



UniV3 Fixed Income Vaults Audit Report

Version 1.0

Lead Auditors

[0xleastwood](#)

January 14, 2026

Contents

1	About Researchers	2
2	Disclaimer	2
3	Risk Classification	2
4	Protocol Summary	2
5	Audit Scope	3
5.0.1	Core Contracts	3
5.0.2	Adapters	3
5.0.3	Interfaces	3
6	Executive Summary	3
7	Findings	5
7.1	Medium Risk	5
7.1.1	Fee-on-transfer tokens are not supported on the variable deposit side	5
7.2	Informational	6
7.2.1	Missing vault address assignment in adapter	6
7.2.2	Redundant check in createAdapter	6
7.2.3	Redundant factory check in <code>UniV3Vault.initialize</code>	6
7.2.4	Potentially unnecessary permission check in <code>RestrictedVaultFactory</code>	6
7.2.5	Ownership renunciation in <code>RestrictedVaultFactory</code> should not be possible	7
7.3	Gas Optimization	8
7.3.1	Use cloning for adapter/vault deployments	8
7.3.2	Storing full bytecode instead of boolean value	8

1 About Researchers

[Oxleastwood](#) is an independent security researcher specializing in Web3 and smart contract security. Contracted directly by the Saffron Finance team for this engagement, he brings extensive experience identifying vulnerabilities and strengthening the security posture of decentralized protocols.

2 Disclaimer

The security researchers make every effort to identify vulnerabilities within the allotted time but hold no responsibility for the findings in this document. A security audit does not endorse the underlying business or product. This engagement was time-boxed and focused solely on the security aspects of the Solidity implementation.

3 Risk Classification

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

4 Protocol Summary

Saffron Fixed Income Vaults facilitate a fixed-for-variable yield exchange mechanism. The protocol enables Uniswap V3 liquidity position holders (fixed side) to sell their future expected earnings in exchange for an upfront fixed payment. Variable side participants pay this upfront amount to receive all yield generated over the vault's duration.

Each vault operates as a capacity-based system with two sides:

- **Fixed Side:** Deposits a Uniswap V3 liquidity position (token0/token1) via an adapter, receives upfront payment in variable asset, and mints a claim token. The adapter manages the LP position and collects Uniswap V3 fees.
- **Variable Side:** Deposits variable asset tokens up to a specified capacity, receives bearer tokens representing their share, and earns all yield (Uniswap V3 fees) generated during the vault period.

Vaults auto-start when both sides reach capacity. After the duration period, earnings are settled and distributed proportionally to variable side participants based on their bearer token holdings. Fixed side participants can claim their upfront payment plus a proportional share of variable side deposits.

The system consists of three main components:

1. **VaultFactory:** Ownable contract that deploys vault and adapter instances, manages vault/adapter type bytecode, and configures protocol parameters (fees, fee receiver).
2. **UniV3Vault:** Vault implementation handling deposits, withdrawals, earnings distribution, and bearer token accounting. Integrates with adapters for Uniswap V3 interactions.
3. **Adapters** (UniV3LimitedRangeAdapter, UniV3FullRangeAdapter): Handle Uniswap V3 PositionManager interactions, liquidity calculations, position minting/burning, and earnings settlement.

Key security considerations include: access control for factory owner operations and vault initialization; reentrancy protection on all state-changing functions; accurate bearer token accounting relative to actual token balances; capacity management preventing over-deposits; precise earnings calculation and proportional distribution; Uniswap

V3 integration handling (slippage protection, liquidity bounds, deadline validation); fee-on-transfer token compatibility; factory bytecode validation and revocation mechanisms; and proper handling of edge cases in vault start conditions and early withdrawals.

5 Audit Scope

The audit scope was limited to:

5.0.1 Core Contracts

- `Vault.sol` - Abstract base vault contract
- `UniV3Vault.sol` - Uniswap V3 vault implementation
- `VaultFactory.sol` - Factory contract for deploying vaults and adapters
- `RestrictedVaultFactory.sol` - Restricted factory with owner-only initialization
- `VaultBearerToken.sol` - ERC20 bearer token implementation

5.0.2 Adapters

- `adapters/AdapterBase.sol` - Base adapter contract
- `adapters/UniV3LimitedRangeAdapter.sol` - Limited range Uniswap V3 adapter
- `adapters/UniV3FullRangeAdapter.sol` - Full range Uniswap V3 adapter

5.0.3 Interfaces

- `interfaces/IAdapter.sol` - Adapter interface
- `interfaces/IUniV3Adapter.sol` - Uniswap V3 adapter interface
- `interfaces/IVault.sol` - Vault interface

6 Executive Summary

Over the course of **5** days, 0xleastwood conducted an audit on the [UniV3 Fixed Income Vaults](#) smart contracts provided by [Saffron Finance](#). In this period, a total of **8** issues were found.

Summary

Project Name	UniV3 Fixed Income Vaults
Repository	fixed-income
Commit	edaacd47bfd8...
Audit Timeline	Nov 3rd - Nov 7th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	0
Informational	5
Gas Optimizations	2
Total Issues	8

Summary of Findings

[M-1] Fee-on-transfer tokens are not supported on the variable deposit side	Open
[I-1] Missing vault address assignment in adapter	Open
[I-2] Redundant check in <code>createAdapter</code>	Open
[I-3] Redundant factory check in <code>UniV3Vault.initialize</code>	Open
[I-4] Potentially unnecessary permission check in <code>RestrictedVaultFactory</code>	Open
[I-5] Ownership renunciation in <code>RestrictedVaultFactory</code> should not be possible	Open
[G-1] Use cloning for adapter/vault deployments	Open
[G-2] Storing full bytecode instead of boolean value	Open

7 Findings

7.1 Medium Risk

7.1.1 Fee-on-transfer tokens are not supported on the variable deposit side

Context: [UniV3Vault.sol#L63-L69](#)

Description: The implementation intends to support fee-on-transfer tokens for variable side deposits but because of the exact transfer amount check in `deposit()`, any deposit will revert. However, if a fee-on-transfer token somehow passes this check (e.g., very small fees that round to zero, or a token that becomes fee-on-transfer after vault initialization), bearer tokens are minted based on the expected amount (line 69) rather than the actual received amount. This creates an accounting mismatch where:

- Bearer token supply exceeds actual token balance.
- Unable to reach variable side deposit cap because transfer will always be less than the available cap space.
- Early withdrawals may fail or send less than expected.
- Claim function may calculate incorrect amounts.

Impact: A list of problems caused when explicitly supporting fee-on-transfer tokens:

- Accounting mismatch between bearer tokens and actual token balance
- Incorrect capacity calculations leading to premature or failed vault starts
- Withdrawal failures or incorrect withdrawal amounts
- Last withdrawer may be unable to fully withdraw
- Incorrect claim amounts for fixed side depositors

Recommendation: Use the actual received amount for bearer token minting:

```
// Transfer (restricted to non-deflationary tokens)
uint256 oldBalance = IERC20(variableAsset).balanceOf(address(this));
IERC20(variableAsset).safeTransferFrom(msg.sender, address(this), amount);
uint256 newBalance = IERC20(variableAsset).balanceOf(address(this));
uint256 actualReceived = newBalance - oldBalance;
require(actualReceived == amount, "NDT");

// Mint bearer tokens based on actual received amount
variableBearerToken.mint(msg.sender, actualReceived);
```

Additionally, consider adding validation during vault initialization to explicitly reject fee-on-transfer tokens, and document that only non-fee-on-transfer tokens are supported if this is no longer intended as a feature.

7.2 Informational

7.2.1 Missing vault address assignment in adapter

Context: [VaultFactory.sol#L149-L188](#)

Description: The `createVault()` function creates a vault and associates it with an adapter, but does not set the vault address in the adapter contract. While vault actions are locked until initialization, setting the vault address during creation would establish the bidirectional relationship immediately and improve code clarity.

Recommendation: Add a call to set the vault address in the adapter within `createVault()` after vault deployment. Since all vault actions are locked until initialization, this can be done safely during creation. This ensures the adapter-vault relationship is fully established from the moment of creation.

7.2.2 Redundant check in `createAdapter`

Context: [VaultFactory.sol#L265](#)

Description: The `createAdapter()` function in `VaultFactory.sol` includes a check `require(defaultDepositTolerance > 0, "DDT")`. The `defaultDepositTolerance` is initialized to 100 in the contract and the `setDefaultDepositTolerance()` function does not allow it to be set to zero. This check can never fail and is therefore redundant, adding unnecessary gas costs.

Recommendation: Remove the `require(defaultDepositTolerance > 0, "DDT")` check from `createAdapter()` function.

7.2.3 Redundant factory check in `UniV3Vault.initialize`

Context: [UniV3Vault.sol#L20-L32](#)

Description: The `UniV3Vault.initialize()` function overrides the base `Vault.initialize()` and includes a factory check `require(msg.sender == factory, "NF")`. The base `Vault.initialize()` function already performs access control through the `onlyFactory` modifier or similar checks. The additional factory check in `UniV3Vault.initialize()` is redundant and adds unnecessary gas costs.

Recommendation: Remove the redundant `require(msg.sender == factory, "NF")` check from `UniV3Vault.initialize()`. The base contract's initialization logic should handle access control. If additional access control is needed, it should be implemented through modifiers rather than inline `require` statements.

7.2.4 Potentially unnecessary permission check in `RestrictedVaultFactory`

Context: [RestrictedVaultFactory.sol#L23-L28](#), [VaultFactory.sol#L205-L206](#), [VaultFactory.sol#L229-L234](#)

Description: The `RestrictedVaultFactory._checkInitializePermissions()` function overrides the base implementation to restrict initialization to `onlyOwner`. This function may be unnecessary because:

1. The `createVault()` function is already restricted to `onlyOwner`.
2. The vault's creator address is checked to be `msg.sender` in the base factory.
3. The restriction is already handled because `creatorAddress` is always `onlyOwner` account.

However, there may be edge cases where this check is needed, such as invalidating un-initialized vaults when the owner address changes.

Recommendation: Review the access control flow to determine if `_checkInitializePermissions()` is truly redundant. If the owner restriction is already enforced through `onlyOwner` on `createVault()` and the base factory's creator check, consider removing this override. However, if there are scenarios where owner changes could affect un-initialized vaults, keep the function but document its purpose clearly.

7.2.5 Ownership renunciation in `RestrictedVaultFactory` should not be possible

Context: [RestrictedVaultFactory.sol#L10](#), [VaultFactory.sol#L14](#)

Description: The `RestrictedVaultFactory` inherits from `Ownable2Step` (via `VaultFactory`), which allows the owner to renounce ownership. If ownership is renounced in `RestrictedVaultFactory`, all actions that require `onlyOwner` (including `createVault()`, `createAdapter()`, and `_checkInitializePermissions()`) would become permanently un-executable. This would effectively brick the factory contract and prevent any future vault or adapter creation.

Recommendation: Consider overriding the `renounceOwnership()` function in `RestrictedVaultFactory` to either:

1. Disable renunciation entirely by reverting, or
2. Add additional safeguards (e.g., time delays, multi-sig requirements) before allowing renunciation

Document the decision clearly and ensure all stakeholders understand the implications of ownership renunciation in this context.

7.3 Gas Optimization

7.3.1 Use cloning for adapter/vault deployments

Context: [VaultFactory.sol#L169-L175](#), [VaultFactory.sol#L275-L280](#)

Description: The factory stores full bytecode for vault and adapter types in storage (`vaultTypeByteCode` and `adapterTypeByteCode` mappings) and deploys new instances via `CREATE`. This approach is gas-inefficient. The factory address could be stored as an immutable variable and made accessible using clones with immutable args via `delegatecall`, significantly reducing deployment costs.

Recommendation: Consider implementing the Clone pattern (e.g., using OpenZeppelin's Clones library) for vault and adapter deployments. Store the factory address as an immutable variable in cloned contracts using immutable args pattern, which can be accessed via `delegatecall`. This would reduce gas costs for each deployment while maintaining the same functionality.

7.3.2 Storing full bytecode instead of boolean value

Context: [VaultFactory.sol#L294-L311](#)

Description: The `VaultFactory` contract stores full bytecode for adapter types in the `adapterTypeByteCode` mapping to determine if an adapter type has been revoked. Storing full bytecode in contract storage is expensive. The contract uses this to check if an adapter type has been revoked (by checking if bytecode length is zero).

A boolean flag would be more gas-efficient for this purpose. The bytecode existence can still be verified within `createVault()` by checking the contract's `codesize` if needed using the bytecode passed as `calldata` to the function instead.

Recommendation: Replace the `adapterTypeByteCode` storage with a boolean mapping (e.g., `mapping(uint256 => bool) public adapterTypeRevoked`) to track revocation status. When calling `createVault()`, the bytecode can be passed as an argument to the function because `calldata` is significantly cheaper.