Amet Finance Security Review

Version 1.1

05.03.2024

Conducted by:
nmirchev8 , Independent Security Researcher

Table of Contents

1 About nmirchev8

Nikola Mirchev, known as nmirchev8, is an independent security researcher experienced in Solidity smart contract auditing and contest participation. Having found over 40 H/M confirmed vulnerabilities, he consistently strives to provide top-quality security auditing services.

2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Amet Finance
Repository	https://github.com/Amet-Finance/contracts/tree/main/contracts/fixed-flex
Commit hash	9b683708ff573a2d1348ea1535ad0dbcdae6d895
Documentation	https://docs.amet.finance/v1/
Methods	Manual review & testing

Scope

contracts/fixed-flex/Bond.sol
contracts/fixed-flex/Issuer.sol
contracts/fixed-flex/Vault.sol

Issues Found

Critical risk	0
High risk	0
Medium risk	1
Low risk	3
Informational	0

5 Findings

5.1 Medium risk

5.1.1 Problems may arrise if issuer select non-standard ERC20 such “fee-on-transffer” or rebalancing for `purchaseToken`

Severity: *Medium risk*

Context: Issuer.sol#34 **Description:** - The issue is not neglectable, because end users has the freedom to choose what ERC20 to use for their bonds - If such token is used, transferred amounts to contracts would be different from what it is written as state variables, which could lead to last referral being unable to claim funds and other issues. E.g. `purchaseAmount = Z` 1. User A purchases a bond with fee on transfer Token A and set refferal address B. 2. The amount transffered to vault address is `quantity * purchaseAmount * purchaseRate / _PERCENTAGE_DECIMAL - token transfer fee` 3. It is a valid scenario that all `purchaseRate` fees are only referral fees 4. If referral try to claim such amount, the result calculation would use raw `purchaseAmount * quantity` to determine the corresponding reward, but in reality `Vault.sol` has received less. And transaction would revert, because contract doesn’t have enough balance.

Recommendation: - It is hard to handle such case and you should decide whether, or not to support such tokens. - If you choose not to support, user may loose gas to depoloy `Bond.sol` contract, which would be unusable.

```
uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(address(this));
IERC20Upgradeable(token).safeTransferFrom(address(msg.sender), address(this), amount
    );
uint256 balanceAfter = IERC20Upgradeable(token).balanceOf(address(this));
uint256 transferedAmount = balanceAfter - balanceBefore;
// if you dont want to support fee on transfer token use below:
require (transferedAmount == amount, ...);
// use transferedAmount if you want to support fee on transfer token
```

Resolution: - “This is more of a problem and not an issue. Generally we do not recommend using FOT tokens(or similar) because the behaviour of the Bond contract will not be as we designed initially. On the new front-end we show disclaimer on issuing bonds such tokens”

5.2 Low risk

5.2.1 If `Issuer.changeVault` is called, purchase and redeem of old bonds would be DoS-ed

Severity: *Low risk*

Context: Issuer.sol#L47

Description: Currently `Bond.sol`, `Issuer.sol` and `Vault.sol` are tightly coupled, because "issuer" holds reference to "vault" and deployed "bond"s are holding reference to "issuer" and they are using current "issuer" relation to "vault" to obtain "self" contract:

```
    function redeem(uint40[] calldata bondIndexes, uint40 quantity, bool
        isCapitulation) external nonReentrant {
        Types.BondLifecycle storage lifecycleTmp = lifecycle;
        IERC20 payoutTokenTmp = payoutToken;

        uint8 earlyRedemptionRate = _issuerContract.vault().getBondFeeDetails(
            address(this)).earlyRedemptionRate;
...
```

Note how if we want to query details for the given `Bond`, we use the current `Vault` implementation inside `Issuer`. But how that could be a problem? Almost in all functions inside `Vault.sol` we have validation that give "bond" address has been initialized from this `Vault.sol`:

```
    function recordReferralPurchase(address operator, address referrer, uint40
        quantity) external {
        _isBondInitiated(_bondFeeDetails[msg.sender]);
```

But having in mind that we have the functionality to change `Vault.sol` reference inside `Issuer.sol` contract and supposing it would be used in some time:

```
    function changeVault(address vaultAddress) external onlyOwner {
        vault = IVault(vaultAddress);
        emit VaultChanged(vaultAddress);
    }
```

What happens inside all previously deployed `Bond.sol` contracts: Most important functions such as `redeem` and `purchase` are blocked, because both uses `issuerContract.vault()` to trigger `recordReferralPurchase`, or to query bond details. And the new vault state data for `_bondFeeDetails[bondAddress]` won't be "initialized". This is major problem, because even if owner return vault reference to old contract, all bonds issued from the new would have the same problem and this leads to bad loop.

Recommendation: Insert new state variable inside `Bond.sol` `address vault;`, which would be set only on initialization and then used in the functions. This way you remove the direct dependency for each "bond" to current vault inside "issuer" and a change of vault won't break old "bond"s, because they will check from the vault, which has initialized them.

Resolution: "This is actually is not an issue. See https://github.com/Amet-Finance/contracts/blob/main/test/fixed-flex/vault.test.ts Issued bond behaviour after changing vault for purchase. The reason why we implemented in this way is because if the vault is compromised or we need an upgrade, all other bond contracts would stop and only re-enable when we call updateBondFeeDetails in the Vault contract. This is a the protocol design, so your point is correct that it will be DoS-ed but it's intended."

5.2.2 User can easily provide address owned by him as referral address, which will be at the expense of the protocol

Severity: *Low risk*

Context: Bond.sol#L69

Description: - Currently there is a `purchase fee`, which is the reward source for the protocol. There is also `refferal fee`, which is being subtracted from `purchase fee` if address different from bond buyer is provided. The thing is that any user can provide address owned by him, so he recover `uint256 rewardAmount = Math.mulDiv((referrer.quantity * purchaseAmount), bondFeeDetails.referrerRewardRate, _PERCENTAGE_DECIMAL);` of what he has paid

Recommendation: Consider implementing additional payment, if there is `refferal fee`, otherwise only `protocol fee`

Resolution: - "This is not an issue as well because if we want to do an on-chain referral system, this problem will arise. Basically it's okay if they do so, we can call it somehow a backdoor that is okay to be used."

5.2.3 User calling `purchase` with amount of 0 could infinitely increment `lifecycleTmp.uniqueBondIndex++`

Severity: *Low risk*

Context: Bond.sol#L88

Description: A user can falsely increment `lifecycleTmp.uniqueBondIndex` if he calls `purchase` with 0 amount. I couldn't find major exploit path, but consider implementing check if purchased amount is greater than 0

Resolution: - "The only problem that can be here is when the person does (2ˆ40-1) times with 0 quantity and then it blocks for other purchasers in total getting $109B(calculated with Polygon's lowest fee) loss."