Cyberscope

Audit Report

Bera Reserve

September 2024

Repository	https://github.com/ReallyGreatTech/BeraReserve-contracts
Commit	770311de3e03d8c5a9cf7d7c7139d5c31c2c8ff8
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/ReallyGreatTech/BeraReserve-contracts
Commit	770311de3e03d8c5a9cf7d7c7139d5c31c2c8ff8

Audit Updates

Initial Audit	04 Sep 2024 https://github.com/cyberscope-io/audits/blob/main/brr/v1/audit.pdf
Corrected Phase 2	09 Oct 2024

Source Files

Filename	SHA256
wOHM.sol	3bcaabffcf448ead1fade190f703a7dc7439bcfa5e1d3511e73882d8e3175553
wETHBondDepository.sol	397594987fb9ac22c0c559714dd32f3c699da821770cc952ddbece69937429a7
sOlympusERC20.sol	904c1a80bc4d255b4084ac58a58376c5a4b360aa4b5b22f988b77a95aca10108
VaultOwned.sol	64e5fb3fbd4d62b98abb97f474a2238c62a3bd9a77c04e07b2344101ddba6633
Treasury.sol	b8683bfc1355b27c5b003603a9bcf1eaceb4df019554d3bb25f4e1ebc86f79bc
StandardBondingCalculator.sol	22f6ee9733cd70d9d8fa13607ea1dff7bdd1706e9b59c2e516a0dec1cad7fb01

StakingWarmup.sol	f468591a60f57c23e7d91341fdb5c385d332e24568b0e15686ef121ab0aa958f
StakingHelper.sol	680bddfa764c9f779b20ae45b3e7e9a60652b42a3312c580eb31aa6d6b6f55ea
StakingDistributor.sol	fda61de749892f81836c6669413d7e0a97fb968fabaabb2f4b891912cec2079b
Staking.sol	0b3f251920df3c955bfd1c39053eb475ac835dec50b7e0d55bb50ebe69f8b9f3
RiskFreeValueOfNonReserve.sol	51e8c085cf952c219c550114a9615ecec45242614c43f6dcbfab00c46b48114c
RedeemHelper.sol	7c0f5cd2f873a99613f887390fb7a924f5a1fce2de08c495444c6f5c0d706a71
BondDepository.sol	bcc317f62e1e917d97334cfcaf3ad7c22f8d6681b8f49b61f2d0756df2de247b
BeraReserveToken.sol	6667224e1e4c6acfb741a22085f0b1781d735415587e0cad0d6b8cda9d3dab42
BeraReservePreBondSale.sol	6077eb7cbb6ec19ed3c40ae4f680ef29b306ad39a5fd86029d7d84bea7e1ef1f
BeraReserveLockUp.sol	35aa919066afb5684990ccd42f570517245bb8e1a1f92d0b3217a1be46922352
BeraRerserveFeeDistributor.sol	046b0dd1953f2b76cb8c87acfd1d44992c2f8139156474ac14214ea8cd0f58d7
utils/BeraReserveTokenUtils.sol	6a2755585b50643371be862eead186baa9a350a137a9eda85d95ccb749d3cb50
types/BeraReserveTypes.sol	b38a8ab4bb41c9044a161362e8a0e205ed9fb42fc7641296b75848ba00ed5c32

Overview

The Bera Reserve contracts implement a comprehensive suite of functionalities to support a decentralized financial ecosystem focused on staking, bonding, treasury management, and rewards distribution. The `BeraReserveToken` and `sOlympus` (staked token) form the core of the tokenomics, enabling users to stake, earn, and wrap tokens while benefiting from the protocol's growth. The `Treasury` manages protocol reserves, allowing for the secure minting, withdrawal, and management of assets. Bonding and staking functionalities are further enhanced by the `BondDepository`, which facilitates bonding operations, and the `Staking` contract, which oversees the staking mechanics and reward distribution. Supporting contracts such as `Distributor`, `StakingHelper`, `RedeemHelper`, and `VaultOwned` ensure seamless interactions, efficient reward distribution, and secure access control. Together, these contracts create a robust and flexible DeFi infrastructure that aligns incentives for all participants and maintains the protocol's integrity and stability.

BeraReserveFeeDistributor File

The `BeraReserveFeeDistributor` contract is designed to manage and distribute fees in the form of a specific token, `beraReserveToken`, to three different entities: the team, the protocol-owned liquidity (POL), and the treasury. The contract maintains and updates the allocations of fees among these entities based on predefined percentage shares (33% each to the team and POL, and 34% to the treasury). It tracks accumulated tokens and periodically distributes them according to these shares. The contract also allows for the adjustment of recipient addresses and their respective shares, ensuring flexibility in fee distribution management. Additionally, it provides functions to query current allocations, recipient addresses, accumulated tokens, and the last update timestamp, thus enabling transparent and efficient management of token reserves.

BeraReserveLockUp File

The `BeraReserveLockUp` contract is designed to implement a vesting schedule for distributing BRR tokens among three groups: the team, marketing members, and seed round investors. The contract defines specific vesting conditions for each group: the team members' tokens vest linearly over one year with a three-month cliff; marketing members' tokens vest linearly over one year with no cliff; and seed round investors receive 30% of their allocation immediately at the token generation event (TGE), with the remaining 70% vesting linearly over six months. The contract allows for adding multiple members to each vesting schedule, allocating tokens according to the vesting rules, and automatically staking these tokens to the BeraReserveStaking contract. Additionally, it provides functionality to unlock vested tokens, ensuring members can access their tokens as they become available according to their schedules. This contract facilitates controlled and structured distribution of tokens to align with the project's long-term objectives.

BeraReservePreBondSale File

The `BeraReservePreBondSale` contract facilitates a pre-bond sale event where users can purchase BRR tokens using USDC. The contract manages the sale by setting a maximum limit on the number of BRR tokens each wallet can purchase, maintaining the current sale state, and enforcing a vesting period of 5 days for purchased tokens. It verifies participants using a Merkle root to ensure eligibility and tracks the total number of BRR tokens sold and USDC raised. Additionally, the contract includes functions to start and end the sale, mint tokens for the sale, set token prices, and manage protocol-related configurations. Users can unlock their vested tokens gradually according to their vesting schedule. The sale process is controlled by the contract owner, who can pause or unpause the sale and handle any unsold tokens appropriately by burning them.

BeraReserveToken File

The `BeraReserveToken` contract is a core token contract for the Bera Reserve ecosystem, implementing ERC20 functionality with additional controls for minting, burning, fee application, and decay mechanisms. The contract defines a total supply of 1,000,000 BRR tokens and allocates them to various purposes such as team vesting, marketing, treasury, liquidity, seed rounds, pre-bond sales, airdrops, and rewards/incentives. It includes multiple roles (such as minters and burners) for controlled token issuance and destruction. Fees can be applied to transactions, including buy and sell fees, and adjusted based on the market conditions, treasury value, and sliding scale mechanisms. The contract also incorporates a decay mechanism to burn tokens from non-exempt accounts over time and dynamically adjusts fees and decay ratios based on contract-specific parameters. Overall, it provides robust functionality for managing token economics and supply within the Bera Reserve ecosystem.

BondDepository File

The `BondDepository` contract is designed to facilitate the issuance and management of bonds within a decentralized finance (DeFi) protocol, similar to DAO. It allows users to deposit a specific token (`principle`) in exchange for the protocol's native token (`OHM`) at a discounted rate. The bond issuance process involves several key elements:

1. **Bond Terms and Adjustments**: The contract allows the protocol's policy team to set and adjust bond terms, including the control variable (which affects bond pricing), vesting terms, minimum price, maximum payout, fee structure, and maximum debt limit. The policy team can also adjust these terms dynamically to manage the bond market's response.
2. **Bond Creation**: Users can create bonds by depositing `principle` tokens. The contract calculates the bond price, payout amount, and ensures that the deposit meets the bond's terms (e.g., maximum payout and slippage protection). The deposited `principle` is sent to the treasury, which in return mints `OHM` tokens. A portion of the bond is taken as a fee and sent to the protocol's DAO.
3. **Bond Redemption**: Bondholders can redeem their bonds either fully or partially, depending on how much of the bond has vested. The contract supports auto-staking, where the redeemed `OHM` tokens can be automatically staked, depending on the user's preference.
4. **Debt Management**: The contract tracks the total outstanding debt (i.e., the total value of all active bonds) and manages it by decaying over time as bonds are redeemed. This helps control the inflationary impact of bond issuance on the token supply.
5. **Bond Pricing and Adjustments**: Bond prices are dynamically calculated based on the current debt ratio (debt relative to the total supply of `OHM`). The contract includes mechanisms to adjust the control variable to influence the bond price, ensuring that the protocol can maintain a balanced supply-demand ratio.
6. **View Functions**: The contract provides several view functions to allow users to see details about their bonds, including the amount of `OHM` they can claim, the current bond price, and how much of their bond has vested.

7. **Recovery Mechanism**: The contract includes a function to recover any lost tokens (other than the `principle` and `OHM` tokens) and send them to the protocol's DAO, ensuring that no funds are accidentally locked within the contract.

Overall, this contract is central to the protocol's ability to raise funds and manage its native token's supply, playing a crucial role in the protocol's monetary policy and liquidity management strategies.

RedeemHelper File

The `RedeemHelper` contract streamlines the redemption of bond payouts across multiple bond contracts for a user. It maintains a list of active bond contracts and allows the `redeemAll` function to automatically redeem all pending payouts for a specified recipient, with the option to stake the redeemed tokens immediately. Authorized managers can add or remove bond contracts, keeping the list current. This contract simplifies the redemption process, reducing the need for multiple transactions.

sOlympusERC20 File

The `sOlympus` contract, also known as `sBRR` (Staked Bera Reserve), represents the staked version of the `OHM` token, which automatically accrues value through a process called rebasing. This token is designed to increase in supply periodically, reflecting the protocol’s profits distributed to stakers. The `sOlympus` contract uses a flexible supply mechanism where each rebase operation increases the total supply of `sBRR` proportionally based on profits generated by the protocol. It allows the staking contract to manage the rebasing process, ensuring that `sBRR` holders receive rewards relative to their share of the circulating supply. The contract maintains granular control over the distribution of staking rewards, offers functionality to convert between the underlying token amount and the adjusted balance (`gons`), and tracks each rebase event for transparency. This system enables users to benefit from the protocol's growth while holding a staked token that appreciates over time.

Staking File

The `Staking` contract manages the staking and unstaking of the protocol's native token (`OHM`) and its staked counterpart (`sOHM`). This contract is designed to facilitate a system where users can stake their `OHM` tokens to receive `sOHM` tokens, which represent a claim on a growing amount of `OHM` through a process called rebasing. The contract incorporates several key functionalities:

1. **Staking and Warmup Period**: Users can stake their `OHM` tokens by transferring them to the contract. When staked, tokens enter a "warmup" period, during which the staked tokens are temporarily locked to prevent instant withdrawal. The warmup period is set by the contract manager and helps to stabilize the staking process by discouraging short-term speculation. After the warmup period, users can claim their `sOHM` tokens.
2. **Claiming and Forfeiting**: After the warmup period, users can claim their `sOHM` tokens, representing their staked `OHM` plus any additional rewards from rebasing. Alternatively, users can forfeit their staked tokens, reclaiming their original `OHM` and effectively canceling their stake.
3. **Rebasing**: The contract periodically triggers a rebase operation, which increases the total supply of `sOHM` tokens according to the protocol's reward mechanism. Rebases are automatically executed if the current block is beyond the epoch end block, redistributing `OHM` rewards to all `sOHM` holders proportionally. The rebase also distributes any excess balance in the contract to stakers.
4. **Epoch Management**: The staking process is governed by epochs, which are fixed periods defined by the protocol (e.g., a certain number of blocks). Each epoch has an associated end block and reward distribution amount. At the end of an epoch, the contract adjusts the total staked balance and recalculates rewards for the next epoch.
5. **Bonus Management**: The contract supports additional bonuses to be given to locked stakers. The bonuses can be granted by a designated `locker` address and reclaimed when necessary. This feature allows for incentivizing long-term staking.
6. **Contract Management**: The contract manager can set various related contract addresses, such as the distributor (responsible for distributing rewards), warmup contract (handling staking warmup mechanics), and locker contract (handling locked staking bonuses). Each of these contracts has specific roles that complement the staking process.
7. **Unstaking**: Users can unstake their `sOHM` tokens to receive back their `OHM` tokens. The unstaking process can trigger a rebase to ensure that the latest rewards are accounted for before the tokens are transferred back to the user.

Overall, the `Staking` contract provides a comprehensive staking mechanism that rewards participants for holding `OHM` tokens and allows the protocol to manage inflationary supply increases effectively while maintaining flexibility and security in the staking process.

The `Staking` contract utilizes the following contracts to manage staking, reward distribution, and bonding calculations:

- **Distributor Contract**:
    - Manages the distribution of staking rewards to recipients at regular intervals (epochs).
    - Maintains a list of recipients and their respective reward rates, which can be adjusted dynamically.
    - Handles minting of new rewards from the treasury and sends them to the designated recipients.
    - Supports reward rate adjustments to maintain the desired distribution rate over time.
- **StakingHelper Contract**:
    - Simplifies the staking process by batching multiple steps into a single transaction.
    - Handles the transfer and approval of `OHM` tokens from users to the staking contract.
    - Calls the staking contract's functions to stake tokens and claim any pending rewards, providing a seamless user experience.
- **StakingWarmup Contract**:
    - Manages the warmup period for new stakers, during which staked tokens are locked temporarily.
    - Ensures that staked tokens (`sOHM`) are held in the warmup contract until the warmup period expires.
    - Provides a function to retrieve staked tokens once the warmup period is complete, safeguarding the integrity of the staking mechanism.
- **BondingCalculator Contract**:
    - Calculates the value of liquidity provider (LP) tokens for the protocol's bonding mechanism.
    - Computes the K-value, total value of reserves, and valuations for LP tokens, which are essential for determining the fair price of bonds.

  - Provides functions to calculate markdowns for LP tokens, assisting in maintaining accurate bond pricing relative to market conditions.

wETHBondDepository File

The `wETHBondDepository` contract is similar to the `BondDepository` contract, both implementing a mechanism to create and redeem bonds, manage bond terms, and adjust control variables for pricing. However, there are notable differences:

1. **Use of wETH as Principle**: The `wETHBondDepository` specifically handles bonds created with wrapped Ether (wETH) as the principle token, as opposed to the general principle token in the `BondDepository`.
2. **Price Feeds and Valuation**: The `wETHBondDepository` incorporates a Chainlink price feed (`AggregatorV3Interface`) to get the real-time price of wETH, which it uses to convert bond prices to USD, providing a dynamic valuation mechanism based on actual market prices.
3. **Different Asset Types**: While the `BondDepository` may handle various asset types, including other cryptocurrencies, the `wETHBondDepository` is specifically tailored to wETH and includes specific functions and logic optimized for handling wrapped Ether transactions.

Overall, while both contracts serve the core purpose of managing bond issuance and redemption, the `wETHBondDepository` is tailored specifically for wETH, with price feed integrations and optimizations for dealing with Ether-based bonds.

Treasury File

The `Treasury` contract manages the reserve assets, liquidity, and minting processes for the protocol's native token, `OHM`. It handles several critical functions:

- **Asset Management**: Allows approved addresses to deposit reserve or liquidity assets in exchange for newly minted `OHM` tokens and facilitates withdrawals by burning `OHM` tokens in return for reserves. It also supports managing assets by allowing approved addresses to move assets from the treasury for liquidity management purposes.
- **Debt Management**: Enables approved debtors to incur debt against the protocol's reserves by borrowing `OHM` tokens or other reserve assets, up to a limit

determined by the amount of staked `sOHM` held by the debtor. Debtors can repay their debt using either reserve assets or `OHM` tokens.

- **Reward Distribution**: Mints new `OHM` tokens as rewards for staking or other incentivized activities. The amount of rewards minted is based on excess reserves, which are reserves not required to back the total supply of `OHM` tokens.
- **Excess Reserves Calculation**: Calculates the amount of excess reserves that are not backing the circulating supply of `OHM`, which can be used for rewards or other purposes.
- **Governance and Permissions**: Manages various roles and permissions, such as reserve depositors, spenders, managers, liquidity providers, and reward managers. Changes to these roles require a queuing process to ensure security and proper governance.
- **Checking and Reserve Valuation**: Performs checks to calculate and update the total reserves held by the treasury, ensuring accurate accounting and transparency. It also provides functions to determine the value of different reserve assets in terms of `OHM`.

Overall, the `Treasury` contract is crucial for maintaining the protocol's financial integrity by managing assets, debt, and rewards while ensuring that the protocol remains sufficiently backed and secure.

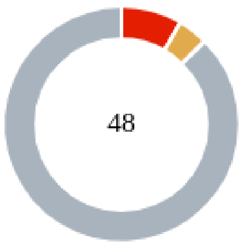
VaultOwned File

The `VaultOwned` contract is an access control utility that ensures specific functions can only be called by designated addresses associated with the protocol's vault, staking, and lockup components. It extends the `Ownable` contract, which restricts certain administrative functions to the contract owner, who can set the addresses for the vault, staking, and lockup contracts using dedicated functions (`setVault` , `setLockUp` , and `setStaking`). The contract provides public view functions (`vault` , `staking` , and `lockUp`) to retrieve these addresses, ensuring transparency. Additionally, it includes modifiers (`onlyVaultOrLockUp` and `onlyStaking`) that restrict function execution to the designated vault, lockup, or staking contracts, enhancing the security of operations that require interaction with these critical components. This contract is essential for maintaining proper access control and ensuring that only authorized contracts can perform sensitive operations within the protocol.

wOHM File

The `wOHM` contract, or "Wrapped sOHM," is an ERC20 token that allows users to wrap their staked `OHM` tokens (`sOHM`) into a wrapped version (`wOHM`) for added flexibility and composability within the broader DeFi ecosystem. The `wOHM` token represents a wrapped version of `sOHM` and provides functions to seamlessly convert between `OHM` , `sOHM` , and `wOHM` . Users can stake `OHM` tokens to receive `sOHM` and subsequently wrap these `sOHM` tokens into `wOHM` . The contract also allows users to unwrap `wOHM` back into `sOHM` or directly unstake `OHM` , making it easier to interact with various DeFi protocols that require a standard ERC20 token format. The conversion between `wOHM` and `sOHM` is determined by the current staking index, ensuring that the wrapped token accurately reflects the rebase-adjusted value of staked `OHM` . This design allows users to gain from the growth of staked assets while maintaining flexibility and compatibility with other platforms.

Findings Breakdown



●	Critical	4
●	Medium	2
●	Minor / Informative	42

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	3	1	0	0
●	Medium	0	2	0	0
●	Minor / Informative	42	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	IAC	Improper Address Check	Unresolved
●	IVC	Incorrect Vesting Calculation	Unresolved
●	MCPM	Market Cap Price Manipulation	Acknowledged
●	MFI	Missing Function Interface	Unresolved
●	ELFM	Exceeds Fees Limit	Acknowledged
●	MDF	Misleading Deposit Function	Acknowledged
●	CO	Code Optimization	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	HTD	Hardcoded Token Decimals	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IRC	Inaccurate Reward Calculation	Unresolved
●	IDLC	Incomplete Debt Limit Check	Unresolved

●	IBA	Inconsistent BRR Allocation	Unresolved
●	IDTU	Inconsistent Data Type Usage	Unresolved
●	IOT	Inconsistent Ownership Transfer	Unresolved
●	IPH	Inconsistent Parameter Handling	Unresolved
●	IRD	Inconsistent Rounding Down	Unresolved
●	IDH	Incorrect Decimal Handling	Unresolved
●	IARH	Insufficient Adjust Rate Handling	Unresolved
●	MPC	Merkle Proof Centralization	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MIEV	Missing Index Existence Validation	Unresolved
●	MPV	Missing Parameter Validation	Unresolved
●	MRRV	Missing Reward Rate Validation	Unresolved
●	MU	Modifiers Usage	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	POSD	Potential Oracle Stale Data	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved

●	TUU	Time Units Usage	Unresolved
●	UCI	Unnecessary Contract Interaction	Unresolved
●	UMF	Unrestricted Mint Function	Unresolved
●	UPF	Unused Pausable Functionality	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

IAC - Improper Address Check

Criticality	Critical
Location	BeraRerserveFeeDistributor.sol#L397
Status	Unresolved

Description

The contract is incorrectly using an `if` statement that only allows setting addresses when they are the zero address (`0x0`) and reverts in all other cases. As a result, legitimate addresses for essential components, such as the treasury, the team, and other critical roles, cannot be initialized or updated. This creates a significant limitation, effectively preventing the contract from being configured properly and may lead to operational failure or inability to deploy the contract effectively.

```
    function updateAddresses(address _team, address _pol,
address _treasury) external override onlyOwner {
        if (_treasury != address(0) || _pol != address(0) ||
_team != address(0)) {
            revert BERA_RESERVE__INVALID_ADDRESS();
        }
```

Recommendation

It is recommended to reconsider the way the `if` statement is applied. The check should be revised to ensure that the function reverts only if the address passed is the zero address (`0x0`), not the contrary. This change will allow proper initialization of the required addresses and ensure the contract functions as intended.

IVC - Incorrect Vesting Calculation

Criticality	Critical
Location	BeraReservePreBondSale.sol#L305
Status	Unresolved

Description

The contract is flawed in its calculation of the vested amount due to an improper determination of the vesting duration. In the function that calculates the vested amount, the `durationPassed` is divided by the `duration`, which is incorrectly set as the current block timestamp plus the vesting duration, instead of using only the vesting duration itself. This results in the duration being an excessively large number, causing the vested amount calculation to be divided by a much larger value than intended. Consequently, this miscalculation leads to a significantly lower amount of vested tokens than expected, undermining the vesting logic and potentially affecting the participants' token distribution.

```
 function vestedAmount(
    InvestorBondInfo memory investorBonds
  ) public view override returns (uint256) {
    if (block.timestamp >= investorBonds.duration) {
      return investorBonds.totalAmount;
    } else {
      uint256 durationPassed = block.timestamp -
investorBonds.start;

      uint256 totalVested = investorBonds.totalAmount.mulDiv(
        durationPassed,
        investorBonds.duration
      );

      return totalVested;
    }
  }
```

Recommendation

It is recommended to calculate the vested amount by dividing the `durationPassed` by the actual duration of the vesting period, excluding the addition of the block timestamp.

This adjustment will ensure that the calculation accurately reflects the intended vesting period and distributes the correct number of vested tokens.

MCPM - Market Cap Price Manipulation

Criticality	Critical
Location	BeraReserveTokenUtils.sol#L37
Status	Acknowledged

Description

The contract is vulnerable to manipulation due to its fee calculation mechanism, which relies on the total market capitalization (market cap) of the token. The fee applied to transactions is determined by the market cap relative to the treasury value, with different fee tiers depending on how much the market cap deviates from the treasury value. An attacker can exploit this mechanism by using a flash loan to artificially inflate or deflate the market cap. This temporary change allows the attacker to influence the fee calculation in a way that benefits them, either by lowering the fees they pay or by increasing fees for other users. This manipulation can undermine the fairness and stability of the fee mechanism, potentially causing financial loss to honest users and destabilizing the token's economy.

```
 function calculateSlidingScaleFee(
     uint256 mCap,
     uint256 treasuryValue,
     uint256 sellFee,
     uint256 tenPercentBelowTreasuryFees,
     uint256 twentyFivePercentBelowTreasuryFees,
     uint256 belowTreasuryValueFees
   ) public pure returns (TreasuryValueData memory rvfData) {
     if (mCap > treasuryValue) {
       rvfData.fee = sellFee;

       return rvfData;
     }

     uint256 tenPercentBelowTreasury =
treasuryValue.mulDiv(9_000, BASIS_POINTS);
     uint256 twentyFivePercentBelowTreasury =
treasuryValue.mulDiv(
       7_500,
       BASIS_POINTS
     );

     uint256 burn_treasuryFee_25Perc =
twentyFivePercentBelowTreasury / 2;
     uint256 burn_treasuryFee_10Perc = tenPercentBelowTreasury /
2;
     uint256 burn_treasuryFee_belowTreasury =
belowTreasuryValueFees / 2;

     if (mCap <= twentyFivePercentBelowTreasury) {
       ...
       return rvfData; // 16%
     } else if (mCap <= tenPercentBelowTreasury) {
       ...
     } else if (mCap <= treasuryValue) {
       ...

       return rvfData;
     }
   }
```

Recommendation

It is recommended to reconsider the reliance on market capitalization as a variable in the fee calculation mechanism. Implementing safeguards, such as time-weighted average market cap calculations or additional constraints on fee adjustments, could mitigate the risk of flash loan attacks and manipulation. Additionally, incorporating checks that detect rapid changes in market cap and adjusting the fee application accordingly could further enhance the contract's security against this type of attack.

Team Update

The team has acknowledged that this is not a security issue.

MFI - Missing Function Interface

Criticality	Critical
Location	IBeraReserveToken.sol#7 BeraReservePreBondSale.sol#L268
Status	Unresolved

Description

The contract is utilizing the `IBeraReserveToken` interface to interact with the `brrToken`, but it does not define the `safeTransfer` function in the `IBeraReserveToken` interface. As a result, when the contract attempts to call `safeTransfer` on the `brrToken`, it fails to compile. The lack of this function in the interface means that the compiler cannot recognize the `safeTransfer` method as a valid function call, leading to an unsuccessful build and potential functional issues if the contract is deployed without resolving this.

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.26;

import { IERC20 } from
"@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IBeraReserveToken is IERC20 {
    function mint(address account_, uint256 amount_) external;

    function burn(uint256 amount_) external;

    function burnFrom(address account_, uint256 amount_)
external;
}

...
brrToken.safeTransfer(msg.sender, releasableBRR);
```

Recommendation

It is recommended to include the `safeTransfer` function in the `IBeraReserveToken` interface to ensure compatibility and successful compilation. This will allow the contract to interact seamlessly with the `brrToken` when calling `safeTransfer` and ensure that the intended transfers are executed without errors.

ELFM - Exceeds Fees Limit

Criticality	Medium
Location	BeraReserveToken.sol#L265,281
Status	Acknowledged

Description

The contract owner has the authority to increase over the allowed limit of 25%. The owner may take advantage of it by calling the `setBuyFee` or `setSellFee` functions with a high percentage value.

```
function setBuyFee(uint256 _buyFee) external onlyOwner {
    if (_buyFee > BPS) revert BERA_RESERVE__FEE_TOO_HIGH();

    buyFee = _buyFee;

    emit BuyFeeUpdated(_buyFee);
}

 function setSellFee(uint256 _sellFee) external onlyOwner {
    if (_sellFee > BPS) revert BERA_RESERVE__FEE_TOO_HIGH();

    sellFee = _sellFee;

    emit SellFeeUpdated(_sellFee);
}
```

Recommendation

The contract could embody a check for the maximum acceptable value. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue.

MDF - Misleading Deposit Function

Criticality	Medium
Location	BondDepository.sol#L790
Status	Acknowledged

Description

The contract contains a `deposit` function that enables users to deposit funds to a recipient address specified as a parameter (`_depositor`). As a result, the function name is misleading, as it implies that the deposit is made on behalf of the caller when, in fact, the funds are deposited directly to the recipient address specified by the parameter. This could cause confusion for users or developers interacting with the contract, as they may mistakenly assume that the deposit is being made to their own account.

```
    function deposit(uint256 _amount, uint256 _maxPrice, address
_depositor) external returns (uint256) {
        require(_depositor != address(0), "Invalid address");

        decayDebt();
        require(totalDebt <= terms.maxDebt, "Max capacity
reached");
...
```

Recommendation

It is recommended to consider renaming the function to a more descriptive name, such as `depositFor` , to accurately convey that the deposit is being made for a specific recipient address rather than the caller. This would improve the contract's clarity and reduce the risk of misuse or misunderstanding by those interacting with it.

Team Update

The team has acknowledged that this is not a security issue.

CO - Code Optimization

Criticality	Minor / Informative
Location	BeraReservePreBondSale.sol#L145
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The `purchaseBRR` function performs multiple USDC transfers, potentially leading to higher gas consumption. The contract first transfers the entire `usdcAmount` from the user to itself. Then, it calculates a potential refund amount and transfers it back to the user if necessary. Finally, it transfers the remaining USDC to the `protocolMultisig` address. This multi-step approach with multiple transfers can be optimized for better gas efficiency.

```
function purchaseBRR(
    uint256 usdcAmount,
    bytes32[] calldata merkleProof
  ) external override {
    if (usdcAmount < 1e6) revert
BERA_RESERVE__INVALID_AMOUNT();
    ...
    uint256 usdcToRefund;

    if (brrAvailable == 0) revert BERA_RESERVE__BRR_SOLD_OUT();

    usdc.safeTransferFrom(msg.sender, address(this),
usdcAmount);

    ...
    if (investorAllocations[msg.sender].totalAmount != 0) {
      unlockBRR();
    }

    if (brrPurchaseAmount >= brrAvailable) {
      brrPurchaseAmount = brrAvailable;

      uint256 valueOfBrrTokensAvailable = brrAvailable.mulDiv(
        1e3,
        tokenPrice,
        Math.Rounding.Ceil
      );

      usdcToRefund = usdcAmount - valueOfBrrTokensAvailable;
    }

    ...
    usdcAmount -= usdcToRefund;

    if (usdcToRefund != 0) usdc.safeTransfer(msg.sender,
usdcToRefund);

    ...

    usdc.safeTransfer(protocolMultisig, usdcAmount);

    emit BRRTokensPurchased(
      msg.sender,
      uint128(brrPurchaseAmount),
      uint128(usdcAmount)
    );
  }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

It is recommended to calculate all necessary values (purchase amount, refund amount) before performing any USDC transfers. This can involve utilizing the `safeTransferFrom` function with a calculated net transfer amount instead of transferring the entire `usdcAmount` initially. By performing a single USDC transfer to the `protocolMultisig` address after all calculations are complete, the contract can significantly reduce gas costs associated with unnecessary transfers. This will optimize the function's efficiency and user experience.

CR - Code Repetition

Criticality	Minor / Informative
Location	BeraRerserveFeeDistributor.sol#L312,330,347
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specificaly the functions `allocateTeam` , `allocatePOL` and `allocateTreasury` share similar code segments.

```
  function allocateTeam() public override onlyOwner returns
(uint256) {
        ...
    }

    ///@dev allocate fees to pol
    function allocatePOL() public override onlyOwner returns
(uint256) {
        ...
        return pendingPolBeraReserve;
    }

    ///@dev allocate fees to treasury
    function allocateTreasury() public override onlyOwner
returns (uint256) {
        updateAllocations();
        ...
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	BeraReservePreBondSale.sol#L98,238,280 BeraReserveToken.sol#L273 RedeemHelper.sol#L88
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, the owner has sole control over critical aspects like setting the initial token price, initiating the sale, and modifying crucial parameters.

Additionally, the contract allows the owner to set the `uniswapV2` router address, which is responsible for determining the token's price. The function `setUniswapRouter` gives the owner the authority to specify the address of the Uniswap V2 router that will be used to calculate the token's price. If the owner sets an incorrect or malicious router address, it could lead to incorrect price calculations, potentially resulting in unfair trading conditions or manipulation of the token price.

Moreover, the policy has the ability to remove bond contracts without requiring any additional checks or validations.

```
 constructor(
    address _brrToken,
    address _usdc,
    address _protocolMultisig,
    uint128 _tokenPrice,
    bytes32 _merkleRoot
  ) Ownable(_protocolMultisig) {
    if (
      _brrToken == address(0) ||
      _protocolMultisig == address(0) ||
      _usdc == address(0)
    ) {
      revert BERA_RESERVE__INVALID_ADDRESS();
    }

    if (_tokenPrice == 0) {
      revert BERA_RESERVE__INVALID_AMOUNT();
    }

    brrToken = IBeraReserveToken(_brrToken);

    usdc = IERC20(_usdc);

    merkleRoot = _merkleRoot;

    protocolMultisig = _protocolMultisig;

    maxBondsPerWallet = 1_000e9;

    tokenPrice = _tokenPrice;
  }
...
  function startPreBondSale() external override onlyOwner {
    currentPreBondSaleState = PreBondSaleState.Live;
    emit PreBondSaleStarted(currentPreBondSaleState);
  }

  function setTokenPrice(uint128 _price) external override
onlyOwner {
    if (_price == 0) revert BERA_RESERVE__INVALID_AMOUNT();

    tokenPrice = _price;

    emit TokenPriceSet(_price);
  }
 function setUniswapRouter(address router) external onlyOwner {
    if (router == address(0)) revert
BERA_RESERVE__INVALID_ADDRESS();

    uniswapV2Router = IUniswapV2Router02(router);

    emit UniswapRouterUpdated(router);
  }
```

```
     function removeBondContract(uint _index) external
onlyPolicy {
         bonds[_index] = address(0);
     }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	BeraReservePreBondSale.sol#L145
Status	Unresolved

Description

The `purchaseBRR` function uses a specific conversion mechanism between USDC and BRR tokens by applying a multiplication factor (`1e9` and `1e3`) that assumes certain decimal precision. The variable `tokenPrice` is used as a conversion rate between USDC and BRR tokens, and it is expected to operate in conjunction with these constants.

However, this implementation assumes that the `usdcAmount` (with 6 decimals typical for USDC) is scaled against a price and an output (`brrPurchaseAmount`) which inherently expects a 9-decimal precision (`1e9`). Similarly, further calculations use a factor of `1e3` to adjust the USDC refund amount. This mismatch of decimals could cause inaccuracies in token conversion or handling.

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8` , it means that the smallest unit of the token is `0.00000001` , and if decimals are set to `18` , it means that the smallest unit of the token is `0.000000000000000001` .

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
function purchaseBRR(
    uint256 usdcAmount,
    bytes32[] calldata merkleProof
  ) external override {
    if (usdcAmount < 1e6) revert
BERA_RESERVE__INVALID_AMOUNT();
    ...
    //calculate amount of brr tokens bought
    uint256 brrPurchaseAmount = usdcAmount.mulDiv(1e9,
tokenPrice);
    ...

    if (brrPurchaseAmount >= brrAvailable) {
      brrPurchaseAmount = brrAvailable;

      uint256 valueOfBrrTokensAvailable = brrAvailable.mulDiv(
        1e3,
        tokenPrice,
        Math.Rounding.Ceil
      );

      usdcToRefund = usdcAmount - valueOfBrrTokensAvailable;
    }

    if (
      investorAllocations[msg.sender].totalAmount +
brrPurchaseAmount >
      maxBondsPerWallet
    ) {
      brrPurchaseAmount =
        maxBondsPerWallet -
        investorAllocations[msg.sender].totalAmount;

      usdcToRefund =
        usdcAmount -
        (brrPurchaseAmount.mulDiv(1e3, tokenPrice,
Math.Rounding.Floor));
    }
    ...
    }
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to a fixed number.

HTD - Hardcoded Token Decimals

Criticality	Minor / Informative
Location	BeraReserveLockUp.sol#L49
Status	Unresolved

Description

The contract is currently using hardcoded values to represent the token decimals (e.g., `200_000e9` for `TEAM_TOTAL_BRR_AMOUNT`). This approach can lead to inaccurate calculations and potential vulnerabilities if the token's decimal precision is ever updated. If the decimal values change, the hardcoded constants will no longer reflect the correct token amounts, resulting in incorrect transactions and distributions.

```
 uint128 public constant TEAM_TOTAL_BRR_AMOUNT = 200_000e9; //
200,000 BRR (20% of total supply)
  uint128 public constant MARKETING_TOTAL_BRR_AMOUNT =
50_000e9; // 50,000 BRR (5% of total supply)
  uint128 public constant SEED_ROUND_TOTAL_BRR_AMOUNT =
200_000e9; // 200,000 BRR (20% of total supply)
  uint128 public constant VESTING_TOTAL_BRR_AMOUNT = 450_000e9;
```

`

Recommendation

It is recommended to retrieve the decimals of the tokens using the `decimals` function provided by the token contract and apply them to the relevant variables. This will ensure that the calculations are always based on the current decimal precision of the token. By using the `decimals` function, the contract will be more flexible and adaptable to changes in the token's characteristics.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	BeraReserveToken.sol#L140,141 BeraReservePreBondSale.sol#L109
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
bera
usdc
maxBondsPerWallet
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

IRC - Inaccurate Reward Calculation

Criticality	Minor / Informative
Location	StakingDistributor.sol#L451
Status	Unresolved

Description

The `nextRewardFor` function within the `StakingDistributor.sol` contract is not accurately calculating the total rewards for a recipient when multiple reward entries exist. The current implementation iterates through the `info` array, but it only assigns the reward from the last matching entry to the `reward` variable. This results in the overwriting of previous rewards, leading to an incorrect calculation of the total amount due to the recipient.

```
    function nextRewardFor(address _recipient) public view
returns (uint256) {
        uint256 reward;
        for (uint256 i = 0; i < info.length; i++) {
            if (info[i].recipient == _recipient) {
                reward = nextRewardAt(info[i].rate);
            }
        }
        return reward;
    }
```

Recommendation

To ensure that the `nextRewardFor` function calculates the total rewards correctly, it is recommended to accumulate the results of the `nextRewardAt` invocations for each matching entry. This can be achieved by adding the individual rewards to a running total within the loop. By accumulating the rewards from all matching entries, this modified function will accurately calculate the total amount due to the recipient, ensuring that all rewards are properly accounted for.

IDLC - Incomplete Debt Limit Check

Criticality	Minor / Informative
Location	BondDepository.sol#L794,827
Status	Unresolved

Description

The `deposit` function checks if the current `totalDebt` is less than or equal to the maximum debt (`terms.maxDebt`) before allowing a deposit. However, it fails to verify if the sum of the current `totalDebt` and the proposed deposit amount (`_amount`) exceeds the maximum debt limit. This oversight could potentially allow for deposits that would push the total debt beyond the acceptable threshold, leading to unintended consequences or vulnerabilities.

```
function deposit(uint256 _amount, uint256 _maxPrice, address
_depositor) external returns (uint256) {
...
    require(totalDebt <= terms.maxDebt, "Max capacity
reached");
    ...
    totalDebt = totalDebt.add(value);
    ...
    }
```

Recommendation

It is recommended to add an additional check within the `deposit` function to verify that the sum of the current `totalDebt` and the `_amount` does not exceed the `terms.maxDebt` limit. By implementing this additional check, the contract will ensure that the total debt remains within the specified bounds, preventing excessive debt accumulation and maintaining the system's stability.

IBA - Inconsistent BRR Allocation

Criticality	Minor / Informative
Location	BeraReserveLockUp.sol#L198,237,297
Status	Unresolved

Description

The contract contains inconsistencies in the allocation of BRR amounts due to a missing deduction of previously unclaimed amounts. Specifically, when a new amount is set for a member, the unclaimed balance from the previous allocation is not accounted for, and the total BRR allocation (i.e., `totalTeamBRRAllocated`) is only increased by the new amount. The owner has the ability to set a new allocation for a member if the duration has passed. However, the contract does not take into account the previously allocated amount. This process can lead to the system allocating more BRR than originally intended, particularly when a new allocation is made after the `duration` time period has passed, allowing a new member schedule to be set without adjusting for the total BRR allocated so far.

```
       if (totalTeamBRRAllocated + totalAmount > TEAM_TOTAL_BRR_AMOUNT)
{
            totalAmount = TEAM_TOTAL_BRR_AMOUNT - totalTeamBRRAllocated;
        }

        if (totalAmount == 0) revert BRR_INVALID_AMOUNT();

        VestingSchedule memory userSchedule = teamsSchedules[_member];

        if (block.timestamp >= userSchedule.cliff +
userSchedule.duration) {
            teamsSchedules[_member].memberType = MemberType.TEAM;
            teamsSchedules[_member].start = uint32(block.timestamp);
            teamsSchedules[_member].cliff = uint32(block.timestamp) +
TEAM_VESTING_CLIFF;
            teamsSchedules[_member].duration = TEAM_VESTING_DURATION;
            teamsSchedules[_member].amountClaimed = 0;
            teamsSchedules[_member].totalAmount = totalAmount;
        } else {
            teamsSchedules[_member].totalAmount += totalAmount;
        }

        totalTeamBRRAllocated += totalAmount;
```

Recommendation

The team is advised to ensure whether overriding a member’s allocation is the intended functionality. If yes, the contract should handle the adjustment of all variables correctly and account for previous allocations. Adjust the logic of the allocation process to consider the unclaimed amounts from previous allocations. This will ensure that the total BRR allocated is consistently tracked and does not exceed the allowed limit. Additionally, thorough testing should be conducted to confirm that the total BRR allocation remains accurate across all member updates and vesting schedules. Otherwise, consider allowing only the possibility of increasing the amount instead of overwriting it.

IDTU - Inconsistent Data Type Usage

Criticality	Minor / Informative
Location	BeraRerserveFeeDistributor.sol#L314,332,349
Status	Unresolved

Description

The contract is currently using an `int256` data type for the variables. This is inconsistent with the usage of `uint256` for all other variables within the contract. Since these variables are specifically used to store accumulated reserves, which should always be positive values, there is no practical reason to use an `int256` data type. Using `int256` introduces unnecessary complexity and potential vulnerabilities, as it allows for negative values that are not logically possible in this context.

```
int256 accumulatedTeamBeraReserve = int256(teamShare *
accumulatedBeraReserveTokenPerContract);
...
int256 accumulatedPolBeraReserve = int256(polShare *
accumulatedBeraReserveTokenPerContract);
...
int256 accumulatedTreasuryBeraReserve = int256(treasuryShare *
accumulatedBeraReserveTokenPerContract);
```

Recommendation

It is recommended to replace the `int256` data type with `uint256` for the variables. This will ensure consistent data type usage throughout the contract and prevent potential errors that could arise from negative values. By using `uint256`, the contract will be more secure and easier to understand.

IOT - Inconsistent Ownership Transfer

Criticality	Minor / Informative
Location	StakingDistributor.sol#L335 BondDepository.sol#L46
Status	Unresolved

Description

The contract's `pull` function allows the current owner to retrieve ownership of the contract, even after a `renounce` has been called. This inconsistency can lead to unexpected behavior and potential security vulnerabilities. If the `push` function is called to transfer ownership to a new address, and then the current owner renounces ownership, a subsequent `pull` call by the previous owner can reclaim control of the contract, undermining the intended transfer of ownership.

```
   function renouncePolicy() public virtual override onlyPolicy
{
        emit OwnershipTransferred(_policy, address(0));
        _policy = address(0);
    }

    function pushPolicy(address newPolicy_) public virtual
override onlyPolicy {
        require(newPolicy_ != address(0), "Ownable: new owner
is the zero address");
        _newPolicy = newPolicy_;
    }

    function pullPolicy() public virtual override {
        require(msg.sender == _newPolicy);
        emit OwnershipTransferred(_policy, _newPolicy);
        _policy = _newPolicy;
    }
    function pullManagement() public virtual override {
        require(msg.sender == _newOwner, "Ownable: must be new
owner to pull");
        emit OwnershipPulled(_owner, _newOwner);
        _owner = _newOwner;
    }
```

Recommendation

It is recommended to implement a mechanism that prevents the `pull` function from being invoked after a `renounce` has been called. This can be achieved by introducing a boolean flag that indicates whether the contract's ownership has been renounced. The `pull` function can then check this flag and only allow the transfer if the `renounce` flag is not set. This will ensure that the `pull` function can only be used to transfer ownership before a `renounce` has been called, preventing unintended consequences and maintaining the integrity of the ownership transfer process.

IPH - Inconsistent Parameter Handling

Criticality	Minor / Informative
Location	BondDepository.sol#L732
Status	Unresolved

Description

The `setBondTerms` function is intended to modify various bond terms, but it currently lacks the ability to adjust the `minimumPrice` parameter. This omission prevents the contract from dynamically adjusting the minimum price of bonds, potentially limiting its flexibility and adaptability to changing market conditions. Additionally, the `vestingTerm` parameter can be both increased and decreased, which can lead to inconsistencies and discrepancies in the vesting system applied to individual bonds. This can disrupt the tokenomics of the system and potentially incentivize users to create new bonds to override existing vesting periods.

```
    function setBondTerms(PARAMETER _parameter, uint256 _input)
external onlyPolicy {
        if (_parameter == PARAMETER.VESTING) {
            // 0
            require(_input >= 10000, "Vesting must be longer
than 36 hours");
            terms.vestingTerm = _input;
        } else if (_parameter == PARAMETER.PAYOUT) {
            // 1
            require(_input <= 1000, "Payout cannot be above 1
percent");
            terms.maxPayout = _input;
        } else if (_parameter == PARAMETER.FEE) {
            // 2
            require(_input <= 10000, "DAO fee cannot exceed
payout");
            terms.fee = _input;
        } else if (_parameter == PARAMETER.DEBT) {
            // 3
            terms.maxDebt = _input;
        }
    }
```

Recommendation

It is recommended to include the `minimumPrice` variable within the `setBondTerms` function, allowing for its adjustment as needed. This will provide greater flexibility in managing the bond terms and adapting to changing market dynamics. Furthermore, we advise restricting the `vestingTerm` value to only allow for increases and to prevent decreases. This will ensure consistent vesting periods for all bonds, avoiding discrepancies and maintaining the integrity of the tokenomics system. By implementing these changes, the contract will be more adaptable, efficient, and less prone to inconsistencies.

IRD - Inconsistent Rounding Down

Criticality	Minor / Informative
Location	BeraReservePreBondSale.sol#L348 BeraReserveLockUp.sol#L351
Status	Unresolved

Description

The `BeraReservePreBondSale` contract does not consistently round down the calculated vested amount using the `Math.Rounding.Floor` method, as is done in the `BeraReserveLockUp` contract. This inconsistency can lead to calculation errors and discrepancies between the two contracts, especially when dealing with fractional amounts. The lack of consistent rounding down can result in overestimations of the vested amount, potentially affecting token distributions and financial calculations.

```
 function vestedAmount(
     InvestorBondInfo memory investorBonds
   ) public view override returns (uint256) {
     ...

       uint256 totalVested = investorBonds.totalAmount.mulDiv(
         durationPassed,
         investorBonds.duration
       );

       return totalVested;
     }
   }

   function vestedAmount(
     address member,
     MemberType memberType
   ) public view override returns (uint256) {
     ...
       uint256 totalVested =
uint256(schedule.totalAmount).mulDiv(
         durationPassed,
         schedule.duration,
         Math.Rounding.Floor
       );

       return totalVested;
     }
```

Recommendation

It is recommended to utilize the `Math.Rounding.Floor` method for rounding down the calculated vested amount in the `BeraReservePreBondSale` contract as well. This will ensure consistent rounding behavior across both contracts, preventing calculation errors and discrepancies. By adopting consistent rounding practices, the contracts will provide more accurate and reliable results, reducing the risk of financial losses or disputes.

IDH - Incorrect Decimal Handling

Criticality	Minor / Informative
Location	StandardBondingCalculator.sol#L262
Status	Unresolved

Description

The contract is incorrectly calculating the `k` value due to improper handling of decimal places. Specifically, the calculation does not correctly account for the different decimals of the tokens involved in the Uniswap pair. Instead of properly adjusting for the decimals of both tokens and the pair itself, the function attempts a simplified calculation, leading to an inaccurate `k` value. This can result in miscalculations affecting trading, liquidity provision, and other critical contract functions that depend on the correct value of `k`.

```
    function getKValue(address _pair) public view returns
(uint256 k_) {
        uint256 token0 =
IERC20(IUniswapV2Pair(_pair).token0()).decimals();
        uint256 token1 =
IERC20(IUniswapV2Pair(_pair).token1()).decimals();
        uint256 decimals =
token0.add(token1).sub(IERC20(_pair).decimals());

        (uint256 reserve0, uint256 reserve1,) =
IUniswapV2Pair(_pair).getReserves();
        k_ = reserve0.mul(reserve1).div(10 ** decimals);
    }
```

Recommendation

It is recommended to revise the calculation logic to properly account for the decimals of each token and the pair. The correct approach should ensure that the `k` value is accurately calculated by adjusting for the differences in decimal places, thereby maintaining consistency and reliability in the contract's operations and preventing potential errors or exploits due to miscalculated values.

IARH - Insufficient Adjust Rate Handling

Criticality	Minor / Informative
Location	StakingDistributor.sol#L424
Status	Unresolved

Description

The contract's `adjust` function aims to adjust the `info[_index].rate` for specific recipients. However, the current implementation has a limitation. If the `adjustment.rate` (step-by-step reduction or increase) is greater than the current `info[_index].rate`, the subtraction operation will result in a negative value. Since negative rates are likely unintended, this scenario would render the `adjust` operation impossible and potentially cause the entire `distribute` function to fail.

```
} else {
    // if rate should decrease
    info[_index].rate = info[_index].rate.sub(adjustment.rate);
// lower rate
    if (info[_index].rate <= adjustment.target) {
        // if target met
        adjustments[_index].rate = 0; // turn off adjustment
    }
```

Recommendation

It is recommended to implement a mechanism for gracefully handling situations where the reduction in a particular recipient's rate is greater than the current rate. This can be achieved by introducing a check within the `adjust` function. If the subtraction operation results in a value less than zero, the `info[_index].rate` should be set to zero instead of the negative value.

By implementing this change, the contract will ensure that the adjustment process handles potential underflows gracefully. Setting the rate to zero implies that the recipient no longer receives rewards, preventing unintended negative rates and potential failures within the `distribute` function. This will improve the robustness and reliability of the contract's rate adjustment mechanism.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	BeraReservePreBondSale.sol#L121,154
Status	Unresolved

Description

The contract uses a Merkle Proof mechanism in order to define many applicable addresses. The verification process is based on an off-chain configuration. The contract owner is responsible for updating the in-chain “Merkle Root” in order to validate correctly the provided message.

```
 constructor(
    address _brrToken,
    address _usdc,
    address _protocolMultisig,
    uint128 _tokenPrice,
    bytes32 _merkleRoot
  ) Ownable(_protocolMultisig) {
    ...

    merkleRoot = _merkleRoot;
    ...
    }

  function purchaseBRR(
    uint256 usdcAmount,
    bytes32[] calldata merkleProof
  ) external override {
    if (usdcAmount < 1e6) revert
BERA_RESERVE__INVALID_AMOUNT();

    if (currentPreBondSaleState != PreBondSaleState.Live)
      revert BERA_RESERVE__PRE_BOND_SALE_NOT_LIVE();

    if (
      !MerkleProof.verify(
        merkleProof,
        merkleRoot,
        keccak256(abi.encodePacked(msg.sender))
      )
      ...
      }
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	wOHM.sol#L935,937,939 wETHBondDepository.sol#L682,684,686,688,690,771,1059,1060 sOlympusERC20.sol#L1010,1058,1059,1071 Treasury.sol#L290,497,544 StandardBondingCalculator.sol#L258 StakingWarmup.sol#L95,97,102 StakingHelper.sol#L91 StakingDistributor.sol#L336,378,380,469,479 Staking.sol#L628,630,778,788 RedeemHelper.sol#L85 BondDepository.sol#L674,676,678,680,772,1091,1092
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(_staking != address(0))
require(_OHM != address(0))
require(_sOHM != address(0))
require(_principle != address(0))
require(_treasury != address(0))
require(_DAO != address(0))
require(_feed != address(0))
require(_token != OHM)
require(_token != principle)
require(msg.sender == stakingContract)
require(msg.sender == initializer)
require(stakingContract_ != address(0))
require(INDEX == 0)
require(_address != address(0))

...
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	VaultOwned.sol#L49
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function setVault(address vault_) external onlyOwner returns
(bool) {
        _vault = vault_;

        return true;
    }

    function setLockUp(address lockUp_) external onlyOwner
returns (bool) {
        _lockUp = lockUp_;

        return true;
    }

    function setStaking(address staking_) external onlyOwner
returns (bool) {
        _staking = staking_;

        return true;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MIEV - Missing Index Existence Validation

Criticality	Minor / Informative
Location	StakingDistributor.sol#L478,491
Status	Unresolved

Description

The `removeRecipient` and `setAdjustment` functions do not include checks to validate whether the specified `_index` corresponds to an existing entry in the `info` array. This oversight could cause the functions to revert without an error message. Consequently, the user will not know if the transaction is being reverted due to a mismatch of the recipient or because the `_index` provided is incorrect.

```
    function removeRecipient(uint256 _index, address _recipient)
external onlyPolicy {
        require(_recipient == info[_index].recipient);
        info[_index].recipient = address(0);
        info[_index].rate = 0;
    }

    function setAdjustment(uint256 _index, bool _add, uint256
_rate, uint256 _target) external onlyPolicy {
        adjustments[_index] = Adjust({ add: _add, rate: _rate,
target: _target });
    }
```

Recommendation

To prevent errors and ensure that the `removeRecipient` and `setAdjustment` functions operate correctly, it is recommended to include checks to verify the existence of the specified `_index` within the `info` array. This can be done by evaluating the `info[_index].recipient` address. By incorporating these checks, the contract will prevent errors and ensure that modifications are only applied to existing entries within the `info` array.

MPV - Missing Parameter Validation

Criticality	Minor / Informative
Location	BondDepository.sol#L696,758
Status	Unresolved

Description

The contract is missing essential checks to ensure that all parameters passed to critical functions are within acceptable limits. Functions such as `initializeBondTerms` and `setAdjustment` lack validation to confirm that the input values, like `_vestingTerm`, `_minimumPrice`, `_maxPayout`, `_fee`, `_maxDebt`, and others, fall within the acceptable ranges or constraints. Without these checks, there is a risk that out-of-bound or malicious values could be set, potentially leading to unexpected behavior, financial loss, or exploitation of the contract's logic.

```
   function initializeBondTerms(
        uint256 _controlVariable,
        uint256 _vestingTerm,
        uint256 _minimumPrice,
        uint256 _maxPayout,
        uint256 _fee,
        uint256 _maxDebt,
        uint256 _initialDebt
    ) external onlyPolicy {
        require(terms.controlVariable == 0, "Bonds must be
initialized from 0");
        terms = Terms({
            controlVariable: _controlVariable,
            vestingTerm: _vestingTerm,
            minimumPrice: _minimumPrice,
            maxPayout: _maxPayout,
            fee: _fee,
            maxDebt: _maxDebt
        });
        totalDebt = _initialDebt;
        lastDecay = block.number;
    }
...
    function setAdjustment(bool _addition, uint256 _increment,
uint256 _target, uint256 _buffer) external onlyPolicy {
        require(_increment <=
terms.controlVariable.mul(25).div(1000), "Increment too
large");

        adjustment =
            Adjust({ add: _addition, rate: _increment, target:
_target, buffer: _buffer, lastBlock: block.number });
    }
```

Recommendation

It is recommended to include additional checks to validate that all parameters fall within the acceptable limits. Implementing strict range validations and sanity checks for each parameter can prevent misconfigurations, reduce the risk of errors or exploits, and ensure that the contract operates as intended. This will enhance the contract's robustness and security by mitigating the impact of potentially harmful or unintended input values.

MRRV - Missing Reward Rate Validation

Criticality	Minor / Informative
Location	StakingDistributor.sol#L468
Status	Unresolved

Description

The `addRecipient` function does not include a validation to ensure that the specified `_rewardRate` does not exceed the maximum achievable value of `1000000`. This oversight could potentially allow for the creation of recipients with arbitrarily high reward rates, which could have unintended consequences on the overall reward distribution system.

```
    function addRecipient(address _recipient, uint256
_rewardRate) external onlyPolicy {
        require(_recipient != address(0));
        info.push(Info({ recipient: _recipient, rate:
_rewardRate }));
    }
```

Recommendation

To prevent the creation of recipients with excessively high reward rates, it is recommended to add a validation check within the `addRecipient` function. This check should verify that the `_rewardRate` is less than or equal to the maximum allowed value of `1000000`. By incorporating this validation, the contract will ensure that only reasonable reward rates are allowed, preventing potential imbalances in the reward distribution system and maintaining its fairness.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	BeraReserveLockUp.sol#L211,246,278
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
   if (totalTeamBRRAllocated + totalAmount >
TEAM_TOTAL_BRR_AMOUNT) {
      totalAmount = TEAM_TOTAL_BRR_AMOUNT -
totalTeamBRRAllocated;
    }
...
    if (totalMarketingBRRAllocated + totalAmount >
MARKETING_TOTAL_BRR_AMOUNT) {
      totalAmount = MARKETING_TOTAL_BRR_AMOUNT -
totalMarketingBRRAllocated;
    }
...
    if (
      totalSeedRoundBRRAllocated + totalAmount >
SEED_ROUND_TOTAL_BRR_AMOUNT
    ) {
      totalAmount = SEED_ROUND_TOTAL_BRR_AMOUNT -
totalSeedRoundBRRAllocated;
    }
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	wETHBondDepository.sol#L969
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
    function assetPrice() public view returns (int256) {
        (, int256 price,,,) = priceFeed.latestRoundData();
        return price;
    }
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	wETHBondDepository.sol#L969
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
    function assetPrice() public view returns (int256) {
          (, int256 price,,,) = priceFeed.latestRoundData();
          return price;
    }
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	BondDepository.sol#L817
Status	Unresolved

Description

The `safeTransferFrom` function are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
IERC20(principle).safeTransferFrom(msg.sender, address(this),
_amount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
 Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	wOHM.sol wETHBondDepository.sol sOlympusERC20.sol Treasury.sol StandardBondingCalculator.sol StakingDistributor.sol Staking.sol BondDepository.sol BeraRerserveFeeDistributor.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	BeraReserveTokenUtils.sol#L14 BeraReserveLockUp.sol#L39
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256 private constant DECAY_PERIOD = 365 * 24 * 60 * 60
seconds; // 1 year
...
uint32 public constant TEAM_VESTING_DURATION = 365 * 24 * 60 *
60 seconds; // 1 year
uint32 public constant TEAM_VESTING_CLIFF = 90 * 24 * 60 * 60
seconds; // 3 months
uint32 public constant MARKETING_VESTING_DURATION =
    365 * 24 * 60 * 60 seconds; // 1 year
uint32 public constant SEED_ROUND_VESTING_DURATION =
        180 * 24 * 60 * 60 seconds; // 6 months
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

UCI - Unnecessary Contract Interaction

Criticality	Minor / Informative
Location	StakingHelper.sol#L106 BondDepository.sol#L894
Status	Unresolved

Description

The contract is currently using the `StakingHelper` contract to perform staking operations. However, the logic implemented within the `StakingHelper` contract could be directly executed within the relevant functions of the current contract. This unnecessary interaction with an external contract can increase gas costs and potentially introduce additional vulnerabilities if the `StakingHelper` contract is compromised.

```
IStakingHelper(stakingHelper).stake(_amount, _recipient);
...
function stake(uint _amount) external {
    IERC20(OHM).transferFrom(msg.sender, address(this),
_amount);
    IERC20(OHM).approve(staking, _amount);
    IStaking(staking).stake(_amount, msg.sender);
    IStaking(staking).claim(msg.sender);
}
```

Recommendation

It is recommended to refactor the code to eliminate the reliance on the `StakingHelper` contract. By directly implementing the staking logic within the relevant functions, the contract can reduce gas costs and improve efficiency. This approach will also reduce the potential attack surface by removing the dependency on an external contract. Additionally, it will provide greater control over the staking process, allowing for potential customization or modifications without relying on changes to the `StakingHelper` contract.

UMF - Unrestricted Mint Function

Criticality	Minor / Informative
Location	BeraReserveLockUp.sol#119 BeraReservePreBondSale.sol#L228
Status	Unresolved

Description

The contract is susceptible to a vulnerability due to the unrestricted usage of the `mint` function. The `mint` function allows the owner to mint specific allocations of tokens to the contract. However, there is no mechanism in place to prevent the owner from invoking the `mint` function multiple times. This oversight means that the owner can mint more tokens than initially intended, which could significantly alter the tokenomics of the contract. If more tokens are minted than expected, this could exhaust the total supply cap, preventing other contracts from minting their allocated supply and causing potential disruptions in the ecosystem and loss of trust among stakeholders.

```
 function mintBRR() external override onlyOwner {
    brrToken.mint(address(this), VESTING_TOTAL_BRR_AMOUNT);

    emit TotalBRRMinted(VESTING_TOTAL_BRR_AMOUNT);
  }
...
  function mintBRR() external override onlyOwner {
    brrToken.mint(address(this),
PRE_BOND_SALE_TOTAL_BRR_AMOUNT);

    emit TotalBRRMinted(PRE_BOND_SALE_TOTAL_BRR_AMOUNT);
  }
```

Recommendation

It is recommended to implement an additional check that prevents the repetitive execution of the `mint` function. This could include a state variable that tracks whether the `mint` function has been executed and ensures that it cannot be called more than once. By adding such safeguards, the contract will be protected from unintentional or malicious over-minting, thereby preserving the intended token supply and maintaining the integrity of the tokenomics.

UPF - Unused Pausable Functionality

Criticality	Minor / Informative
Location	BeraReservePreBondSale.sol#L272
Status	Unresolved

Description

The contract implements the `Pausable` interface, providing functions to pause and unpause its functionality. However, the contract does not utilize the `whenNotPaused` and `whenPaused` modifiers provided by the `Pausable` interface. As a result, the intended pausable functionality is not effectively enforced, and the contract can continue to execute even when paused.

```
 function pause() external override onlyOwner {
     _pause();
 }

  function unpause() external override onlyOwner {
     _unpause();
 }
```

Recommendation

It is recommended to consider the actual need for the pausable functionality within the contract. If the ability to temporarily halt contract execution is essential, the contract should utilize the `whenNotPaused` and `whenPaused` modifiers appropriately. By applying these modifiers to relevant functions, the contract can effectively pause and unpause its operations, ensuring that only authorized actions can be executed during specific periods.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	wOHM.sol#L921,927,948,964,980,993,1006,1015 wETHBondDepository.sol#L15,16,237,280,282,284,286,288,290,429,549,553,632,635,704,705,706,707,708,709,736,758,770,790,839,874,938,1024,1041,1058 VaultOwned.sol#L14,45,46,47 utils/BeraReserveTokenUtils.sol#L16 Treasury.sol#L121,122,241,258,262,268,272,312,338,358,384,404,420,441,481,496,543,670,686 StandardBondingCalculator.sol#L123,129,255,262,271,275,283 StakingWarmup.sol#L101 StakingHelper.sol#L85,95,104 StakingDistributor.sol#L278,306,307,352,414,442,451,468,478,491 Staking.sol#L413,503,504,599,650,677,708,720,777,787,803,822 sOlympusERC20.sol#L441,536,539,542,545,548,551,903,964,965,1006,1033,1070 RedeemHelper.sol#L15,16,74,84,89 BondDepository.sol#L15,16,237,280,282,284,286,288,290,429,551,555,594,595,621,624,698,699,700,701,702,703,704,733,759,771,791,853,886,953,1041,1059,1074,1090 BeraReserveToken.sol#L58,215,242,258,266,274,282,290,296,304,312 BeraReservePreBondSale.sol#L215,233 BeraReserveLockUp.sol#L192,231,267 BeraRerserveFeeDistributor.sol#L193,397,408,430
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).

4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
contract wOHM is ERC20 {
    using SafeERC20 for ERC20;
    using Address for address;
    using SafeMath for uint;

    address public immutable staking;
...
        @param _amount uint
        @return uint
     */
    function wOHMValue(uint _amount) public view returns (uint)
{
        return _amount.mul(10 **
decimals()).div(IStaking(staking).index());
    }
}

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	wETHBondDepository.sol#L559,560 StandardBondingCalculator.sol#L135,136 sOlympusERC20.sol#L539 BondDepository.sol#L561,562 BeraReserveToken.sol#L45,50,51,52
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
uint256 private constant Q224 =
0x100000000000000000000000000000000000000000000000000000000
uint256 private constant LOWER_MASK =
0xffffffffffffffffffffffffffff
mapping(address => mapping(address => uint256)) internal
_allowances

rnal constant BURNER_ROLE = keccak256("BURNER_ROLE");
    uint25
rnal constant TREASURY_TOTAL_BRR_AMOUNT = 200_000e9; // 200,000
rnal constant PRE_BONDS_TOTAL_BRR_AMOUNT = 50_000e9; // 50,000
rnal constant AIRDROP_TOTAL_BRR_AMOUNT = 120_000e9; // 120,000
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	VaultOwned.sol#L50,56,62
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
_vault = vault_
_lockUp = lockUp_
_staking = staking_
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	wETHBondDepository.sol#L719 Staking.sol#L725,779,789,823 BondDepository.sol#L715 BeraReserveToken.sol#L313
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
totalDebt = _initialDebt
totalStaked = totalStaked.sub(_amount)
totalBonus = totalBonus.add(_amount)
totalBonus = totalBonus.sub(_amount)
warmupPeriod = _warmupPeriod

e = _treasuryValue;
    }
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	wOHM.sol#L281,310,342,355,374,394,407,767,805,812,831,850,868,893 wETHBondDepository.sol#L102,126,134,145,149,187,191,203,207,218,237,365,373,416,484,492,497,562 Treasury.sol#L89 StandardBondingCalculator.sol#L40,85,139,165 StakingDistributor.sol#L9,13,17,25,30,36,98,111,115,119,123,128,132,156,169,177,181,188,192,204,230,234,246,250,259,278 Staking.sol#L233,259,287,297,341,351,369,379,390,413,449,461,466 sOlympusERC20.sol#L160,176,183,187,196,201,205,228,257,283,293,311,325,337,369,379,397,407,418,441,731,753,773,797,832,854 BondDepository.sol#L102,126,134,145,149,187,191,203,207,218,237,365,373,416,484,492,497,564 BeraRerserveFeeDistributor.sol#L82,90,101,105,143,147,159,163,174,193,218,222,230,235
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function isContract(address account) internal view returns
(bool) {
        // This method relies in extcodesize, which returns 0
for contracts in
        // construction, since the code is only stored at the
end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	Treasury.sol#L340
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(isReserveSpender[msg.sender] == true, "Not approved")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	wETHBondDepository.sol#L523,524,527,528,529,530,531,532,533,534,535 StandardBondingCalculator.sol#L14,15,18,19,20,21,22,23,24,25,26 BondDepository.sol#L524,525,529,530,531,532,533,534,535,536,537
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
d /= pow2
r *= 2 - d * r
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	StandardBondingCalculator.sol#L289 StakingDistributor.sol#L452 BeraReserveToken.sol#L221,230 BeraReservePreBondSale.sol#L140 BeraReserveLockUp.sol#L124,137,150
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 reserve
uint256 reward
 < count;
uint256 usdcToRefund
 < numOfM
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	VaultOwned.sol#L50,56,62 Staking.sol#L806 BondDepository.sol#L683
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
_vault = vault_
_lockUp = lockUp_
_staking = staking_
distributor = _address
bondCalculator = _bondCalculator
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	wOHM.sol#L288,427 wETHBondDepository.sol#L120,177,227,433 Treasury.sol#L53,79,99 StakingDistributor.sol#L163,220,268 Staking.sol#L211,325,403 sOlympusERC20.sol#L235,353,431,907 BondDepository.sol#L120,177,227,433 BeraRerserveFeeDistributor.sol#L76,133,183
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            size := extcodesize(account)
        }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }

assembly {
            chainID := chainid()
        }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	utils/BeraReserveTokenUtils.sol#L3
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	wOHM.sol#L949,971,981,997 wETHBondDepository.sol#L877 Treasury.sol#L374 StakingWarmup.sol#L103 StakingHelper.sol#L98,105 BondDepository.sol#L889 BeraReserveLockUp.sol#L311
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(OHM).transferFrom(msg.sender, address(this), _amount)
IERC20(OHM).transfer(msg.sender, value)
IERC20(sOHM).transferFrom(msg.sender, address(this), _amount)
IERC20(sOHM).transfer(msg.sender, value)
IERC20(OHM).transfer(_recipient, _amount)
IERC20(_token).transfer(msg.sender, _amount)
IERC20(sOHM).transfer(_staker, _amount)

nsfer(_member, amountUnlocked);

        e
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Inheritance Graph

For a detailed inheritance graph, please refer to the GitHub repository of the audit.

Flow Graph

For a detailed flow graph, please refer to the GitHub repository of the audit.

Summary

Bera Reserve contract implements a comprehensive system for token management, financial operations, staking, locking, rewards distribution, and vesting mechanisms. This audit investigates security issues, business logic concerns, and potential improvements to ensure the integrity and efficiency of the protocol's smart contracts.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io