SHERLOCK

Security Review For Bondi Finance

Collaborative Audit Prepared For:	**Bondi Finance**
Lead Security Expert(s):	**iamnmt**
	jennifer37
Date Audited:	**September 8 - September 15, 2025**
Final Commit:	**1508e56**

Introduction

Bondi Finance tokenizes publicly traded corporate bonds as fully fungible ERC-20 Bond Tokens, making fixed income previously reserved for institutions accessible on-chain. This audit focuses on the v2 contracts, including crowdfunding via the funding module, programatic refunds in case of failed funding, Bond Token distribution, AccountingToken mirroring, Merkle-based coupon distribution, and multichain deployment with local Bond Token issuance with zero bridges or cross-chain messaging.

Scope

Repository: ASMAK-Bilisim/bondi-contracts-audit-v2

Audited Commit: 7c6b42209b2e3fde82fbcc172bc2ac772caf72da

Final Commit: 1508e56108c826c44be0ffb2b40261ada7be610f

Files:

- contracts/AccountingToken.sol
- contracts/BondToken.sol
- contracts/Distribution.sol
- contracts/document-management/upgradeable/ERC1643Upgradeable.sol
- contracts/FpUSD.sol
- contracts/Funding.sol
- contracts/Handler.sol
- contracts/InvestorNFT.sol
- contracts/kyc/IKYCRegister.sol
- contracts/kyc/KYCRegister.sol
- contracts/upgrades/Proxies.sol

Final Commit Hash

1508e56108c826c44be0ffb2b40261ada7be610f

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	6	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Auditors' Note

The Bondi Finance v2 smart contracts have undergone a comprehensive security review. This release builds on a **production-proven foundation**: v1 successfully raised and deployed $205,000 on Base Mainnet, completed a full bond lifecycle, and repaid investors with principal plus interest.

Core Security Strengths

- **Zero Bridge Risk Multichain Design**
  Funds never leave their origin chain. Cross-chain coordination is handled via mirrored AccountingTokens and an off-chain watcher, ensuring global target tracking without exposing assets to bridge exploits
- **Battle-Tested Financial Flows**
  The same funding, extraction, bond purchase, and coupon/principal repayment logic proven in production is preserved in v2, now enhanced with FpUSD receipts and Merkle-based distributions
- **Defense-in-Depth Role Architecture**
  Access control follows the principle of least privilege with strict separation:
  - Multisig Gnosis Safe holds all admin and fund deposit powers

  - Backend services (Watcher, Orchestrator, Relayer) are limited to narrow, non-financial roles

- **Verifiable Merkle Distributions**
  Incentives and coupons are consolidated into a single Merkle root, which anyone can independently reconstruct to verify accuracy. Users claim through Merkle proofs, while Bondi's relayer handles automatic claims for KYC-verified users to ensure a seamless experience.

- **Critical fixes applied and reviewed**
  - Block-level investment restriction
  - Cooldown enforcement
  - Immutable funding parameters
  - AccountingToken burn safeguard against overflow-DoS
  - KYC bypass prevention via automatic bond token blacklisting when KYC is revoked
  - Cross-chain funding sync grace period (6-hour cooldown)
  - Token donation DoS prevention via tracked investment amounts

Security Assessment Following these fixes, the contracts now present zero critical vulnerabilities. All risks have been understood and mitigated, operational responsibilities are tightly controlled, and every core financial flow has already been validated on-chain. The codebase is regarded as production-ready.

Issue H-1: Funding sync may be blocked

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/18

Summary

Funding sync may be blocked

Vulnerability Detail

Bondi will be deployed on multiple chains, including mantle, base, injective and Plume. Once we invest funds on one chain, the watcher role will monitor the related event and trigger `mirrorDeposit` on all other chains.

We will generate one unique id according to the `srcChainId, investor, timestamp`. And we don't allow to use the same `id` twice to prevent the possible replay attack.

The problem here is that users may invest multiple times on the same chain, the same block. The watcher role will generate the same id for different transactions, and we will fail to sync the fund amount among different chains.

```
/*
 *          // 84 bytes packed (32 + 20 + 32) → keccak256   bytes32
 *          bytes32 depositId = keccak256(
 *              abi.encodePacked(
 *                  uint256(srcChainId),    // EIP-155 chainId
 *                  address(investor),      // depositor
 *                  uint256(timestamp)      // unix-seconds
 *              )
 *          );
 * /
function mirrorDeposit(uint256 amt, bytes32 id, uint256 srcChain) external
 ↪  onlyRole(WATCHER_ROLE) {
    // Gas optimization: return early if amount is zero
    if (amt == 0) return;
    if (processed[id]) revert DepositAlreadyProcessed(id);
    processed[id] = true;
    // Here the amt is 6 decimal.
    accountingToken.mint(address(funding), amt);
    // Emit event for off-chain tracking
    emit DepositMirrored(amt, id, srcChain);
}
```

Impact

Fail to sync the fund amount among different chains.

Code Snippet

https://github.com/sherlock-audit/2025-09-bondi-finance/blob/10762cbcccb99097cb1d26562bd6ac19acea73d0/bondi-contracts-audit-v2.0.0/contracts/Handler.sol#L174-L182

Tool Used

Manual Review

Recommendation

1.Add a nonce for each investor, and the nonce should be increased on invest, and the nonce should be used to derive the depositId 2. Prevent investor to calls invest in a same block

Issue M-1: Direct token donation enables denial of service on investment function

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/12

Relevant Context

The `Funding` contract allows investors to contribute stablecoin tokens through the `Funding#invest` function during the funding phase. The contract tracks the total raised amount across multiple chains using the `totalRaised()` function, which sums the local stablecoin balance and accounting tokens representing cross-chain deposits. The contract enforces a strict target amount limit through the `checkInvestmentRequirements` modifier, which reverts with `FundingExceedsTargetAmount` if the investment would cause the total to exceed the target.

Finding Description

The Funding#invest function is vulnerable to denial of service attacks through direct token donations to the contract. The vulnerability stems from the target amount validation in the checkInvestmentRequirements modifier, which checks `if (totalRaised() + investmentAmount_ > targetAmount)` and reverts if the condition is true.

The `totalRaised()` function calculates the total by summing `usdToken.balanceOf(address(this)) + accountingToken.balanceOf(address(this))`. Since anyone can directly transfer stablecoin tokens to the contract address without going through the `invest` function, an attacker can artificially inflate the `usdToken.balanceOf(address(this))` component.

When the total raised amount approaches the `targetAmount`, an attacker can send as little as 1 wei of stablecoin directly to the contract. This increases the `totalRaised()` value, causing subsequent legitimate `invest` calls to fail when `totalRaised() + investmentAmount_ > targetAmount` becomes true. The attack is particularly effective when the remaining investment capacity is small, as the attacker can prevent all future investments with minimal cost.

Impact

Legitimate investors cannot complete their investments when the funding round is near completion. The attacker spends only the gas cost and 1 wei of stablecoin to block all remaining investment attempts, creating a complete denial of service for the investment functionality.

Proof of Concept

1. The funding round is ongoing with `targetAmount = 1,000,000 USDC` and `totalRaised() = 999,999 USDC`
2. Alice attempts to invest the remaining 1 USDC by calling `Funding#invest(1000000)` (1 USDC in 6 decimals)
3. Before Alice's transaction is mined, an attacker directly transfers 1 wei of USDC to the contract address using `usdToken.transfer(fundingContract, 1)`
4. The attacker's transaction increases `usdToken.balanceOf(address(this))` from 999,999 USDC to 999,999.000001 USDC
5. When Alice's transaction executes, the `checkInvestmentRequirements` modifier calculates `totalRaised() + investmentAmount_ = 999,999.000001 + 1 = 1,000,000.000001 > 1,000,000`
6. Alice's transaction reverts with `FundingExceedsTargetAmount`, preventing her from investing
7. The attacker can repeat this process to block any future investment attempts for the cost of 1 wei plus gas

Tool Used

Manual Review

Recommendation

Implement a mechanism to handle partial investments when the requested amount would exceed the target. Modify the `invest` function to allow `investmentAmount_ = type(uint256).max` as a special value, automatically calculating the actual investment amount as `targetAmount - totalRaised()`. This ensures that legitimate investors can always invest up to the remaining capacity, regardless of small direct donations to the contract.

```
function invest(uint256 investmentAmount_) public
↪ checkInvestmentRequirements(investmentAmount_) nonReentrant
↪ onlyWhenFundingOngoing whenNotPaused {
    if (!kycRegister.hasCompletedKYC(msg.sender)) revert
    ↪ CannotInvestWithoutKYC(msg.sender);

    // Handle maximum investment request
    uint256 actualInvestmentAmount = investmentAmount_;
    if (investmentAmount_ == type(uint256).max) {
        actualInvestmentAmount = targetAmount - totalRaised();
    }

    usdToken.safeTransferFrom(msg.sender, address(this), actualInvestmentAmount);
    fpUSDToken.mint(msg.sender, actualInvestmentAmount * 1e12);
    _updateInvestorWithInvestment(msg.sender, actualInvestmentAmount);
    emit InvestmentMade(msg.sender, actualInvestmentAmount, block.timestamp);
}
```

Additionally, update the `checkInvestmentRequirements` modifier to handle the special case:

```
modifier checkInvestmentRequirements(uint256 investmentAmount_) {
    // ... existing checks ...
    if (investmentAmount_ != type(uint256).max && totalRaised() + investmentAmount_
    ↪   > targetAmount) {
        revert FundingExceedsTargetAmount(totalRaised(), targetAmount);
    }
    // ... existing checks ...
    _;
}
```

Issue M-2: Blacklisted investor in sequential refund process leads to denial of service for all subsequent refunds

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/13

Relevant Context

The `Funding` contract implements a refund mechanism through the `refundInvestors` function that processes multiple investor refunds in a single transaction. The contract uses USDC as the funding token, which implements a blacklisting mechanism controlled by Centre Consortium. When an address is blacklisted by USDC, any transfer to that address will revert, causing the entire transaction to fail.

Finding Description

The Funding#refundInvestors function processes refunds sequentially in a loop without proper error handling for individual transfer failures. The vulnerability occurs in the second loop where the function executes usdToken.safeTransfer(investorsToRefund[i], refundAmounts[i]) for each investor. Since USDC implements blacklisting functionality where transfers to blacklisted addresses revert, a single blacklisted investor in the refund queue will cause the entire `refundInvestors` transaction to revert.

When the `safeTransfer` call encounters a blacklisted address, the entire transaction reverts. This means that the refund mechanism becomes completely unusable as long as any blacklisted address remains in the investors array.

The root cause is the lack of error handling around the `usdToken.safeTransfer` call, which does not account for the possibility of individual transfer failures due to USDC's blacklisting mechanism.

Impact

The protocol suffers a denial of service where the function `refundInvestors` becomes unusable. Legitimate investors cannot receive their refunds when a blacklisted address exists anywhere in the refund queue, effectively blocking the entire refund process.

Tool Used

Manual Review

Recommendation

Implement a try-catch mechanism around the `usdToken.safeTransfer` call to handle individual transfer failures gracefully:

```
// Then refund the investors
for (uint256 i = 0; i < amountOfInvestorsToRefund; i++) {
    fpUSDToken.burn(investorsToRefund[i], refundAmounts[i] * 1e12);

    try usdToken.safeTransfer(investorsToRefund[i], refundAmounts[i]) {
        // Transfer successful, continue to next investor
    } catch {
        // Transfer failed (e.g., blacklisted address)
        // Log the failure and continue processing other investors
        emit RefundFailed(investorsToRefund[i], refundAmounts[i]);

        // Remint the fpUSD tokens since transfer failed
        fpUSDToken.mint(investorsToRefund[i], refundAmounts[i] * 1e12);
    }
}
```

This approach ensures that failed transfers to USDC-blacklisted addresses do not prevent legitimate investors from receiving their refunds, maintaining the protocol's availability and allowing the refund process to continue for non-blacklisted addresses.

Issue M-3: Unclaimed bond tokens in distribution contract leads to permanent loss of earned coupon payments

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/14

This issue has been acknowledged by the team but won't be fixed at this time.

Relevant Context

The Bondi Finance protocol operates through a multi-phase bond distribution mechanism:

1. Bond tokens are initially minted to the `Distribution` contract via `Handler#emitBonds`
2. Users claim bond tokens by calling `Distribution#claimBonds`, which burns their FpUSD tokens and transfers bond tokens from the Distribution contract
3. Coupon distributions are calculated based on bond token balances at the time `Distribution#depositCoupon` is called, with the snapshot block number recorded in `CouponMeta.blockNumber`
4. The orchestrator backend service calculates Merkle tree distributions based on these snapshots and calls `Distribution#finaliseCoupon` to set the Merkle root
5. Users claim their proportional coupon payments via `Distribution#claimCoupon` or `Distribution#claimCouponForUser`

Finding Description

The `Distribution` contract doesn't account for scenarios where users have not claimed their bond tokens before coupon distribution snapshots occur. When `depositCoupon` is called, the snapshot includes all bond token balances at that block, including any unclaimed bonds still held by the `Distribution` contract itself.

The orchestrator service calculates coupon distributions proportionally based on bond token holdings, meaning the `Distribution` contract becomes entitled to coupon payments for any unclaimed bonds in its balance. However, the contract lacks any mechanism to sweep or recover these earned `usdToken` (coupon) payments, causing them to become permanently locked within the contract.

This issue is exacerbated by the fact that the `Distribution` contract cannot claim its own coupon allocation through normal means, as `claimCoupon` requires KYC validation via `kycRegister.hasCompletedKYC(msg.sender)`, and the contract address will not have completed KYC procedures.

Impact

Protocol users suffer loss of coupon payments proportional to the amount of unclaimed bond tokens at the time of each coupon distribution. The `Distribution` contract permanently traps these coupon funds, with no recovery mechanism available. In scenarios where a significant portion of bonds remain unclaimed (due to user inactivity, lost access, or delayed claiming), the impact could be substantial across multiple coupon periods.

Tool Used

Manual Review

Recommendation

Implement one of the following solutions:

Option 1: Modify coupon distribution calculation Update the implementation specification to exclude the `Distribution` contract from coupon calculations in the orchestrator backend service. This ensures unclaimed bonds do not earn coupons.

Option 2: Add administrative coupon recovery Modify `claimCouponForUser` to allow claiming coupons earned by the Distribution contract:

```
function claimCouponForUser(address user, uint256 couponId, uint256 amount,
 ↪  bytes32[] calldata proof) external onlyRole(RELAYER_ROLE) nonReentrant
 ↪  whenNotPaused {
    if (coupons[couponId].root == bytes32(0)) revert NotFinalized();
    if (claimed[couponId][user]) revert AlreadyClaimed();

    // Verify Merkle proof
    bytes32 leaf = keccak256(abi.encodePacked(user, amount));
    if (!MerkleProof.verify(proof, coupons[couponId].root, leaf)) revert
     ↪  InvalidProof();

    claimed[couponId][user] = true;

    // Skip KYC validation for Distribution contract and send to protocol treasury
    if (user == address(this)) {
        usdToken.safeTransfer(protocolTreasury, amount);
    } else {
        if (!kycRegister.hasCompletedKYC(user)) revert NoKYC(user);
        usdToken.safeTransfer(user, amount);
    }

    emit CouponClaimed(user, couponId, amount);
}
```

Issue M-4: Direct token transfer manipulation leads to asymmetric funding states across chains

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/16

Relevant Context

The Bondi protocol operates across multiple blockchains with a synchronized funding mechanism. The `Funding` contract uses two token systems to track investments:

1. `FpUSD` tokens: Minted to investors at a 1:1e12 ratio when they call `Funding#invest`
2. `AccountingToken`: Minted by the `Handler` contract via `Handler#mirrorDeposit` to represent investments made on other chains

The `Funding#totalRaised` function calculates the total funding across all chains by summing the contract's `usdToken` balance with its `AccountingToken` balance. This total is used to determine whether the funding target has been reached and controls functionality like fund extraction and investor refunds.

Finding Description

The Funding#totalRaised function includes all `usdToken` held by the contract, regardless of how those tokens arrived. The current implementation sums `usdToken.balanceOf(address(this))` with `accountingToken.balanceOf(address(this))`, treating any direct token transfers as legitimate investments.

An attacker can exploit this by directly transferring `usdToken` to the `Funding` contract on one chain, artificially inflating the `totalRaised` calculation on that chain. This creates an asymmetric state where different chains report different funding levels, even though only legitimate investments (tracked via `invest` calls) should be mirrored across chains.

The vulnerability occurs because the `totalRaised` calculation does not distinguish between tokens received through legitimate `Funding#invest` calls and tokens received through direct transfers. Only tokens from `invest` calls are guaranteed to be mirrored to other chains via the `Handler#mirrorDeposit` mechanism, as they trigger `FpUSD` minting which serves as the canonical record of legitimate investments.

Impact

Legitimate investors suffer fund recovery complications when the funding target is not reached globally. The attacker causes a subset of investors to become unable to withdraw their funds through normal mechanisms, forcing manual administrative intervention.

Proof of Concept

Consider a two-chain deployment with chains A and B, where the `targetAmount` is 1,000,000 USDC:

1. **Initial State:** Chain A has 400,000 USDC in legitimate investments, Chain B has 590,000 USDC in legitimate investments. Both chains show `totalRaised()` = 990,000 USDC (400,000 local + 590,000 mirrored or vice versa).
2. **Attack Execution:** An attacker directly transfers only 10,000 USDC to the `Funding` contract on chain B, bypassing the `invest` function.
3. **Asymmetric State:**
   - Chain A: `totalRaised()` = 990,000 USDC (still below target)
   - Chain B: `totalRaised()` = 1,000,000 USDC (590,000 + 400,000 + 10,000 donated)
4. **Funding Period Ends:** When the funding period expires without additional legitimate investments:
   - Chain A: Target not reached, investors can call `Funding#withdraw` or `Funding#refundInvestors`
   - Chain B: Target artificially reached, `Funding#_checkGeneralRefundRequirements` reverts with `FundingTargetAmountReached()`
5. **Fund Recovery Issue:** Legitimate investors on chain B cannot recover their 590,000 USDC through normal withdrawal mechanisms. The only recovery path requires administrative intervention via `Funding#extractFunds`, followed by manual distribution to investors.

This attack is particularly effective when the funding is very close to the target, requiring only a small donation to tip one chain over the threshold while others remain below it.

Tool Used

Manual Review

Recommendation

Modify the `Funding#totalRaised` function to only count funds that were received through legitimate investment calls. Since `FpUSD` tokens are minted to investors at a 1:1e12 ratio during legitimate investments, the calculation should use the total supply:

```
function totalRaised() public view returns(uint256) {
    return (fpUSDToken.totalSupply() / 1e12) +
    ↪  accountingToken.balanceOf(address(this));
}
```

This ensures that only tracked investments (those that generate `FpUSD` tokens and can be mirrored across chains) contribute to the funding target calculation, preventing manipulation through direct token transfers. The `fpUSDToken.totalSupply()` accurately represents the sum of all legitimate investments made through the `Funding#invest` function across the funding period.

Issue M-5: Malicious users can block extractFunds

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/17

Summary

Malicious users can block extractFunds

Vulnerability Detail

When users invest funds via bondi, we have some security checks for `invest` function. The invest amount cannot be less than `minimumInvestmentAmount` and we cannot exceed the `targetAmount`.

The problem here is that malicious users can make the remaining invest amount less than `minimumInvestmentAmount`. This will cause that nobody cannot invest into this funding. Because we don't reach the `targetAmount`, the bondi team cannot extract these funds.

```
modifier checkInvestmentRequirements(uint256 investmentAmount_) {
    // If the funding is on-going, then we can invest.
    if (currentFundingPhase != FundingPhase.Ongoing) revert
     ↪  FundingInInvalidPhase(uint(currentFundingPhase));
    // whaleNFT and ogNFT must exist.
    if (address(whaleNFT) == address(0) || address(ogNFT) == address(0)) revert
     ↪  InvestorNFTCannotBeZeroAddress();
    if (fundingPeriodLimit <= block.timestamp) revert
     ↪  FundingPeriodFinished(fundingPeriodLimit);
    if (totalRaised() + investmentAmount_ > targetAmount) revert
     ↪  FundingExceedsTargetAmount(totalRaised(), targetAmount);
    if (investmentAmount_ < minimumInvestmentAmount) revert
     ↪  FundingBelowMinimum(investmentAmount_, minimumInvestmentAmount);
    uint256 investorBalance = usdToken.balanceOf(msg.sender);
    // make sure that the msg.sender has enough balance to invest.
    if (investorBalance < investmentAmount_) revert
     ↪  InvestorBalanceInsufficient(investorBalance, investmentAmount_);
    _;
}
```

Impact

This will cause that users may fail to invest to meet the `targetAmount`.

Code Snippet

https://github.com/sherlock-audit/2025-09-bondi-finance/blob/10762cbcccb99097cb1d26562bd6ac19acea73d0/bondi-contracts-audit-v2.0.0/contracts/Funding.sol#L89-L98

Tool Used

Manual Review

Recommendation

Consider to allow investing any amount for the last investor. Or we don't allow the remaining invest amount is less than `minimumInvestmentAmount` after one investment.

Issue M-6: Final raised funds may be less than target amount

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/19

Summary

Final raised funds may be less than target amount

Vulnerability Detail

In Bondi, investors can invest funds into contracts in the funding period. If the funding period is expired, then we reach the target funding amount, then the admin role will extract funds. Otherwise, investors can withdraw their assets.

The problem here is that the actual funding amount is async among chains. It's possible that we meet the target amount on one chain and don't meet the target amount on another chain because we don't sync the deposit in other chain timely.

For example:

1. fundingPeriodLimit is timestamp X. The target amount is 100 USD.
2. Current raised amount is 95 USD.
3. In the last block before timestamp X, Alice invests 5 USD on chain A to reach the target. Because we have a little bit time to sync the 5 USD to chain B.
4. It's quite possible that the total raised fund is 95 USD in timestamp X on Chain B.
5. Then investors can withdraw their funds on ChainB. After one withdraw, e.g 6 USD, current total raised funds in Chain B is 89 USD.

```
modifier checkWithdrawalRequirements {
    _checkGeneralRefundRequirements();
    Investor memory currentInvestor = investorByAddress[msg.sender];
    uint256 amountToWithdraw = currentInvestor.investedAmount;
    if (amountToWithdraw == 0) revert FundingNoRegisteredFunds();
    uint256 fundingBalance = usdToken.balanceOf(address(this));
    if (fundingBalance < amountToWithdraw) revert
     ↪  FundingBalanceInsufficient(fundingBalance, amountToWithdraw);
    _;
}
```

Impact

The final raised fund amount may be less than expected amount.

Code Snippet

https://github.com/sherlock-audit/2025-09-bondi-finance/blob/10762cbcccb99097cb1d26562bd6ac19acea73d0/bondi-contracts-audit-v2.0.0/contracts/Funding.sol#L103-L111

Tool Used

Manual Review

Recommendation

Add a grace period after the fundingPeriodLimit, and users can withdraw the fund back when after fundingPeriodLimit + gracePeriod.

Issue L-1: Redundant conditional logic in pause bypass mechanism leads to unnecessary gas consumption

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/15

Finding Description

The BondToken#_update function implements a redundant conditional branch that unnecessarily increases gas consumption for all transfer operations. The function contains logic to handle admin operations during paused states by conditionally calling either `ERC20Upgradeable._update` or `super._update` based on whether the operation is performed by an admin during a paused state.

The current implementation checks:

```
if (isAdminOperation && paused()) {
    ERC20Upgradeable._update(from, to, value);
} else {
    super._update(from, to, value);
}
```

However, this conditional logic is redundant because pause validation occurs earlier in the function. For non-admin operations, _requireNotPaused() is called at the beginning, which will revert if the contract is paused. This means that if the function execution reaches the conditional block, the contract is either not paused or the operation is being performed by an admin.

The `super._update` call would invoke `ERC20PausableUpgradeable._update`, which has a `whenNotPaused` modifier. Since the pause check has already been performed for non-admin operations, and admin operations are intended to bypass pause restrictions, the function can safely call `ERC20Upgradeable._update` directly in all cases without the conditional check.

The root cause is the implementation of unnecessary branching logic that duplicates pause validation already performed earlier in the function flow. This results in additional gas overhead for every token transfer operation due to the redundant conditional check and the `paused()` state query.

Tool Used

Manual Review

Recommendation

Simplify the `_update` function by removing the conditional logic and calling `ERC20Upgradeable._update` directly. This optimization is safe because:

1. Non-admin operations are already validated for pause state via `_requireNotPaused()`
2. Admin operations are intended to bypass pause restrictions
3. The function will not reach this point if pausing should prevent the operation

```
function _update(address from, address to, uint256 value) internal
 ↪  override(ERC20Upgradeable, ERC20PausableUpgradeable) {
    // Check if this is an admin operation
    bool isAdminOperation = hasRole(DEFAULT_ADMIN_ROLE, msg.sender);

    // Admin can extract tokens from blacklisted addresses but not send them
    if (!isAdminOperation) {
        _requireNotPaused();
        if (isBlacklistedAddress[from]) revert BlacklistedAddress(from);
    }
    if (isBlacklistedAddress[to]) revert BlacklistedAddress(to);

    // Update bond balances tracking
    if (from != address(0)) {
        uint256 currentSenderAmount = EnumerableMap.get(_bondBalanceByHolder, from);
        EnumerableMap.set(_bondBalanceByHolder, from, currentSenderAmount - value);
    }
    (,uint256 currentReceiverAmount) = EnumerableMap.tryGet(_bondBalanceByHolder,
     ↪  to);
    EnumerableMap.set(_bondBalanceByHolder, to, currentReceiverAmount + value);

-   // For admin operations when paused, bypass the pausable check by calling
 ↪  ERC20Upgradeable._update directly
-   if (isAdminOperation && paused()) {
-       // Call the base ERC20Upgradeable._update directly to bypass pause checks
-       ERC20Upgradeable._update(from, to, value);
-   } else {
-       // Normal flow - call all parent _update methods
-       super._update(from, to, value);
-   }
+   // Call ERC20Upgradeable._update directly since pause validation is handled
 ↪  above
+   ERC20Upgradeable._update(from, to, value);
}
```

This optimization reduces gas consumption for all transfer operations while maintaining the same security guarantees and functional behavior.

Issue L-2: Users can bypass the KYC check to redeem the principles

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/20

Summary

Users can bypass the KYC check to redeem the principles

Vulnerability Detail

In Distribution contract, users can claim their principles after the principle distribution. In `redeemPrincipal`, we will not allow non-KYC users to redeem principles.

The problem here is that users will redeem principles via holding bond tokens. If one user's KYC is revoked, then this user can transfer his bond token to another KYC accountant and withdraw principle directly.

```
function redeemPrincipal(uint256 burnAmt) external nonReentrant whenNotPaused {
    if (!principalSet) revert PrincipalNotSet();
    if (!principalDistributionStarted) revert PrincipalDistributionNotStarted();

    // Validate burn amount
    uint256 bal = bondToken.balanceOf(msg.sender);
    if (burnAmt == 0 || bal < burnAmt) revert InvalidBurnAmount();

    if (!kycRegister.hasCompletedKYC(msg.sender)) revert NoKYC(msg.sender);
```

Impact

Users who's KYC is revoked can still redeem principles.

Code Snippet

https://github.com/sherlock-audit/2025-09-bondi-finance/blob/10762cbcccb99097cb1d26562bd6ac19acea73d0/bondi-contracts-audit-v2.0.0/contracts/Distribution.sol#L364-L382

Tool Used

Manual Review

Recommendation

Don't allow bond token transfer if this user's KYC is revoked.

Issue L-3: Missing initialization for ERC1643 contract

Source: https://github.com/sherlock-audit/2025-09-bondi-finance/issues/21

Summary

Missing initialization for ERC1643 contract

Vulnerability Detail

In BondToken, we will inherit `ERC1643Upgradeable` contract. In contract `ERC1643Upgradeable`, we wish to trigger `__ERC1643_init` to finish the related initialization work for `ERC1643Upgradeable`. But we fail to trigger this in BondToken.

Because the `__ERC1643_init_unchained` is one empty function. So there is not any security issue, just one best practice.

```
function initialize(
    string memory bondName,
    string memory bondSymbol,
    address defaultAdmin,
    address pauser,
    address minter
) initializer public {
    __ERC20_init(bondName, bondSymbol);
    __ERC20Burnable_init();
    __ERC20Pausable_init();
    __AccessControl_init();
    __Ownable_init(defaultAdmin);
    __ERC20Permit_init(bondName);

    _grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
    _grantRole(PAUSER_ROLE, pauser);
    _grantRole(EMITTER_ROLE, minter);

    _supplyMinted = false;
}
function __ERC1643_init() internal onlyInitializing {
    __ERC1643_init_unchained();
}
```

Impact

Best practice for initialization.

Code Snippet

https://github.com/sherlock-audit/2025-09-bondi-finance/blob/10762cbcccb99097cb1d26562bd6ac19acea73d0/bondi-contracts-audit-v2.0.0/contracts/BondToken.sol#L49-L68

Tool Used

Manual Review

Recommendation

```
    function __ERC1643_init(address initialOwner) internal onlyInitializing {
+        __Ownable_init(initialOwner);
        __ERC1643_init_unchained();
    }
```

```
    function initialize(
        string memory bondName,
        string memory bondSymbol,
        address defaultAdmin,
        address pauser,
        address minter
    ) initializer public {
        __ERC20_init(bondName, bondSymbol);
        __ERC20Burnable_init();
        __ERC20Pausable_init();
        __AccessControl_init();
-        __Ownable_init(defaultAdmin);
+    __ERC1643_init(defaultAdmin);
        __ERC20Permit_init(bondName);

        _grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
        _grantRole(PAUSER_ROLE, pauser);
        _grantRole(EMITTER_ROLE, minter);

        _supplyMinted = false;
    }
```

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.