Pashov Audit Group

Saffron Security Review

July 31st 2025 - August 3rd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Saffron Fixed Income

Saffron Fixed Income vaults are integrated with Uniswap V3 and let users participate on a fixed side for stable returns or a variable side for higher, riskier rewards from liquidity fees. It employs adapters for yield platform integration, bearer tokens to represent user stakes, and automated vault lifecycle management for deposits, earnings settlement, and fee distribution.

5. Executive Summary

A time-boxed security review of the **saffron-finance/fixed-income** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Saffron Fixed Income**. A total of **12** issues were uncovered.

Protocol Summary	
Project Name	Saffron Fixed Income
Protocol Type	Fixed Income
Timeline	July 31st 2025 - August 3rd 2025

Review commit hash:
- 65b34c3a07c8f8e38c4b999a166c529048de8465 (saffron-finance/fixed-income)

Fixes review commit hash:
- 64e488539107548ab88521afe339e157df8a4f6a (saffron-finance/fixed-income)

Scope

`RestrictedVaultFactory.sol` `UniV3Vault.sol` `Vault.sol` `VaultBearerToken.sol` `VaultFactory.sol` `AdapterBase.sol` `UniV3FullRangeAdapter.sol` `UniV3LimitedRangeAdapter.sol`

6. Findings

Findings count

Severity	Amount
Medium	3
Low	9
Total findings	**12**

Summary of findings

ID	Title	Severity	Status
[M-01]	Vaults should use a factory for fee receiver instead of storing	Medium	**Resolved**
[M-02]	Hardcoded address restricts multi-chain deployments	Medium	**Resolved**
[M-03]	Malicious adapter pool can censor fixed investors or overcollateralize	Medium	**Resolved**
[L-01]	Arithmetic overflow or underflow in `PositionValue` library	Low	**Resolved**
[L-02]	Variable side depositors may deposit less than intended	Low	**Resolved**
[L-03]	Vault can be initialized with unexpected `feeBps`	Low	**Resolved**
[L-04]	Missing vault expiration enables indefinite dormant vaults	Low	Acknowledged
[L-05]	No access control in `initializeVault()` allows unauthorized init	Low	**Resolved**
[L-06]	Incorrect variable in `liquidityRemoved` event	Low	**Resolved**
[L-07]	Malicious `variableAsset` token griefing attack	Low	Acknowledged
[L-08]	Bytecode validation missing for adapter instances	Low	**Resolved**
[L-09]	Pre-start trading fees may result in unfair investor benefits	Low	Acknowledged

Medium findings

[M-01] Vaults should use a factory for fee receiver instead of storing

Severity

Impact: Medium

Likelihood: Medium

Description

Currently, the `feeReceiver` address is stored as a state variable within each Vault contract during initialization. It is passed from the Vault Factory at deployment time via the `initializeVault(...)` function:

```
IVault(vaultInfo[vaultId].addr).initialize(..., feeBps, feeReceiver);
```

This design means that if the Factory's `feeReceiver` address changes later (e.g., due to a governance update or operational reconfiguration), previously deployed Vaults will continue using the outdated address. As a result:

- Fees may be sent to obsolete or incorrect recipients.
- Fee routing becomes inconsistent across Vaults over time.

This approach tightly couples fee logic to a single deployment-time value, which undermines centralized or upgradable fee management.

Recommendations

Rather than persisting the `feeReceiver` in each Vault, Vaults should dynamically query the current address from the Factory. This ensures consistent behavior across the protocol. For example:

```
function getFeeReceiver() public view returns (address) {
    return IVaultFactory(factory).feeReceiver();
}
```

[M-02] Hardcoded address restricts multi-chain deployments

Severity

Impact: Medium

Likelihood: Medium

Description

In the `UniV3LimitedRangeAdapter` contract, the address of the **Uniswap V3 NonfungiblePositionManager** is hardcoded as:

```
INonfungiblePositionManager public constant positionManager =
INonfungiblePositionManager(0xC36442b4a4522E871399CD717aBDD847Ab11FE88);
```

While this address is correct for **Ethereum mainnet**, **Arbitrum**, and **Optimism**, it does **not match the deployed address on Base** and may differ on other Layer 2s (e.g., Base, Unichain, zkSync, Scroll).

If the contract is deployed on a network where the PositionManager address differs, all interactions with Uniswap V3 positions (minting, burning, querying) will **revert**, rendering the adapter and dependent vaults non-functional.

Given that the project explicitly plans to support **Base** and potentially other L2s, this hardcoding introduces a **high likelihood of deployment failure** and requires re-compilation or code edits for every chain.

Note: The severity is rated as High due to its Deployment Critical Impact, even though it is not a direct security vulnerability.

Recommendations

- **Parameterize the PositionManager address as an immutable constructor argument:**

```
INonfungiblePositionManager public immutable positionManager;

constructor(INonfungiblePositionManager _positionManager) {
    require(address(_positionManager).code.length > 0, "Invalid PositionManager address");
    positionManager = _positionManager;
}
```

[M-03] Malicious adapter pool can censor fixed investors or overcollateralize

Severity

Impact: High

Likelihood: Low

Description

The protocol currently lacks an upper bound check on the actual liquidity provided by Fixed Side investors during capital deployment. In the `deployCapital()` process, token amounts (amount0, amount1) are calculated using pool parameters obtained via `pool.slot0()`. However, the provided pool address is not verified against Uniswap V3 Factory to ensure authenticity.

Adapter Deployment (Unverified Pool):

```
function createAdapter(
    uint256 adapterTypeId,
    address poolAddress,
    bytes calldata data
) public virtual {
    ...
    IAdapter(adapterAddress).initialize(adapterId, poolAddress, defaultDepositTolerance, data);
}
```

Adapter Initialization:

```
function initialize(
    uint256 _id,
    address _pool,
    uint256 _depositTolerance,
    bytes memory uniV3InitData
) public virtual override onlyWithoutVaultAttached onlyFactory {
    ...
    IUniswapV3Pool uniswapV3Pool = IUniswapV3Pool(_pool);
    pool = uniswapV3Pool;
}
```

A malicious actor can deploy an adapter pointing to a fake pool contract that returns manipulated `sqrtPriceX96` values, causing incorrect computation of required token amounts. As a result, the Fixed Side investor (even unintentionally) may provide **significantly more liquidity** than intended when minting the position.

Token Amount Calculation using slot0():

```
(uint160 sqrtRatioX96, , , , , , ) = pool.slot0();
(uint256 amount0, uint256 amount1) = LiquidityAmounts.getAmountsForLiquidity(
    sqrtRatioX96,
    poolMinTick.getSqrtRatioAtTick(),
    poolMaxTick.getSqrtRatioAtTick(),
    fixedSideCapacity
);
```

While the protocol verifies that actual minted liquidity meets a minimum threshold (`l * invDepositTolerance`), there is no check ensuring that the liquidity does **not exceed** a maximum limit:

```
require(uint256(l) * invDepositTolerance / 10000 < _liquidity, "L");
```

Attack Scenarios:

1. **Overcollateralization Exploit:**
2. Overcollateralizing Fixed Side shifts the position's weight towards Variable Side investors.
3. Variable Side participants unfairly receive a larger share of the position's yield, while Fixed Side investors risk more capital than intended.
4. **Censorship of Fixed Side Participation (Denial of Service):**
5. By setting the malicious pool's slot0() to return extreme values, an attacker can cause the calculated `amount0` and `amount1` to be unrealistically small.
6. Honest Fixed Side investors' calls will be reverted due to current slippage protection, effectively **blocking their ability to deploy capital**.
7. This **censorship vector** can be used to paralyze Fixed Side participation in specific Vaults.

This scenario can occur even if Fixed Side investors are non-malicious. However, a deliberate attacker could exploit this by:

1. Deploying a malicious adapter with a manipulated pool.
2. Forcing Fixed Side investors to overcontribute liquidity.
3. Or, preventing Fixed Side investors from participating altogether (censorship).
4. Then, entering as a Variable Side investor to gain an outsized share of profits without proportional risk.

Recommendations

1. **Enforce Upper Bound Liquidity Checks:**
2. After minting, verify that the actual liquidity does not exceed a defined tolerance:

   `solidity require(_liquidity <= uint256(l) * depositTolerance / 10000, "Excessive liquidity");` * Where `depositTolerance` could be set to a reasonable upper slippage (e.g., 10100 = +1%).
3. **Ensure Adapter Pool Authenticity:**
4. Validate the pool address against Uniswap V3 Factory in the adapter constructor to prevent malicious pool injection.

Low findings

[L-01] Arithmetic overflow or underflow in `PositionValue` library

Some libraries from the Uniswap V3 Core and Periphery codebases have been adapted so that they can be used with Solidity 0.8.0 and above.

In the case of the `PositionValue` library, the adaptation has not been done correctly, as the fee calculations rely on overflows and underflows, so in Solidity 0.8.0 and above, the execution can panic with an arithmetic overflow or underflow error.

The following code snippet shows the issue:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import {PositionValue} from "../../contracts/vendor/@uniswap/v3-periphery/contracts/libraries/
PositionValue.sol";
import {INonfungiblePositionManager} from "../../contracts/vendor/@uniswap/v3-periphery/
contracts/interfaces/INonfungiblePositionManager.sol";

contract PoC is Test {
    INonfungiblePositionManager positionManager =
INonfungiblePositionManager(0xC36442b4a4522E871399CD717aBDD847Ab11FE88);

    // forge test --fork-url https://eth.llamarpc.com --fork-block-number 23054470 --mt
test_PositionValue
    function test_PositionValue() public {
        PositionValue.fees(positionManager, 19);
    }
}
```

```
Ran 1 test for test/foundry/PoC.sol:PoC
[FAIL: panic: arithmetic underflow or overflow (0x11)] test_PositionValue() (gas: 52155)
```

It is recommended to wrap the arithmetic operations in the `PositionValue` library with `unchecked` blocks.

[L-02] Variable side depositors may deposit less than intended

In `UniV3Vault`, deposits of variable side caps the amount to be deposited to the remaining capacity of the variable side.

```
      // Deposit only up to capacity
      amount = (amount + variableBearerToken.totalSupply() >= variableSideCapacity)
        ? variableSideCapacity - variableBearerToken.totalSupply()
        : amount;
      require(amount > 0, "NZD");
```

As a result, users may deposit less than they were interested in depositing.

Consider the following scenario: - Alice wants to deposit assets for the total variable side capacity of the vault. - Bob front-runs Alice and deposits for the total variable side capacity minus a few tokens. - Alice is left with a small amount of variable bearer tokens.

It is recommended to allow depositors of the variable side to specify a minimum amount to deposit and revert if the amount deposited is less than the minimum.

[L-03] Vault can be initialized with unexpected `feeBps`

`VaultFactory.initializeVault()` calls `Vault.initialize()` passing `feeBps` as one of the parameters. This state variable can be changed by the owner, so the creator of the vault might initialize it with a different value than the one expected.

Consider the following scenario: - Alice creates a vault and, at that time, the `feeBps` is 100 (1%). - Owner changes the `feeBps` to 500 (5%). - Alice initializes the vault assuming that a 1% fee will be applied, but in reality, a 5% fee will be applied.

The following change is suggested to ensure that the `feeBps` expected by the vault creator matches the one set in the factory:

```
  function initializeVault(
    uint256 vaultId,
    uint256 fixedSideCapacity,
    uint256 variableSideCapacity,
    uint256 duration,
-   address variableAsset
+   address variableAsset,
+   uint256 expectedFeeBps
  ) public {
    // Get vault info for the vault we want to initialize and make sure msg.sender is the
creator
    require(vaultInfo[vaultId].creatorAddress == msg.sender, "CMI");

+  require(feeBps == expectedFeeBps, "FEE_MISMATCH");
```

[L-04] Missing vault expiration enables indefinite dormant vaults

The protocol lacks a mechanism to expire vaults that never get fully funded. This can lead to:

- **Stale vaults**: vaults starting with different market conditions than when configured.
- **Opportunistic starts**: leftover funds in old vaults might tempt someone to fill the missing side, just to trigger the vault and claim those funds (this is attractive if the vault has a short duration).

Example dormant vault: [0x2443eEAC7B1523636c94aFbf4D6fc55fd1507E3b]

Consider adding an optional expiration duration for vault configuration (e.g., `vaultStartDeadline`). If unmet, the vault becomes unstartable, and participants can still withdraw their funds.

[L-05] No access control in `initializeVault()` allows unauthorized init

The `RestrictedVaultFactory` contract restricts vault creation to the current contract owner. However, it fails to override the `initializeVault()`, which is still accessible to the previous owner who is recorded as the vault's creator.

If ownership of the `RestrictedVaultFactory` contract changes after vault creation but before initialization or there are vaults that were left uninitialed, the previous owner can still initialize those vaults with arbitrary parameters, bypassing the intended restricted control.

```
  function initializeVault(
    uint256 vaultId,
    uint256 fixedSideCapacity,
    uint256 variableSideCapacity,
    uint256 duration,
    address variableAsset
  ) public {
    // Get vault info for the vault we want to initialize and make sure msg.sender is the
creator
@>  require(vaultInfo[vaultId].creatorAddress == msg.sender, "CMI");

    --- SNIPPED ---
  }
```

Consider overriding the `initializeVault()` in `RestrictedVaultFactory` to add the `onlyOwner` modifier.

[L-06] Incorrect variable in `liquidityRemoved` event

In the `_removeLiquidity` function, the `LiquidityRemoved` event emits the variable `liquidity`, which is a storage variable. However, the actual liquidity removed is obtained dynamically via:

```
(,,,,,, uint128 currentLiquidity,,,,) = positionManager.positions(tokenId);
...
emit LiquidityRemoved(vaultAddress, address(pool), tokenId, liquidity, amount0, amount1);
```

This `currentLiquidity` value is then passed to `decreaseLiquidity`, making it the accurate representation of the removed liquidity.

Using the storage `liquidity` variable in the event may result in discrepancies between the event log and the actual action, especially if the storage variable is outdated or out of sync with the real-time position state. While in most cases they will match, any deviation could mislead off-chain systems relying on accurate event data (e.g., indexers, dashboards).

[L-07] Malicious `variableAsset` token griefing attack

The protocol allows the creation of Vaults where the `variableAsset` is a user-supplied ERC20 token address. There are no enforced on-chain validations (e.g., whitelisting of trusted tokens) to ensure the token's behavior is compliant and safe.

```
  function initializeVault(
    uint256 vaultId,
    uint256 fixedSideCapacity,
    uint256 variableSideCapacity,
    uint256 duration,
    address variableAsset
  ) public {
    // Get vault info for the vault we want to initialize and make sure msg.sender is the
creator
    require(vaultInfo[vaultId].creatorAddress == msg.sender, "CMI");

    // Initialize vault and assign its corresponding adapter
>>  IVault(vaultInfo[vaultId].addr).initialize(vaultId, duration,
vaultInfo[vaultId].adapterAddress, fixedSideCapacity, variableSideCapacity, variableAsset,
feeBps, feeReceiver);
...
```

```
  function initialize(
    uint256 _vaultId,
    uint256 _duration,
    address _adapter,
    uint256 _fixedSideCapacity,
    uint256 _variableSideCapacity,
    address _variableAsset,
    uint256 _feeBps,
    address _feeReceiver
  ) public virtual override notInitialized {
    // Validate args
    // vaultId and feeBps are already checked in the VaultFactory
    require(msg.sender == factory, "NF");
    require(_duration != 0, "NEI");
    require(_adapter != address(0), "NEI");
    require(_variableSideCapacity != 0, "NEI");
    require(_fixedSideCapacity != 0, "NEI");
>>  require(_variableAsset != address(0), "NEI");
...
>>  variableAsset = _variableAsset;
...
  }
```

An attacker can deploy a malicious ERC20-compliant token that:

- Passes basic ERC20 interface checks (`totalSupply` , `balanceOf`).
- Implements a honeypot behavior where `transfer()` or `transferFrom()` functions **selectively** revert under certain conditions (e.g., only when called during withdrawals).

Attack scenario:

1. The attacker creates a Vault using a malicious `variableAsset` token.
2. The attacker fulfills the variable side capacity using their honeypot token.
3. An unsuspecting fixed side investor deposits real assets (e.g., USDC/DAI) into the Vault, seeing seemingly normal capacity and yield.
4. Fixed side investor tries to invoke `claim()` to receive `variableAsset` tokens.
5. The malicious token reverts on the transfer, causing the entire transaction to fail.
6. Upon Vault settlement/maturity, the attacker can receive yeild from the pool.
7. **Fixed Side funds remain locked indefinitely**, as the Vault logic assumes token transfers succeed and fixed side investor has nonzero amount of `fixedBearerToken`.

This vector can result in a **Denial of Service (DoS) for withdrawals**, making Fixed Side investors' funds unrecoverable unless a protocol upgrade or manual intervention occurs. The impact is critical for Fixed Side investors as their principal becomes permanently locked.

While the attack vector is straightforward and low-cost, its success depends on Fixed Side investors willingly interacting with Vaults that use untrusted variableAsset tokens. This makes the likelihood Low under the assumption that:

- The frontend filters or flags vaults using unknown tokens.
- Experienced users perform due diligence on the Vault's asset composition.

However, in permissionless systems, with the presence of automated bots and social engineering (e.g., vaults offering unusually high yields), there remains a tangible risk of exploitation.

Recommendations

To mitigate this attack vector:

Enforce an On-Chain Whitelist for Variable Assets:

- Only allow Vault creation with pre-approved ERC20 tokens.
- Governance or admin-controlled process for token approval.

[L-08] Bytecode validation missing for adapter instances

When a vault is created via the `createVault` function, the factory verifies that the adapter address provided by the user corresponds to a registered adapter. However, it does **not verify** whether the adapter's associated bytecode (template) is still valid and present in the factory's `adapterTypeByteCode` mapping.

The system allows the contract owner to "delete" (deactivate) adapter templates by clearing their bytecode from `adapterTypeByteCode`. Nevertheless, a malicious user can pre-deploy multiple adapter instances (using a soon-to-be-deprecated adapter type), and after the

bytecode has been deleted by the owner, these instances can still be used for vault creation. This allows the bypassing of owner-intended deactivation of adapter types, leading to the unintended creation of vaults with deprecated adapter logic.

The issue arises from missing logic that should enforce that **only adapter instances with an active (undeleted) adapter type bytecode can be used.**

Recommendations

Add a check in the `createVault` function to ensure that the adapter's associated type ID still has valid bytecode in the factory's storage. Specifically:

```
bytes memory adapterBytecode = adapterTypeByteCode[_adapterInfo.adapterTypeId];
require(adapterBytecode.length != 0, "BDE"); // Bytecode Deleted
```

This ensures that even if an adapter instance was deployed prior to bytecode deletion, it cannot be used to create new vaults after its template has been deactivated.

[L-09] Pre-start trading fees may result in unfair investor benefits

In the current design, the Fixed Side investor's liquidity can be added to the Uniswap V3 pool and begin generating trading fees **before** the Vault officially starts (`isStarted == false`).

However, once the Vault is started, these **pre-start fees become indistinguishable from post-start fees**, and may be indirectly distributed to Variable Side investors during withdrawal or claim processes. This creates an **economic imbalance**, where:

- The Fixed Side investor **assumes all early risk** and provides capital to the pool,
- While Variable Side investors **may benefit from pre-start fees** without contributing any risk or capital during that time.

This behavior contradicts the intended design of the Vault, in which Variable Side investors are expected to earn yield **only** from post-start activity.

Recommendations

Introduce a mechanism to **isolate or attribute pre-start fees**, such as:

- In `start()` , call `collect()` and assign all accumulated pre-start fees exclusively to the Fixed Side investor before enabling Variable Side participation.
- Alternatively, track and subtract pre-start fees from total earnings, ensuring that Variable Side investors only benefit from fees earned **after** the Vault has officially started.