BLOCKAPEX

# SMART CONTRACT SECURITY

V 1.0

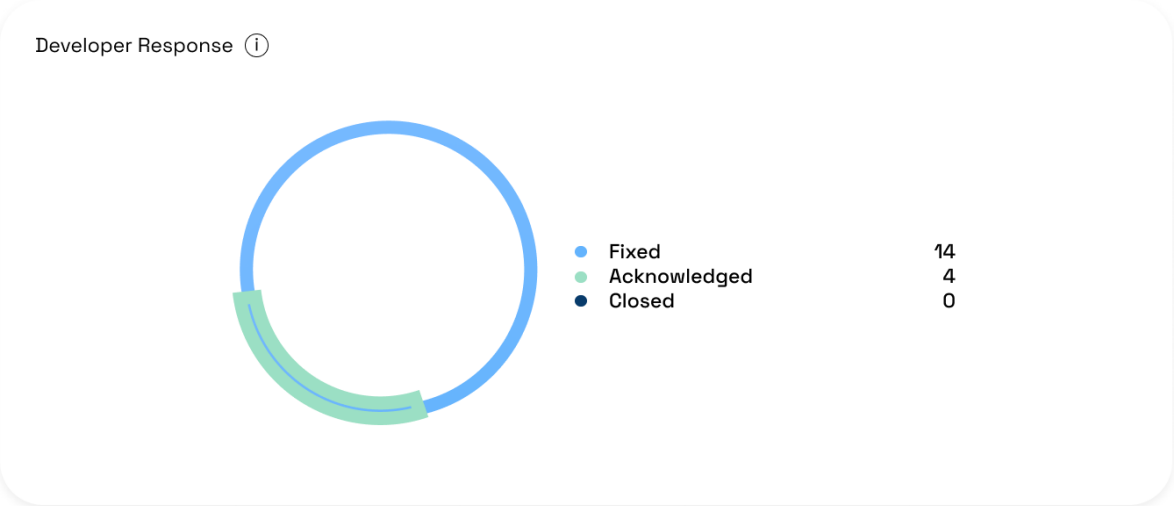
DATE: 10th FEB 2024

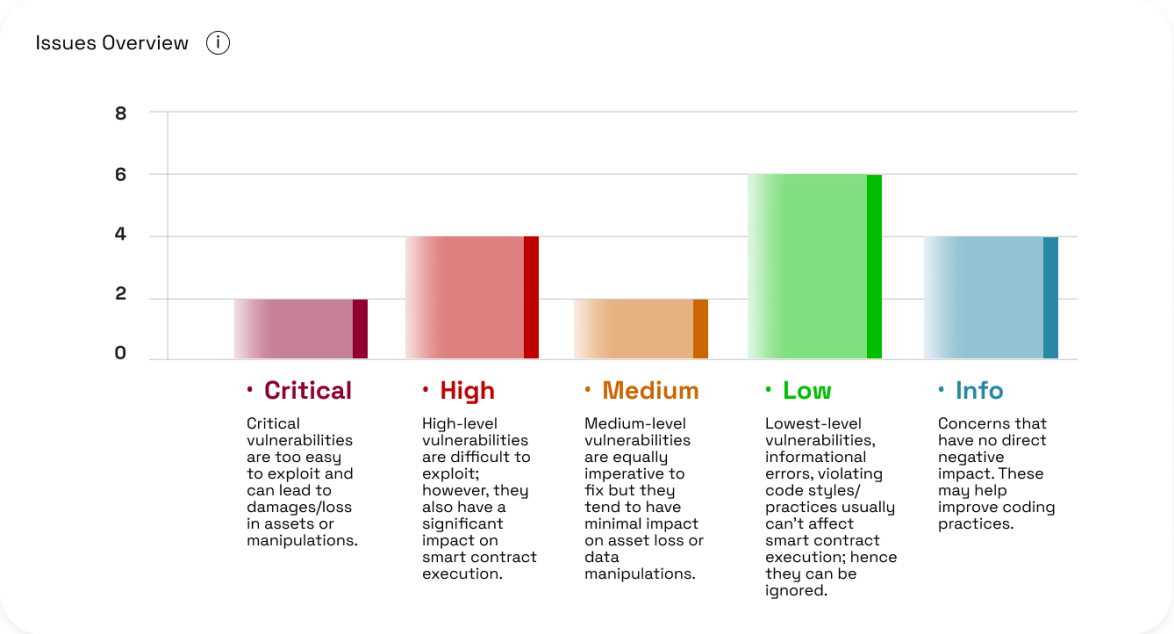
PREPARED FOR: AMET FINANCE

# Contents

# 1 Executive Summary

Our team performed a technique called Filtered Audit, where three individuals separately audited the Amet Finance Smart Contracts. After a thorough and rigorous manual testing process involving line by line code review for bugs, an automated tool-based review was carried out. All the raised flags were manually reviewed and re-tested to identify any false positives.



## 1.1 Scope

### 1.1.1 In Scope

**Overview of the Contracts:** Amet Finance is a DeFi platform with an easy-to-use interface that allows individuals, businesses, and crypto projects to securely and efficiently trade, issue, and manage on-chain bonds. The contracts under audit are designed for EVM Compatible blockchains primarily focusing on Vault Mechanism , Issuance and Management of Zero Coupon Bonds. It serves as a decentralized finance (DeFi) application allowing Bond issuers to issue ZCBs with flexible configs and allow users to buy ZCBs. Users can flexible redeem their bonds during and after maturity period. Issuer can raise investments by issuing bonds and users can warn lucrative interests on the bonds.

**Contracts in Scope:**

1. AmetVault.sol
2. ZeroCouponBonds.sol
3. ZeroCouponBondsIssuer.sol

**Initial Commit Hash:** f4c387b88e95a2ac3ee44805f410ac9e154da7ed
**Final Commit Hash:** bf6cabc0f67054c2ba1b219fa6cffce1b0087030

### 1.1.2 Out of Scope

Any features or functionalities not explicitly mentioned in the “In Scope” section are also considered outside the scope of this audit.

## 1.2 Methodology

The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process for a time spanning 1.5 weeks. Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations,software, and security design patterns, code styles and best practices

**Project Coverage**

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. The coverage report generated by running ***npx hardhat coverage*** provides a detailed overview of the testing coverage across various contracts within the project. Below is a summary of key findings.

**Contracts Overview:**

**AmetVault.sol**
**Lines Covered:** A small percentage of 18.18% lines are covered , indicating basic testing has been performed.
**Branch Coverage:** There's minimal branch coverage of 16.67%, suggesting many conditional paths remain untested.

**ZeroCouponBonds.sol**
**Lines Covered:** Moderate coverage observed with 52.11% of lines covered.
