / 2`.

If the `updateNav()` was called after 200, the new epoch would have been instantly locked.

Recommendation

Update `nextLock` as follows:

```
nextLock = LOCK_FREQUENCY; + nextLock = block.timestamp + LOCK_FREQUENCY;
```

Resolution

Bracket Team: Acknowledged.

L-20 | Changing Multisig Won't Affect The Vault

Category	Severity	Location	Status
Code Best Practices	● Low	VaultFactory.sol	Acknowledged

Description

In the VaultFactory there is a setMultisig() function which allows the DEFAULT_ADMIN_ROLE to change the multisig. However, all the previously deployed vaults will still have the old multisig set.

Recommendation

Document this to avoid unexpected behaviors.

Resolution

Bracket Team: Acknowledged.

L-21 | claimWithdrawal Overestimates Available Balance

Category	Severity	Location	Status
Validation	● Low	BracketVault.sol: 259	Acknowledged

Description

In the BracketVault contract, the claimWithdrawal function checks the vault’s entire token balance via token.balanceOf(address(this)) to determine how much can be paid out for a user withdrawal.

This raw balance includes:

• Queued deposits: User funds transferred during locked epochs, recorded as totalQueuedDeposits, which are slated to be converted to shares in the next updateNav call.

• Unclaimed fees: Manager or bracket fees stored in the vault’s balance (accruedManagerPerformanceFees, accruedManagerTvlFees, accruedBrktTvlFees) that remain physically in the contract but are not intended to fund user withdrawals.

Because claimWithdrawal does not subtract these queued deposits or outstanding fees from the vault’s “available” funds, a user can withdraw more than the true liquid balance the vault should be able to commit.

Once updateNav processes queued deposits (or fees get claimed), the vault may find itself short of tokens to fulfill its obligations.

Recommendation

In claimWithdrawal, adjust the availability check so it excludes both totalQueuedDeposits and any unclaimed fees that should not fund user withdrawals.

Resolution

Bracket Team: Acknowledged.

L-22 | Oracle Does Not Validate Fetched Price

Category	Severity	Location	Status
Validation	● Low	BracketOracle.sol	Acknowledged

Description

In BracketOracle.sol, the getRate function retrieves price from each LST/LRT oracle but does not validate whether the fetched price is zero.

If an oracle returns a zero price, BrktETH's getTotalValue would calculate an incorrect total collateral value, potentially enabling opportunistic attacks that exploit the mispriced collateral.

Recommendation

For all functions used to fetch the LST/LRT price (e.g. _getWstethRate), validate that the fetched price is not zero.

Resolution

Bracket Team: Acknowledged.

L-23 | Inactive Balance Doesn't Include Claimable Withdrawals

Category	Severity	Location	Status
Logical Error	● Low	BracketVault.sol	Acknowledged

Description

`BracketVault.balanceOf(address)` aims to return the underlying token's value of the provided address parameter. It currently returns the `activeBalance + inactiveBalance`, where the active balance is the amount of minted shares + the amount of pending shares that we know the NAV for.

The inactive balance are the rest of the assets (deposited and queued) which we don't know the NAV for. This calculation doesn't include the current claimable assets that can be withdrawn. In result, the `balanceOf()` will report lower value in this cases.

Recommendation

Consider tracking individual withdrawals per epoch and account for them in the `balanceOf()` function.

Resolution

Bracket Team: Acknowledged.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian Audits

Founded in 2022 by DeFi experts, Guardian Audits is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian Audits upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits