QuillAudits

AUDIT REPORT

June , 2025

For

Table of Content

Table of Content 02

Executive Summary 05

Number of Issues per Severity 08

Checked Vulnerabilities 09

Techniques & Methods 11

Types of Severity 13

Types of Issues 14

High Severity Issues 15

1. Signature replay in both updateGlobalState() and getPoolData() 15

2. Cross chain signature replay in both updateGlobalState() and getPool-Data() 16

Medium Severity Issues 17

1. Incorrect selector calculations 17

2. No Validation for pool id existence for crosschain stake and raid 19

3. Using State Variables in Facets Can Cause Storage Collision 20

4. Incorrect initialization of function selectors 21

5. Possible denial of service while raiding the pool cross chain 22

6. denial of service (DoS) for unstaking buds 23

7. Note about updateGlobalState() 24

8. Possible problems of response verification in updateGlobalState() and getPoolData() functions 26

9. Denial of service while delegatecall-ing to getPoolData() 27

Low Severity Issues 28

1. finalizeRaid() always emits the event with isSuccess as false 28

2. Incorrect payload creation 29

3. endpoint() and peers() does not return expected values 30

4. Prevent farmer delegation changes to the same pool id as the old 31

5. Raid can revert in rare case when the updated lastRaidTs is greater than block.timestamp 32

6. Inconsistent implementation in crossChainRaid function compared to other functions 33

7. Missing Some Event Emission for Incoming calls made from source chains 34

8. Remove unused function from in the CrossChainManager Facet 35

9. The user can get the reward for more than 1 epoch in the rare case 36

10. Incorrect _rewardsPooled passed to LibRaiding.raidPool() 38

11. The contract should not allow staking and raiding the blacklisted pool 39

12. Add required state updates 40

13. Prevent farmer delegation changes to the pool id 0 41

14. The user should not be able to delegate the NFT to the blacklisted pool. 42

15. The user should not be able to raid the blacklisted pool. 43

16. Pools with 0 TDFs can be skipped while sending the reward. 44

17. Redundant functionality 45

Informational Severity Issues 46

1. Floating pragma 46

2. Incorrect comment 47

3. Hardcoded level requirements 48

4. Redundant import 49

5. Incorrect check as PoolId cannot be less than zero 50

6. The meaningful error message can be added 51

7. Contradictory comment 52

8. Fix the typo 53

9. Add events according to the requirement 54

Functional Tests 55

Closing Summary & Disclaimer 57

Executive Summary

Project Name	Bakeland
Project URL	https://x.com/bakelandxyz?lang=en
Overview	The Bakeland contract is a cross-chain DeFi protocol built on a Diamond upgradeable architecture, integrating LayerZero and Wormhole for seamless cross-chain operations. The system enables users to participate in staking pools, conduct cross-chain raids, and manage token delegations across multiple chains. At its core, the protocol implements a unique raiding mechanism where users can stake BUDS tokens in pools and participate in risk-based raiding activities. The system leverages LayerZero's OApp protocol for cross-chain messaging and Wormhole's query response system for secure state verification. This dual-bridge approach ensures reliable cross-chain communication while maintaining data integrity.
Audit Scope	The scope of this Audit was to analyze the Bakeland Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/bakestake/Bakeland-Pool-Factory.git
Branch	main
Contracts in Scope	contracts/Manager/Facets/DiamondFacets/DiamondCut.sol contracts/Manager/Facets/DiamondFacets/ DiamondLoupe.sol CrossChainManager.sol FarmerAlloc.sol GetterSetter.sol LevelManager.sol PoolFactory.sol RaidHandler.sol Router.sol StateUpdate.sol contracts/Manager/Initializer/Diamond.init.sol contracts/Manager/Initializer/Lz.init.sol LibDiamond.sol LibFarmerAlloc.sol LibGlobal.sol LibLevelManager.sol

	LibLz.sol LibPoolManager.sol LibRaiding.sol contracts/Manager/Diamond.sol contracts/Manager/interfaces:- IAsset.sol IBudsToken.sol IDiamondCut.sol IDiamondLoupe.sol IERC165.sol IERC173.sol IGauge.sol IStBuds.sol IStakingPool.sol ISupraRouter.sol IXp.sol contracts/Manager/Utils/lzSupport: OAppCoreUp.sol OAppReceiverUp.sol OAppSenderUp.sol OAppUp.sol
Commit Hash	af568ef25e621d8f85146f334ef7986e9ea05f73
Language	Solidity
Blockchain	Berachain
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	2nd April 2025 - 9th May 2025
Updated Code Received	2nd June 2025 to 24th June 2025
Review 2	2nd June 2025 to 24th June 2025
Fixed In	Branch: Audit Fixes 3883844e39329ab9f6ed91a71fef6767c78afaf9

Note

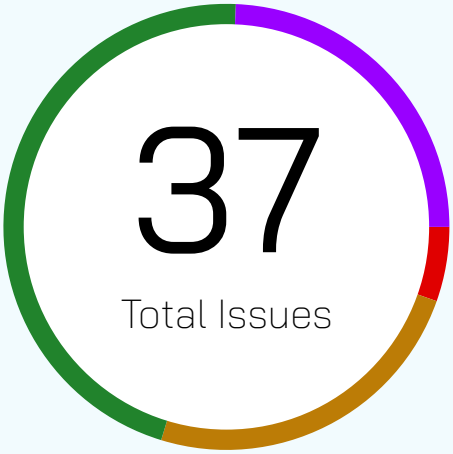
The QuillAudits team has completed an audit of the Bakeland codebase, as detailed in the executive summary. We identified and reported multiple issues, which the Bakeland team has since resolved. During this rectification process, significant changes were also made to the codebase.

Our final review focused exclusively on the implemented fixes. The other significant changes made to the codebase were outside the scope of this audit and have not been reviewed. We strongly recommend that the Bakeland team arranges for these changes to be audited in the near future, prior to deploying on the mainnet. Some of these unreviewed changes include:

1. raid
   - bonus
   - multiplier
     - affects raid fee, reward, and success probability
2. stake
   - stBuds mints on bera only
   - cross-chain call to mint on bera in case of staking on other chain
3. state update functions
   - added state update functions for narc ownership and pool existance check
   - added function for pool data query
4. farmer delegation
   - added weighthed farmer delegation
   - based on farmer level bonus amount is decided

Structural changes
   - removed Router, poolFactory facets
   - removed lz init initializer
   - stake, raid, unstake functions added to crossChainManager facet
   - Function from stateUpdate facet now called using interface

Number of Issues per Severity



High	2 (5.41%)
Medium	9 (24.32%)
Low	17 (45.95%)
Informational	9 (24.32%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**2**	**9**	**17**	**7**
Acknowledged	0	0	0	**2**
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ✔ Access Management
- ✔ Arbitrary write to storage
- ✔ Centralization of control
- ✔ Ether theft
- ✔ Improper or missing events
- ✔ Logical issues and flaws
- ✔ Arithmetic Computations Correctness
- ✔ Race conditions/front running
- ✔ SWC Registry
- ✔ Re-entrancy
- ✔ Timestamp Dependence
- ✔ Gas Limit and Loops
- ✔ Exception Disorder
- ✔ Gasless Send
- ✔ Use of tx.origin
- ✔ Malicious libraries
- ✔ Compiler version not fixed
- ✔ Address hardcoded
- ✔ Divide before multiply
- ✔ Integer overflow/underflow
- ✔ ERC's conformance
- ✔ Dangerous strict equalities
- ✔ Tautology or contradiction
- ✔ Return values of low-level calls
- ✔ Missing Zero Address Validation
- ✔ Private modifier
- ✔ Revert/require functions
- ✔ Multiple Sends
- ✔ Using suicide
- ✔ Using delegatecall
- ✔ Upgradeable safety
- ✔ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

Signature replay in both updateGlobalState() and getPoolData()

Resolved

Path

StateUpdate.sol

Function

updateGlobalState(), getPoolData()

Description

Replaying the signature in both updateGlobalState(), getPoolData() is possible if done within 5 minutes of response/signature creation.

The parseAndVerifyQueryResponse() returns r of type ParsedQueryResponse, which contains nonce, this nonce is not used in the function implementation to validate that this response uses a new nonce.

This should be noted that the nonce needs to be different for every wormhole query request. So that the nonce in the bytes response can be different, which can be used in the contract for nonce validation.

The impact of this issue can allow a malicious user to update the data according to the past data. Even if this attack works within the 5 minutes of response/signature creation, it should be noted that the actions/calls that increase protocol tvl can happen within these 5 minutes, which has the ability to impact the calculations for pool APR and saturation as these state updates will happen on this/current chains.

Recommendation

For this nonce validation, the contract should maintain uint InContractNonce variable in the contract, which should be checked against the r.nonce in the updateGlobalState()/getPoolData() function. the InContractNonce will be increased for every updateGlobalState()/getPoolData() call and should match the nonce of the current wormhole query request.
If r.nonce and the InContractNonce are not same, then the transaction should revert, assuming that the signature is getting reused.

Cross chain signature replay in both updateGlobal-State() and getPoolData()

Resolved

Path

StateUpdate.sol

Function

updateGlobalState(), getPoolData()

Description

Generally, the signature and response for the contract deployed on the Chain 2, for example, will be used on Chain 1.

In case there are another chain, Chain 4, where the contract is deployed and the response and signatures are generated for that chain to use on Chain 3, Then these responses and signatures can be used on chain 1 because there are no checks happening for the received response in both update-GlobalState() and getPoolData().

Recommendation

The r.responses[i].chainId should be used in the for{} loop of updateGlobalState()/getPoolData() to check if it matches the expected chain ID. For this, a variable like uint expectedChainId can be maintained in the contract storage (depending on the requirement, the setter method can be added to change it by an authorized account). This maintained expectedChainId will be compared with the r.responses[i].chainId to check that it's coming from the expected chain.

Medium Severity Issues

Incorrect selector calculations

Resolved

Path

LibGlobal.sol

Function

callStateUpdate(), updatePoolTvl()

Description

Function selector calculation in callStateUpdate() and updatePoolTvl() on L181 and L210, respectively, contains the parameter names and other incorrect representations for including IWormhole.Signature[] data type.

The effect of this would be that the function will make the delegatecall on the address(0) as there would be no function selector added through diamondCut() for the incorrect function selector. The delegatecall on the address(0) will return true as there would be no code on address(0). which will make the function pass without updating the state i.e, without calling actual updateGlobalState().

E.g. the accurate selector in the case of updateGlobalState() is 0x62f3c06d. This can be confirmed in the solidity by:

```
function updateGlobalState(address user, bytes memory response, IWormhole.Signature[] memory signatures) public {
 emit stateUpdate("updateGlobalState called!");
}

function getSelector() public pure returns(bytes4) {
 return this.updateGlobalState.selector;
}

function getSelectorWithKeccak() public pure returns (bytes4) {
return bytes4(keccak256("updateGlobalState(address,bytes,(bytes32,bytes32,uint8,uint8)[])"));
}
```

Here both getSelector() and getSelectorWithKeccack() will return 0x62f3c06d

Similarly, the selector for getPoolData() is incorrect and should be corrected.

Recommendation

Use bytes4(keccak256("updateGlobalState(address,bytes,(bytes32,bytes32,uint8,uint8)[])")) and bytes4(keccak256("getPoolData(uint32,uint256,bytes,(bytes32,bytes32,uint8,uint8)[])")) for function selector calculation respectively in callStateUpdate() and updatePoolTvl().

No Validation for pool id existence for crosschain stake and raid

Resolved

Path

CrossChainManager

Function

crossChainStake

Description

The crossChainStake() and crossChainRaid() function in the CrossChainManager contract lacks validation of the pool ID's existence before initiating a cross-chain stake operation. The source chain does not validate the existence of pool IDs before burning tokens and initiating cross-chain stakes. This can lead to permanent token loss for users.

Recommendation

Verify that a pool ID exists on the destination chain before source chain stakes operations are initiated.
Additionally, the transaction should revert if the pool ID entered on the source chain is 0.

Using State Variables in Facets Can Cause Storage Collision

Resolved

Path

StateUpdate, DiamondInit

Function

wormhole

Description

The use of state variables directly in facets can lead to storage collisions in the Diamond pattern implementation. This is because each facet's state variables are stored in the same storage space as the Diamond contract, and if not properly managed, they can overwrite each other's data.

Recommendation

The wormhole address declared to take a storage slot in the QueryResponse inherited by this contract should be declared as constant to avoid occupying a storage and causing a facet to own a storage.

Incorrect initialization of function selectors

Resolved

Path

Diamond.init.sol

Function

init()

Description

In DiamondInit.init(), LibGlobal.bytesStore().GetQueryDataSelector and LibGlobal.bytesStore().GetPoolTvlSelector are initialized incorrectly
In both strings getqueryData(address user) and getPoolData(uint256 poolId), the parameter name is used. Only the parameter data type should be used as a part of the string for creating the selector.

This can affect the signature verification in the StateUpdate.updateGlobalState().

Recommendation

Correct the selector calculation by removing the parameter names and only keeping the values.

Possible denial of service while raiding the pool cross chain

Resolved

Path

CrossChainManager.sol

Function

_lzReceive()

Description

No LibGlobal.preCheck() is called in the _lzReceive() when calling LibRaiding.raidPool() on dest chain. Generally, when the RaidHandler.raid() executes the LibGlobal.preCheck() called which increases the current epoch (if block.timestamp >= endTime), after that in LibRaiding.raidPool() it checks the getRaidStorage().lastRaidBoost[_raider][LibGlobal.getCurrentEpoch()] to check if the max boost is not reached yet.

Since the _lzReceive() is not calling LibGlobal.preCheck() before calling LibRaiding.raidPool(), it can happen that in case the user has used max boosts (4) in that epoch on dest chain even if the block.timestamp >= endTime is true, the epoch won't be increased, and the transaction will revert with the MaxBoostReached().

Recommendation

Call LibGlobal.preCheck() before calling LibRaiding.raidPool() in the _lzReceive().

denial of service (DoS) for unstaking buds

Resolved

Path

CrossChainManager.sol

Function

_onStake()

Description

_onStake() does not mint stBuds to the user as it normally does when the user stakes his buds with Router.addStake().

It creates other problems like the user won't be able to unstake, etc, because he doesn't own any stBuds to burn while unstaking with Router.unStakeBuds() on the destination stake.

Recommendation

Mint the _budsAmount amount of stBuds to the user in _onStake() as a share. Which can be used while unstaking.

Note about updateGlobalState()

Resolved

Path

StateUpdate.sol

Function

updateGlobalState()

Description

updateGlobalState() updates the LibGlobal.intStore().globalStakedBuds and LibGlobal.intStore().noOfPoolGlobal according to the data received from the crosschain through the query response provided.
since the LibGlobal.intStore().globalStakedBuds used in getSaturationFactor() and the LibGlobal.intStore().noOfPoolGlobal is used in getBaseApr(), updating it can affect the output of getCurrentApr() for the current chain.

It should be noted that the LibGlobal.intStore().noOfPoolGlobal is assigned a value of LibPoolManager.getNumberOfPools() received as a response of getqueryData() from cross chain. which returns getPoolStorage().poolSerialNumber value, which is increased when the pool is deployed in the PoolFactory. So it can happen that the number of pools deployed cross chain is less or more than what is on the current chain, so assigning the value directly doesn't make sense, additionally because there will be multiple updateGlobalState() calls happening as there will be multiple raids, directly assigning a value won't make sense even if the addition is used (i.e. adding the number of pools created on other chain to the LibGlobal.intStore().noOfPoolGlobal on this chain)

The values like LibRaiding.getRaidStorage().lastRaidTs[user], LibRaiding.getRaidStorage().streakByUser[user], LibGlobal.mappingStore().aggreagatedRecord[user].latestStakedTs, LibGlobal.mappingStore().aggreagatedRecord[user].stakeStartTs are updated directly according to the cross chain value received, it can happen that the value received is less or greater than what it is on this chain. Depending on the future logic, it can create confusion or problems.

E.g On this chain, the LibRaiding.getRaidStorage().streakByUser[user] is more than what it is on the cross chain, Now while assigning the value on this chain in updateGlobalState(), it can happen that the lesser value of streakByUser will be assigned, affecting some future or off-chain calculation logic.

Additionally, this can happen while aggregating the global count where the globalState.globalBudsCount += budsCount, globalState.globalPoolCount += poolCount,

globalState.globalStakersCount += stakersCount are updated according to the cross chain. But since the updateGlobalState() will be called multiple times, it won't make sense to everytime add the global counts on this chain.

Remediation

Verify the required things that should be updated according to the cross-chain.

Possible problems of response verification in updateGlobalState() and getPoolData() functions

Resolved

Path

StateUpdate.sol

Function

updateGlobalState(), getPoolData()

Description

The eqrRes[0].result.length validation needs to be changed to the valid length:

1. updateGlobalState() on L64 checks that the eqrRes[0].result.length should be 32. But since the updateGlobalState() expects the response as an output/returned values from GetterSetter.getqueryData() function, the data length will be greater than 32 bytes.

In the current case, the response's result length is 224. So It should be validated to be 224, not 32.

2. getPoolData() on L124 checks that the eqrRes[0].result.length should be 32. But since the getPoolData() expects that the response as an output/returned value from GetterSetter.getPoolData() function, the data length will be greater than 32 bytes.

In the current case, the response's result length is 192. So It should be validated to be 192, not 32.

validAddresses[0] needs to be changed in both functions:
Currently, the validAddresses[0] is set to address(this), This address is used in validateMultipleEthCallData() to check that the eqr.result is coming from the expected address.
This address can vary depending on the method of deployment. The correct address needs to be added here if the Diamond proxy address (i.e., address(this) in this case) on the other chain is not equal.

Because of this issue(s) the state update can fail.

Recommendation

Correct the length in the length validation check as suggested above. Consider checking the address on the other chain is equal to address(this) on this chain when verifying the query response.

Denial of service while delegatecall-ing to getPoolData()

Resolved

Path

StateUpdate.sol

Function

getPoolData()

Description

crossChainRaid() calls LibGlobal.updatePoolTvl() which makes delegatecall to StateUpdate.getPoolData().

The delegatecall to getPoolData() will revert because the crossChainRaid() is a payable function, as it is the entry point to take the LZ fees. When LibGlobal.updatePoolTvl() function makes the delegatecall to the getPoolData() the transaction will revert because the msg.value will be non-zero for the delegatecall to non payable getPoolData() function.

The impact of this issue is that the functionality becomes useless.

Recommendation

Make the getPoolData() payable so the msg.value can be non-zero while making delegatecall to it.

Low Severity Issues

finalizeRaid() always emits the event with isSuccess as false

Resolved

Path

LibRaiding.sol

Function

finalizeRaid()

Description

finalizeRaid() on L117 even though the sendRewards() is executed and emits the Raided() with the isSuccess value as true. But every time the control flow reaches the end of the function, on L121 it emits the Raided() with the isSuccess value as false.

It happens because it never returns and halts the function's execution with the return keyword after executing the code in the if{}.

The impact of this can vary depending on the usage of the event. But because of the false emission of the event, any service will get a false representation of the event.

Recommendation

Add a return statement after L118 in the if{} to avoid emitting the emit Raided() event again, even after sending rewards in the if{}.

Incorrect payload creation

Resolved

Path

CrossChainManager.sol

Function

getCctxFees()

Description

getCctxFees() calculates the native fee required for the payload. In this function the payload is created with abi.encode(msgtype, abi.encode(budsAmount, poolId, sender)).

Since, the contract has different functionalities, the payload creation for each one differs. E.g. The crossChainStake() uses abi.encode(LibLz.getLzStorage().CROSS_CHAIN_STAKE_MESSAGE, abi.encode(_budsAmount, msg.sender, poolId)).

The crossChainRaid() uses abi.encode(LibLz.getLzStorage().CROSS_CHAIN_RAID_MESSAGE, abi.encode(boosted, riskLevel, msg.sender, poolId))

The crossChainBudsTransfer() uses abi.encode(LibLz.getLzStorage().CROSS_CHAIN_BUDS_TRANSFER, abi.encode(_amount, msg.sender, _to))

crossChainPolStake uses abi.encode(LibLz.getLzStorage().CROSS_CHAIN_POL_STAKE, abi.encode(msg.sender, _amount)) as a payload.

Since the payload created in getCctxFees() for getting a quote won't everytime match the payload that is created in the other functions while sending it with _lzSend().
The endpoint.quote() won't be able to calculate the correct fee, as the payload created in the getCctxFees() is incorrect. So, getCctxFees() eventually can return an incorrect fee.

Recommendation

Consider fixing the getCctxFees() to calculate the payload similarly for every message type calculated in the specific functions.
Here if{} can be used to calculate it according to every message type.

endpoint() and peers() does not return expected values

Resolved

Path

CrossChainManager.sol

Function

endpoint(), peers()

Description

The endpoint() declares the ILayerZeroEndpointV2 iEndpoint, but no value is assigned to it. Resulting in returning the address(0) even when the endpoint is a non-zero address.

The peers() declares the bytes32 peer, but no value is assigned to it. Resulting in returning the bytes32(0) value.

Recommendation

Consider returning LibLz.getLzStorage().endpoint in the endpoint() and LibLz.getLzStorage().peers[_eid] in peers().

Prevent farmer delegation changes to the same pool id as the old

Resolved

Path

FarmerAlloc.sol

Function

changeDelegation()

Description

In the changeDelegation function of the FarmerAlloc contract, there is no validation to prevent a user from changing their delegation to the same pool they are already delegated to. This creates unnecessary gas consumption as users can waste gas by changing delegation to the same pool.

Recommendation

Add a validation check to prevent changing delegation to the same pool.

Raid can revert in rare case when the updated lastRaidTs is greater than block.timestamp

Resolved

Path

StateUpdate.sol, LibRaiding.sol

Function

updateGlobalState()

Description

RaidHandler.raid() calls LibGlobal.callStateUpdate(), which then delegatecalls the updateGlobalState().

In updateGlobalState(), when the lastRaidTsIn is greater than globalState.lastRaidTs then the globalState.lastRaidTs is updated to lastRaidTsIn. Now in RaidHandler.raid() the LibRaiding.raidPool() calls toggleStreak(), which subtracts updated lastRaidTs from block.timestamp .

Now it can happen that the lastRaidTs is greater than block.timestamp and that's why the transaction will revert because of an arithmetic error.

This can happen if the block.timestamp on another chain can be even slightly greater than what is on this chain. Assuming a possible variation in the block.timestamp results for multiple chains.

Recommendation

The condition can be added to check if the block.timestamp is less than the lastRaidTsIn, then it should not update the globalState.lastRaidTs

Inconsistent implementation in crossChainRaid function compared to other functions

Resolved

Description

Unlike other cross chain operations, the crossChainRaid uses an inconsistent implementation that utilizes the exact fee estimation. Other implementations are in line with the layerzero documentation on how to use the lzSend with Message(msg.value, 0) with the intention to refund users if excess tokens remain after crosschain message completion.

Recommendation

Maintain the implementation used in other operations for the crossChainRaid function.

Missing Some Event Emission for Incoming calls made from source chains

Resolved

Path

CrossChainManager

Function

1. crossChainStake
2. crossChainRaid
3. crossChainPolStake

Description

While the crossChainBudsTransfer function has an event emitted, the other cross chain operations do not have an event emitted.

Recommendation

Emit events for other operations to ease on-chain tracking.

Remove unused function from in the CrossChainManager Facet

Resolved

Path

CrossChainManager.sol

Function

_payNative

Description

The _payNative function is declared in the CrossChainManager facet but is not being utilized within or out of the facet.

Recommendation

Unused functions should be removed.

The user can get the reward for more than 1 epoch in the rare case

Resolved

Path

RaidHandler.sol

Function

raid(), sendRaidResult()

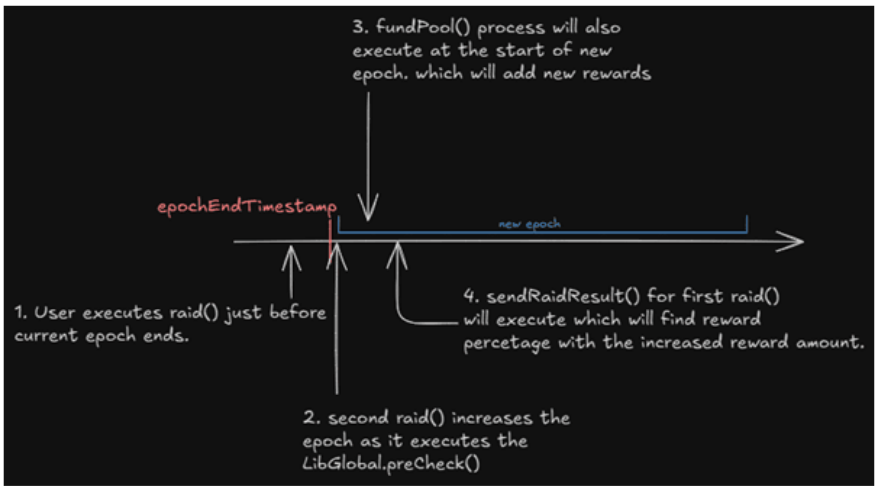
Description

The fundPool() process will be called in each epoch to fund the pool. But raid() calculates the fee based on that specific reward present at that moment/epoch for that pool in the normal case.

This type of scenario can happen in the rare case:
Example:
1. User raided the pool at the very last of the epoch timestamp (i.e., the first user's raid was at the very end of the current epoch)
2. The timestamp on the blockchain increases. Any action that increases the epoch on the contract level, like another raid() will happen.
3. fundPool() process will also be called at the start of the new epoch, which will add another reward amount.
4. Assuming that sendRaidResult() (initiated by the first raid()) will execute after the reward for the second epoch is added. While distributing the reward, it will be distributed for two epochs, even though the user paid the raid fee according to the previously present reward at the previous epoch.

This happens because when the pool is funded, the amount is stored in the LibReward.getRewardStorage().currentPooledRewards and LibRaiding.getRaidReward(), which calculates the payout amount, takes the currently present LibReward.getRewardStorage().currentPooledRewards into account while calculating the raid rewards on L76-L78.

This happens because the callback delay for the sendRaidResult() as shown in the diagram below.



Recommendation

This can be mitigated by maintaining the new record for the rewards per epoch when the user raids the pool, so the reward's current amount present for the epoch can be recorded, and that can be used when the sendRaidResult() for that user's raid will be executed for reward calculation. Instead of using the current LibReward.getRewardStorage().currentPooledRewards amount, which can be different as mentioned in the above description.

Incorrect _rewardsPooled passed to LibRaiding.raidPool()

Resolved

Path

CrossChainManager.sol

Function

_lzReceive()

Description

Incorrect _rewardsPooled argument passed to LibRaiding.raidPool(), it should be currentPooledRewards from IStakingPool(LibPoolManager.getPoolAddressById(poolId)).getPoolInfo() instead of IStakingPool(LibPoolManager.getPoolAddressById(poolId)).getTvlInPool() in _lzReceive()

The currentPooledRewards from IStakingPool(LibPoolManager.getPoolAddressById(poolId)).getPoolInfo() is the rewards amount added while funding the pool. Which is similar to what is used in the RaidHandler.raid() for passing _rewardsPooled to the LibRaiding.raidPool().

The IStakingPool(LibPoolManager.getPoolAddressById(poolId)).getTvlInPool() returns the _stakedBudsVolume which should not be used in this case.

Recommendation

Use the correct argument as suggested.

The contract should not allow staking and raiding the blacklisted pool

Resolved

Path

CrossChainManager.sol

Function

_onStake(), _lzReceive(), crossChainRaid(), crossChainStake()

Description

Currently in the _onStake() its not checked that the poolAddress is blacklisted or not.
Also while raiding on destination chain on L157 it's not checked that the pool is blacklisted or not.

Recommendation

Check the poolAddress is not blacklisted.

In case of _onStake() if it is checked that the pool is not blacklisted on smart contract level and the transaction is reverted on the destination chain. Then again, the functionality to recall or to get the burned token back on the source chain needs to be added.

In case of raiding crosschain with crossChainRaid(), if it is checked that the pool is not blacklisted on smart contract level and the transaction is reverted on the destination chain. Then again, the functionality to recall or to get the raid fee back on the source chain needs to be added.

Other solution can be to handle case on frontend so users can't select the blacklisted pool for cross-chain staking and raiding
(which will keep this issue open on the smart contract level)

Add required state updates

Resolved

Path

CrossChainManager.sol#L201

Function

crossChainStake()

Description

LibGlobal.preCheck(), LibGlobal.intStore().noOfStakerOnThisChain increment and LibGlobal.mappingStore().aggreagatedRecord[msg.sender] related updates are not happening in the crossChainStake() or in the _onStake().

Care of these state changes can be taken while executing the _onStake() on the dest chain.

Recommendation

Consider updating the required state in the crossChainStake() that is normally updated when staking with Router.addStake().

Prevent farmer delegation changes to the pool id 0

Resolved

Description

The function changeDelegation() should revert if the poolId entered is 0.

Recommendation

Add a check to revert the transaction if the poolId entered is 0.

The user should not be able to delegate the NFT to the blacklisted pool.

Resolved

Path

FarmerAlloc.sol

Function

delegateFarmer()

Description

Check that the user should not be able to delegate the NFT to the discarded/blacklisted pool ID using LibPoolManager.isWhitelistedPoolById()

Recommendation

Consider adding a require check

The user should not be able to raid the blacklisted pool.

Resolved

Path

RaidHandler.sol

Function

raid()

Description

The user should not be able to raid the blacklisted pool.

Recommendation

Consider adding a require check, which will revert if the poolId for which the user is raiding is black-listed.

Pools with 0 TDFs can be skipped while sending the reward.

Resolved

Path

FarmerAlloc.sol

Function

fundFarmerBonus()

Description

fundFarmerBonus() distributes the given allocation to each pool based on how much TDFs are present for that pool. While sending it uses transferFrom() to send bonusPerFarmer*tdf amount of tokens to each pool.

This block of code can be optimized by using continue statement which will skip the current iteration instead of sending 0 Buds amount to the pool if the TDF is 0 for that pool on L199.

Recommendation

Consider using continue if the tdf for that specific pool is 0.

Redundant functionality

Acknowledged

Path

FarmerAlloc.sol

Function

-

Description

The contract has functionality to change the pool id for which the NFT is delegated, but the user is going to get a reward accordingly to the bonusPerFarmer calculated in the fundFarmerBonus() depending on what was the allocation and the number of tdf at that (past or the epoch in which the funding is happening) epoch.

This can be understood by looking at the claimDelagationReward(), which uses LibFarmerAlloc.getAllocationByEpoch(i) for getting rewards at the specific epoch. Which shows the pool id doesn't matter. Still, we acknowledge that it can happen that the number of delegated users per pool will be used in any other future functionality where delegating to the specific pool or changing the delegated pool makes sense.

Recommendation

Consider verifying the functionality

Informational Severity Issues

Floating pragma

Acknowledged

Path

contracts/

Function

-

Description

Multiple contracts are using version ^0.8.0 with a floating pragma instead of locking to a specific version. Floating pragmas allow the contract to be compiled with any version greater than or equal to the specified version for that major version. If the contract wasn't thoroughly tested with that version, this can introduce possible bugs.

Recommendation

Consider using a fixed solidity version whenever possible.

Incorrect comment

Resolved

Path

LibRaiding.sol

Function

finalizeRaid()

Description

In finalizeRaid() the comment says 5% on 1st boost, 10 on 2nd, 15 on 3rd and 15 on 4th.
But for the 4th boost, it would be 20%.

successThreshold = successThreshold * (100 + (boosts * 5)) / 100
successThreshold = 5000 * (100 + (4 * 5)) / 100
successThreshold = 6000

So it is 1000 in addition to 5000
=1000/5000 = 0.2
= 0.2 * 100 = 20

So it shows it is 20 percent on top of the dthe successThreshold and not 15 for the 4th boost.
(It can be directly calculated as 4 * 5 = 20)

Recommendation

Verify and correct the comment.

Hardcoded level requirements

Resolved

Path

LibLevelManager

Function

levelUpFarmer, levelUpNarc

Description

The level requirements were hardcoded which makes adjusting impossible after the contract has been deployed.

Recommendation

To achieve a flexible and dynamic flow, design this so that level requirements could be adjusted and configured.

Redundant import

Resolved

Path

LibLz.sol, LevelManager.sol, OAppCoreUp.sol

Function

-

Description

These imported libraries are not used within the contracts/libraries.

IAsset.sol is imported in the LibLz.sol but never used.

BytesParsing.sol, LibDiamond.sol are imported in the LevelManager.sol but never used.

Initializable.sol, UUPSUpgradeable.sol, OwnableUpgradeable.sol are imported in the OAppCoreUp.sol but never used.

Recommendation

Unused imported libraries should be removed.

Incorrect check as PoolId cannot be less than zero

Resolved

Path

FarmerAlloc.sol

Function

delegateFarmer

Description

The first check is only reachable when the passed poolId parameter is zero but cannot be less than zero because it is not a signed integer.

Recommendation

Address the check to use the strict equal to rather than less than or equal to.

The meaningful error message can be added

Resolved

Path

LibGlobal.sol

Function

callStateUpdate(), updatePoolTvl()

Description

Both callStateUpdate() and updatePoolTvl() perform delegatecall on the facet addresses. The return value for those calls is checked in both functions with the require statement, the error message string is myFunction failed which is less clear for debugging.

A more meaningful error message can be added.

Recommendation

Consider adding more meaningful error messages.

Contradictory comment

Resolved

Path

Diamond.init.sol

Function

init()

Description

For init(), the comment mentions the first parameter's included addresses are buds token, farmer token, narc token, xp token, stbuds token but the init() assigns the addresses in the order where _tokenAddresses[3] is stBuds address and _tokenAddresses[4] is xp address. So there is a difference between comments where at the 3rd index the XP address and on the 4th the stBuds is there.

While the order in the comment doesn't create a problem, the team decides to highlight it to avoid any confusion.

Recommendation

Correct the order of addresses in the comment.

Fix the typo

Resolved

Path

FarmerAlloc.sol#L133

Function

claimDelagationReward()

Description

Typo in claimDelagationReward() function name can be fixed by making it claimDelegationReward().

Recommendation

Consider fixing the typo.

Add events according to the requirement

Resolved

Path

FarmerAlloc.sol

Function

-

Description

There are no events emitted in any of the functions present in FarmerAlloc contract. Depending on the requirement, the events can be added

Recommendation

Consider checking the requirements and adding events if required.

Functional Tests

Some of the tests performed are mentioned below:

DiamondCut.sol

- ✓ Should be able to add a facet with a diamond cut
- ✓ Should be able to remove the function
- ✓ Should be able to replace the facet
- ✓ Should revert if a non-owner try to adding a facet

DiamondLoupe.sol

- ✓ Should be able to get the function selectors associated with the facet address
- ✓ Should be able to get all the facet addresses
- ✓ Should be able to get the facet address associated with the function selector

CrossChainManager.sol

- ✓ Should be able to cross-chain stake
- ✓ Should be able to cross-chain raid
- ✓ Should be able to transfer tokens cross-chain
- ✓ Should be able to stake POL cross-chain
- ✓ Should return the correct endpoint
- ✓ Should return the correct peer

FarmerAlloc.sol

- ✓ Should be able to delegate the farmer NFT to the pool ID
- ✓ Should be able to change the delegated pool ID
- ✓ Should be able to remove the delegation
- ✓ Should be able to claim the delegation reward
- ✓ The claimed rewards should be according to the allocation and the tdf for that epoch

LevelManager.sol

- ✓ Should be able to level up Farmer NFT

- ✓ Should be able to level up NARC NFT
- ✓ Should change the token URI when the level is increased

PoolFactory.sol

- ✓ Only the contract owner should be able to deploy the new pool

RaidHandler.sol

- ✓ Should be able to raid the pool
- ✓ Should take a valid raid fee based on the reward, raid level and fee percentage
- ✓ Should revert if the risk level is not valid
- ✓ Should revert if the max boost is already used in one epoch while raiding
- ✓ Should increase the streak if the time elapsed after the last raid is between 24 to 28 hours
- ✓ Should increase the epoch number if the epoch time has elapsed

StateUpdate.sol

- ✗ updateGlobalState() should be able to update the state
- ✗ getPoolData() should be able to update the state
- ✗ Should protect against replay attacks

Diamond.sol

- ✓ Should delegatecall to the valid facet
- ✓ Should revert if there's no valid facet present for the specific msg.sig

Router.sol

- ✓ Should be able to add stake
- ✓ Staking, claiming, unstaking, discarding, and funding should increment the epoch if the epoch time has elapsed
- ✓ Should mint stBuds in a 1:1 ratio while staking
- ✓ Should be able to claim rewards
- ✓ Should be able to unstake the staked buds amount
- ✓ Should burn the stBuds amount in 1:1 while unstaking
- ✓ Owner should be able to fund the pool for the current epoch
- ✓ Owner should be able to blacklist a specific pool ID

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Bakeland. We performed our audit according to the procedure described above.

Issues of High, Medium, low, and informational severity were found. The Bakeland team acknowledged two of them and resolved the rest of the issues

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty pro-gram to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

June , 2025

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com