**Branch Coverage:** Indicates some testing of conditional logic but many branches remain untested. An overall of 29.41% of branches were found to be covered.

**ZeroCouponBondsIssuer.sol**
**Lines Covered:** A good coverage was observed here with 100% lines covered.
**Branch Coverage:** Some branches tested, but gaps in coverage suggest not all logical paths are verified. An overall of 71.43% of branches were covered.

**Conclusion**
The current test coverage indicates a preliminary level of testing that covers basic functionalities of some contracts. However, to ensure the security and reliability of the smart contracts, it is crucial to significantly expand the testing framework to achieve comprehensive coverage across all contracts, functions, and branches. This will not only help in identifying and mitigating potential vulnerabilities but also in enhancing the overall integrity of the smart contract system.

**Questions for Security Assessment**

1. Are all inputs and state-changing operations validated and sanitized to prevent invalid operations?
2. Could the Bond Issuer, with elevated privileges, manipulate the system in a way that could harm users or the protocol's integrity?
3. Are there safeguards against potential malicious actions by the Bond Issuer, such as issuing an excessive number of bonds or manipulating fees?
4. Are only Owner and similar access control modifiers used correctly and securely across the system to safeguard against unauthorized actions?
5. Is the contract logic designed to ensure that all funds can be accounted for and withdrawn under appropriate conditions?
6. Does the system accurately calculate interest, fees, and rewards, ensuring fairness and correctness in financial transactions?
7. Are there checks to prevent arithmetic overflows and underflows, particularly in balance and fee calculations?
8. Has the system been tested against edge cases and extreme scenarios to ensure robustness and reliability?
9. Are there any vulnerabilities or design flaws that could lead to direct financial loss for users or the protocol?
10. Is it possible for users to manipulate transactions or state to unjustly benefit at the expense of others?
11. Does the system employ safe mathematical operations to prevent overflows and underflows, especially in financial computations?
12. Can a user exploit the system to purchase more bonds than entitled or at a lower price than specified?
13. Does the system maintain consistent state across transactions, even in complex sequences of interactions?
14. Are reentrancy guards in place to prevent attacks that exploit contract calls to re-enter and manipulate state?

**Status Description**

**Acknowledged:** The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

**Fixed:** The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

**Closed:** This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. The team values the input and understands the importance of each reported issue. While no immediate action is being taken, the feedback provided is greatly appreciated and will inform future developments and decisions.

## 1.3 Summary of Findings Identified

| S.No | Severity | Findings | Status |
|---|---|---|---|
| #1 | CRITICAL | Unintended Fee Deduction Post-Maturity Due to Misapplied Capitulation Flag | FIXED |
| #2 | CRITICAL | Inflexible Vault Address Update Mechanism Exposes Funds To Risks | FIXED |
| #3 | HIGH | Referrals Can Never Claim the Rewards | FIXED |
| #4 | HIGH | Issuance Fee Handling Leads To Potential Under Payment Of Issuance Fees. | FIXED |
| #5 | HIGH | Lack of Token Standards Enforcement Risks Protocol Fee Integrity | ACKNOWLEDGED |
| #6 | HIGH | Self Referral In Bond Purchase Allows For Unintended Rewards | FIXED |
| #7 | MEDIUM | Missing Enforcement to Decrease Supply Post Settlement | FIXED |
| #8 | MEDIUM | Uninitialized Vault Address Risks Bond Functionality and Financial Losses | ACKNOWLEDGED |
| #9 | LOW | Missing 2 Step Transfer Ownership Pattern | FIXED |
| #10 | LOW | Uninitialized Referrer Fee Percentage Nullifies Referral Rewards | FIXED |
| #11 | LOW | Ownership Renouncement Locks Contract Management | FIXED |
| #12 | LOW | Missing Input Validations | FIXED |

| S.No | Severity | Findings | Status |
|---|---|---|---|
| #13 | LOW | Missing Validation On Fees Can Lead To Loss Of Bonds Issuer Funds | ACKNOWLEDGED |
| #14 | LOW | Lack of Reentrancy Protection In Critical Functions | FIXED |
| #15 | INFO | Missing Zero Count Validation in Bond Minting Process | ACKNOWLEDGED |
| #16 | INFO | Logic For Pausing Can Be Optimized | FIXED |
| #17 | INFO | Use Custom Errors For Gas Optimization | FIXED |
| #18 | INFO | Separate require Statements Can Optimize Gas | FIXED |

# 2 Findings and Risk Analysis

## 2.1 Unintended Fee Deduction Post-Maturity Due to Misapplied Capitulation Flag

**Severity:** Critical

**Status:** Fixed

**Description** The redeem function in the smart contract allows users to redeem their bonds. A critical aspect of this function is the handling of the `isCapitulation` flag, particularly in relation to the `maturity_period` of the bonds.

Before Maturity (purchasedBlock + bondInfoTmp.maturityPeriod > block.number): If isCapitulation is false, the function reverts with OperationFailed(OperationCodes.RedemptionBeforeMaturity), disallowing early redemption. However, if isCapitulation is true, this check is bypassed, allowing early redemption but applying an earlyRedemptionFeePercentage.

After Maturity: The maturity check doesnt affect redemption, regardless of the isCapitulation flag's state. If isCapitulation is true, the earlyRedemptionFeePercentage still applies, which could be an unintended consequence for post-maturity redemptions. It was observed that there is a time window based on the maturity period for which this issue is valid. If the maturityPeriod is set to 6 months the time window is 4 days.

**Location** `ZeroCouponBonds.sol`

**Impact** Charging an earlyRedemptionFeePercentage post-maturity (if isCapitulation is true) might be an unintended feature, potentially leading to loss of User's funds.

**Recommendation** Modify the contract to ensure that the earlyRedemptionFeePercentage is applied only before duration: bool isPostMaturity = block.number >= purchasedBlock + bondInfoTmp.maturityPeriod;

```
    // -- Code snip --
    // Apply early redemption fee only if isCapitulation is true and it&#x27;s before maturity
    if (!isPostMaturity &amp;&amp; !isCapitulation) {
        revert OperationFailed(OperationCodes.RedemptionBeforeMaturity);
    }
     if (isCapitulation &amp;&amp; !isPostMaturity) {
        //-- code snip --
    }
```

### 2.2 Inflexible Vault Address Update Mechanism Exposes Funds to Risks

**Severity:** Critical

**Status:** Fixed

**Description** The `changeVaultAddress()` function in the `ZeroCouponBondsIssuer.sol` contract allows updating the vault address to which fees are sent, including issuance fees, purchase fees, and early redemption fees. However, this update mechanism only affects new bonds issued post-update, leaving previously issued bonds to continue interacting with the old, potentially compromised vault. This design oversight means that if the original vault becomes compromised or if there's a need to transition to a new vault for operational reasons, all fees from existing bonds continue to be sent to the old vault address, exposing these funds to potential loss or unauthorized access.

**Location** `ZeroCouponBondsIssuer.sol`

**Impact** This vulnerability leaves financial assets at risk if the initial vault becomes compromised or operationally obsolete.

**Recommendation** Implement a more flexible vault address update mechanism that retroactively applies to all existing bonds, not just future issuances. This could involve modifying the `ZeroCouponBonds` contracts to reference the vault address from the `ZeroCouponBondsIssuer` contract dynamically, rather than storing it statically, or implementing a method to update the vault address in all existing bonds securely.

### 2.3 Referrals can Never Claim the Rewards

**Severity:** High

**Status:** Fixed

**Description** The function claimReferralRewards in the provided smart contract code is intended to allow users to claim referral rewards based on certain conditions. An issue was identified in the reward distribution logic. The function attempts to transfer rewards using the interestToken associated with the bondContract. However, the rewards are intended to be in the form of investmentToken, not interestToken, because the protocol collects purchase fees in investmentToken. This discrepancy means that, in practice, there may not be sufficient (or any) balance of interestToken in the vault to cover the reward payments, leading to a failure in executing the reward transfers.

**Location** `AmetVault.sol`

**Impact** The primary impact of this issue is the potential inability for referrers to claim their referral rewards. This failure not only undermines the trust and participation in the referral program but also could lead to reputational damage for the protocol.

**Recommendation** Update the claimReferralRewards function to transfer investmentToken instead of interestToken for referral rewards.

### 2.4 Issuance Fee Handling Leads to Potential under payment of issuance fees.

**Severity:** High

**Status:** Fixed

**Description** The `issueBondContract` function in the `ZeroCouponBondsIssuer.sol` contract has a flaw in its handling of payable issuance fees. Specifically, when a bond issuer sends an amount of ETH that exceeds the set issuance fee (e.g., 1.1 ETH sent when the fee is 1 ETH), the surplus funds (in this example, 0.1 ETH) are not refunded to the issuer but remain within the `ZeroCouponBondsIssuer` contract. This behavior not only results in the unintentional loss of funds for the user but also creates a scenario where subsequent bond issuers can underpay the issuance fee by relying on the contract's existing balance to cover the shortfall, effectively bypassing the required fee payment.

**Location** `ZeroCouponBondsIssuer.sol`

**Impact** This vulnerability exposes the protocol to financial risks in two primary ways:

1. **User Fund Loss**: Users overpaying the issuance fee do not receive refunds for their excess payments, leading to direct financial loss.
2. **Fee Under Payment**: Allows subsequent issuers to underpay the issuance fee by exploiting the contract's retained excess funds, undermining the protocol's economic model.

**Recommendation** To mitigate this issue, it is recommended to modify the `issueBondContract` function to refund any excess ETH sent during the contract call. Implementing checks to ensure that only the exact issuance fee amount is accepted (or that any excess is promptly refunded) will prevent both the unintentional loss of funds and the potential for fee bypass. Additionally, consider requiring exact fee payments to further streamline the process and enforce financial discipline among users.

### 2.5 Lack of Token Standards Enforcement Risks Protocol Fee Integrity

**Severity:** High

**Status:** Acknowledged

**Description** The current implementation in `ZeroCouponBondsIssuer.sol` and `ZeroCouponBonds.sol` contracts does not enforce any checks or restrictions on the types of ERC20 tokens that can be used as investment and interest tokens. This oversight allows for the use of fee-on-transfer (FOT) tokens and rebasing tokens, which inherently alter token balances upon transfers or rebase operations. Such tokens can disrupt the intended fee collection, distribution mechanisms, and the accurate accounting of bond values, leading to potential financial discrepancies and undermining the protocol's economic model.

**Location** `ZeroCouponBondsIssuer.sol`, `ZeroCouponBonds.sol`

**Impact** The use of non-standard ERC20 tokens (FOT and rebasing tokens) without appropriate checks can lead to:

1. Inaccurate fee collection, affecting the protocol's revenue.
2. Discrepancies in bond valuation and interest calculations, potentially disadvantaging investors.
3. Compromised integrity of the economic model, leading to trust and adoption issues among users.

**Recommendation** To mitigate these risks, it is advised to implement a whitelist mechanism for investment and interest tokens, allowing only standard ERC20 tokens without FOT or rebasing features. This can be achieved through:

1. Developing a token standard check function to validate tokens against a set of criteria ensuring compatibility with the protocol's economic model.
2. Incorporating a governance process to review and approve tokens for use within the protocol, ensuring they meet the necessary standards for operation.

**Developer's Response**

From our persepective, we are trying to minimize the governance(centralized) actions so that everyone can effectively issue bonds on our platform. From our side we'll show a disclaimer and will add one in the contract @dev section so bond issuer would know.

### 2.6 Self-Referral in Bond Purchase Allows for Unintended Rewards

**Severity:** High

**Status:** Fixed

**Description** In the `ZeroCouponBonds` contract's purchase function, there is no validation to ensure that the referrer parameter is different from the `msg.sender`. This oversight allows a purchaser to set themselves as their referrer, enabling them to unjustly earn referral rewards. This issue is compounded by the `AmetVault` contract's `claimReferralRewards` function, which disburses rewards based on the stored referral information without validating the legitimacy of the referral relationship. This flaw undermines the integrity of the referral incentive mechanism, potentially leading to exploitation where users claim more rewards than deserved.

**Location** `ZeroCouponBonds.sol`, `AmetVault.sol`

**Impact** The ability for users to self-refer undermines the referral reward system, potentially leading to inflated reward claims and draining the resources allocated for genuine referrals. This could reduce the overall effectiveness of the referral program, impact the financial sustainability of the project, and diminish trust in the platform's fairness.

**Recommendation** Implement a validation check within the purchase function of the ZeroCoupon-Bonds contract to ensure that the referrer address is different from msg.sender. Additionally, consider adding logic in the AmetVault contract to verify the legitimacy of referral relationships before disbursing rewards.

```
    // In ZeroCouponBonds.sol - Function Purchase()
    require(referrer != msg.sender, &amp;quot;Referrer cannot be the purchaser&amp;quot;);
```

### 2.7 Missing Enforcement to Decrease Supply Post Settlement

**Severity:** Medium

**Status:** Fixed

**Description** The `settleContract` function in the smart contract is designed to finalize the state of the bond contract, ensuring that no further bonds can be issued or burned. It signifies that the bond has reached its final state, where the total interest required is calculated based on the difference between the total bonds and the redeemed bonds multiplied by the interest amount per bond. The function performs a check to ensure there are sufficient interest tokens in the contract to cover this calculated total interest requirement. If the check passes, the contract's `isSettled` flag is set to true, and a SettleContract event is emitted.

However, there is an oversight in the contract's logic: Before the settlement there is no enforcement for decreasing the Bond Supply hence the interest amount will be stuck in the Bond unless someone buys the bonds and complete the maturity period.

**Location** `ZeroCouponBonds.sol`

**Recommendation** Implement the enforcement to decrease the bond supply before the settlement to cater the potential scenario.

### 2.8 Uninitialized Vault Address Risks Bond Functionality and Financial Losses

**Severity:** Medium

**Status:** Fixed

**Description** The design of the `ZeroCouponBondsIssuer.sol` contract permits the issuance of bond contracts without specifying a vault address upfront; instead, setting this critical address post-deployment via the `changeVaultAddress()` function. This approach introduces a significant vulnerability under conditions where the contract owner neglects to immediately set the vault address or if the transaction intended to set this address is either delayed or preempted by bond issuance transactions. In particular, should the `changeVaultAddress()` function not be invoked prior to any bond issuances, or if bond issuance transactions are executed before the vault address change is processed, the vault address is defaulted to the zero address (`0x0000000000000000000000000000000000000000`). This defaulting scenario has dire consequences: attempts to purchase bonds via the `purchase()` function in the `ZeroCouponBonds.sol` contract will fail if the vault address is either not set or erroneously set to the zero address. Such failures not only render the issued bonds nonfunctional, thereby negating their intended utility, but also result in the irrevocable loss of issuance fees. These fees, expected to be directed to a valid vault, are instead sent to the zero address, leading to direct financial loss to both the protocol and the bond issuers.

**Location** : `ZeroCouponBondsIssuer.sol`

**Impact** : This vulnerability has a significant financial impact, directly causing loss of funds (issuance fees) intended for the protocol's vault. Moreover, it undermines the trust in the protocol, as users (bond issuers) suffer financial losses due to a preventable oversight in contract initialization. Additionally, it affects the usability of the issued bonds, as they cannot be purchased, effectively rendering any such bond useless and compromising the protocol's functionality.

**Recommendation** :

**Mandatory Vault Initialization**: Modify the `ZeroCouponBondsIssuer constructor()` to require the vault address as a parameter, ensuring it is set upon contract deployment and eliminating the risk of it being uninitialized.

**Validation Check**: Implement a validation check within the `issueBondContract()` function to assert the vault address is not the zero address before proceeding with the bond issuance process.

### 2.9 Missing two-step transfer ownership Pattern

**Severity:** Low

**Status:** Fixed

**Description** The above mentioned contracts use OpenZeppelin's Ownable contract, which provides a single-step mechanism to transfer ownership. The new owner isn't required to confirm their new ownership status. If a mistake occurs during transfer (e.g., a wrong address is provided, or the renounceOwnership function is mistakenly called), it might lead to the contract becoming non-functional if essential functionalities are restricted to the owner

**Location** `ZeroCouponBondsIssuer.sol` , `ZeroCouponBonds.sol`, `AmetVault.sol`

**Impact** Mistakenly transferring ownership to an incorrect or malicious address can lead to potential manipulation of the contract, or the contract might become completely non-functional if the new owner doesn't cooperate. It can also lead to potential financial loss if the contract holds any funds or valuable assets.

**Recommendation** Implement a two-step transfer ownership mechanism. This would entail:

1. A function for the current owner to propose a new owner.
2. A function for the proposed new owner to confirm and finalize the ownership transfer.
3. Follow the pattern from OpenZeppelin's Ownable2Step
4. This two-step process ensures that the new owner is aware and willing to accept the ownership, reducing the risk of accidental ownership transfers.

### 2.10 Uninitialized Referrer Fee Percentage Nullifies Referral Rewards

**Severity:** Low

**Status:** Fixed

**Description** In the `AmetVault` contract, the referral reward mechanism is contingent upon the `referrerPurchaseFeePercentage` variable, which determines the percentage of interest rewards allocated to referrers upon successful bond purchases and settlements. However, the contract is deployed without initializing this crucial variable, implicitly setting it to 0. This oversight becomes problematic when the `changeReferrerPurchaseFeePercentage` function is not invoked to assign a valid percentage before referrals begin to claim their rewards. As a result, even when referrals meet all conditions for reward eligibilitysuch as the bond being fully settled and purchasedclaiming the reward via `claimReferralRewards` yields no actual transfer of funds to the referrer. The function's execution marks the `isRepaid` flag as true, precluding any future attempts to claim rewards, effectively denying referrers their due incentives without recourse.

**Location** `AmetVault.sol`

**Impact** This vulnerability undermines the integrity of the referral incentive mechanism, potentially impacting user engagement and protocol growth. Referrers, motivated by the prospect of rewards for facilitating bond purchases, are left uncompensated, which can erode trust in the protocol's reward structures and deter future participation.

**Recommendation** To rectify this issue and safeguard the referral reward process, it is imperative to initialize the `referrerPurchaseFeePercentage` within the `AmetVault` contract's `constructor`. This ensures that a valid referral fee percentage is in place from the outset, eliminating the risk of reward denial due to oversight or delayed administrative action.

### 2.11 Ownership Renouncement Locks Contract Management

**Severity:** Low

**Status:** Fixed

**Description** The contracts `ZeroCouponBondsIssuer.sol`, `ZeroCouponBonds.sol`, and `AmetVault.sol` inherit from OpenZeppelin's `Ownable` contract, which includes the `renounceOwnership()` function. This function allows the current owner to permanently transfer the contract's ownership to a zero address, effectively relinquishing any administrative control over the contract. Given the critical role of the owner in managing these contracts from issuing bonds, changing fees, to updating vault addresses renouncing ownership could freeze essential functionalities, leading to operational paralysis and potential financial loss.

**Location** `ZeroCouponBondsIssuer.sol`, `ZeroCouponBonds.sol`, `AmetVault.sol`

**Impact** Renouncing ownership in these contracts could result in:

- Loss of control over fee adjustments and vault management which is currently required for proper functioning.

**Recommendation** To mitigate this risk, it is recommended to override the `renounceOwnership()` function to prevent ownership from being renounced or to ensure that such a feature is intentionally included with understood implications. An explicit override within each contract could look like:

```
    function renounceOwnership() public override onlyOwner {
        revert(&quot;Renouncing ownership is disabled&quot;);
    }
```

Furthermore, documenting the potential impact of renouncing ownership within the project documentation can provide clarity and caution to future contract administrators about this irreversible action.

### 2.12 Missing Input Validations

**Severity:** Low

**Status:** Fixed

**Description** In the smart contracts `ZeroCouponBonds.sol` and `ZeroCouponBondsIssuer.sol`, critical functions lack proper input validation, potentially leading to unintended consequences and vulnerabilities within the system. Specifically, the `issueBondContract` function does not thoroughly validate its numerical inputs, such as `totalbonds`, `maturityPeriod`, `investmentAmount`, and `interestAmount`. This oversight allows for the passage and acceptance of zeros as valid inputs, which could result in the creation of non-functional bond contracts or bonds with unintended attributes. Additionally, the withdrawExcessInterest function fails to validate the toAddress parameter for the zero address, posing a risk of unintentional loss of funds or locked assets within the contract.

**Location** `ZeroCouponBonds.sol, ZeroCouponBondsIssuer.sol`

**Developer's Response**

Partially Fixed.User can create bond with 0s if he wishes. Fixed only for withdrawExcess

### 2.13 Missing Validation on Fees can lead to Loss of Bonds Issuer Funds

**Severity:** Low

**Status:** Acknowledged

**Description** The `ZeroCouponBondsIssuer` smart contract includes functions allowing the owner to adjust various fees related to bond issuance, early redemption, and purchase. Specifically, these functions are `changeIssuanceFee`, `changeEarlyRedemptionFeePercentage`, and `changePurchaseFeePercentage`. While these functions provide necessary flexibility for managing the bond contract, they currently lack validation checks to enforce reasonable limits on the fees set. This oversight permits the contract owner to set the fees to any value, including 0 or excessively high values (up to 100% or more), without restriction.

**Location** `ZeroCouponBondsIssuer.sol`

**Impact Potential for Financial Loss**: Setting fees to excessively high levels (e.g., a purchase fee percentage of 100 or more) could result in significant, if not total, loss of funds for users interacting with the contract. Such settings could make bond purchases or redemptions financially unviable.

**Recommendation** Amend the fee-setting functions to include validation checks that enforce minimum and maximum limits for each fee. For example, ensure that percentage-based fees do not exceed 100% and that the issuance fee remains within a sensible range for the platform, economic model.

**Developer Response**

The percentage will be changed for the bonds that will be created after(does not affect already created bonds) and the bond issuer sees the percentage, so no need

### 2.14 Lack of Reentrancy Protection in Critical Functions

**Severity:** Low

**Status:** Acknowledged

**Description** The smart contracts `ZeroCouponBonds`, `AmetVault`, `and ZeroCouponBondsIssuer` lack reentrancy protection mechanisms, specifically the usage of OpenZeppelin's `ReentrancyGuard` library. Functions such as purchase and redeem in `ZeroCouponBonds`, `and claimReferralRewards in AmetVault`, interact with external contracts via token transfers and are susceptible to reentrancy attacks. Without proper reentrancy guards, malicious actors could exploit these functions to perform unintended actions, potentially leading to loss of funds or manipulation of contract states.

**Location** `ZeroCouponBonds.sol`, `AmetVault.sol`

**Recommendation** Implement OpenZeppelin's ReentrancyGuard in all contracts and functions that are susceptible to reentrancy attacks. This can be achieved by inheriting the ReentrancyGuard contract and using the nonReentrant modifier on vulnerable functions

**Developers Response**

Acknowledged. The re-enterancy in the purchase is due to ERC1155(onERC1155Received or onERC1155BatchReceived) but the logic was adjusted accordingly.

**Auditor's Response**

Although Developer has acknowledged and mentioned the applied logic , it is advised to the team to add OZ's Reentrancy Guards to cater any unexpected scenario.

### 2.15 Missing Zero Count Validation in Bond Minting Process

**Severity:** Info

**Status:** Acknowledged

**Description** In the `ZeroCouponBonds` smart contract, particularly within the purchase function, there exists an oversight wherein the function does not explicitly validate the count parameter to ensure it is greater than zero. This parameter represents the number of bonds a user intends to purchase. As currently implemented, the function permits users to call purchase with a count of zero, leading to unnecessary execution and gas expenditure without any real effect on the contract state or token balances, since no bonds are actually purchased or minted in such cases.

**Location** `ZeroCouponBonds.sol`

**Impact Wasted Gas and Resources:** Users can inadvertently or maliciously call the purchase function with a count of zero, leading to wasted gas and computational resources on the Ethereum network. Although the financial impact per transaction might be small, it introduces inefficiency and potential for spamming the contract.

**Recommendation Implement Parameter Validation:** Update the purchase function to include a validation check for the count parameter, ensuring it is greater than zero before proceeding with the rest of the function's logic. This can be implemented as a simple require statement at the beginning of the function:

```
require(count &gt; 0, &quot;Purchase count must be greater than zero&quot;);
```

**Developer's Response**

This a good one but no need to fix as the UI should handle this kind of things

### 2.16 Logic For Pausing Can Be Optimized

**Severity:** Info

**Status:** Fixed

**Description** Using bools for storage incurs overhead. Booleans are more expensive than uint256 or any type that takes up a full word because each write operation emits an extra SLOAD to first read the slot's contents, replace the bits taken up by the boolean, and then write back.

**Location** `ZeroCouponBondsIssuer.sol`

**Recommendation** Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (20000 gas) when changing from false to true, after having been true in the past

### 2.17 Use Custom Errors For Gas Optimization

**Severity:** Info

**Status:** Fixed

**Description** Custom errors from 0.8.4, leads to cheaper deploy- and run-time costs. Note: the run time cost is only relevant when the revert condition is met. In short, replace revert strings with custom errors

**Location** `AmetVault.sol`

### 2.18 Separate require Statements Can Optimize Gas

**Severity:** Info

**Status:** Fixed

**Description** The `claimReferralRewards` function in the `AmetVault` contract utilizes a combined logical condition within a single require statement to validate whether a referrer is eligible to claim rewards. This design pattern, while concise, may not be the most gas-efficient under certain conditions. By splitting the combined condition into separate require statements, there is a potential for gas savings, especially with the Solidity optimizer enabled. The optimizer can more effectively optimize individual conditions compared to combined logical operations. Furthermore, separating the conditions enhances code readability and allows for earlier termination of the function if the first condition fails, potentially saving gas.

**Location** `AmetVault.sol`

**Recommendation** By implementing the recommendation, each condition is evaluated independently, allowing the EVM to exit early if the first condition fails. This can lead to minor gas savings per transaction, which accumulates across numerous transactions, contributing to overall efficiency improvements in contract execution

**Disclaimer:**

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts.