33AUDITS & CO.

VII Finance Audit Report

Introduction

A security review of the VII Finance Yield Harvesting Hook protocol was conducted, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Samuel and Mahdi as Security Researchers.

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 vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

Commit: 2f54c5011a051c5000afc5203be785cf078359b7- Scope: `YieldHarvestingHook`, `ERC4626VaultWrapperFactory`, `ERC4626VaultWrapper`, `AaveWrapper`

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 30 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 6 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About VII Finance Yield Harvesting Hook

This audit is being performed on the core protocol contracts for VII Finance's Yield Harvesting Hook. The protocol introduces a novel Uniswap V4 hook that allows liquidity providers to earn both traditional trading fees and interest from any lending protocol simultaneously. The contracts in scope include `YieldHarvestingHook`, `ERC4626VaultWrapperFactory`, `ERC4626VaultWrapper`, and `AaveWrapper`.

The protocol achieves dual yield by wrapping yield-bearing tokens (such as ERC4626 vault shares or Aave aTokens) into "VII Wrapped" tokens, which separate the principal from the interest. Before each liquidity add, remove, or swap, the hook donates the accrued interest to the pool, allowing active LPs to benefit from the interest on top of the swap fees.

Severity Definitions

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[L-01]	Hash Collision Vulnerability in Salt Generation Using abi.encodePacked	Low	Acknowledged
[L-02]	Deterministic Vault Creation Vulnerable to Front-Running Attacks	Low	Acknowledged

Low

[L-01] Hash Collision Vulnerability in Salt Generation Using abi.encodePacked

Description

The salt generation functions in `ERC4626VaultWrapperFactory` and related wrapper factories use `abi.encodePacked()` instead of `abi.encode()` to generate deterministic salts for CREATE2 deployments. This creates a vulnerability where different input parameters can produce identical hash values, leading to address collisions and potential wrapper overwrites.

The vulnerability affects all wrapper and oracle deployments, making the entire protocol susceptible to hash collision attacks. While the current implementation uses only static types (addresses and uint256), making collisions less likely, the Solidity documentation recommends using `abi.encode()` unless there's a compelling reason to use `abi.encodePacked()`.

Snippet

```
function _getWrapperSalt(address vault, address unitOfAccount, address
poolAddress)
    internal
    view
    returns (bytes32)
{
    return keccak256(abi.encodePacked(evc, nonFungiblePositionManager,
vault, unitOfAccount, poolAddress));
}
```

Impact

- **Low Impact**: Potential address collisions could lead to wrapper overwrites, but risk is minimal with static types
- **Low Likelihood**: Very unlikely to be exploited given only static types are used in salt generation

Recommendations

- Use `abi.encode()` instead of `abi.encodePacked()` for salt generation
- The Solidity documentation states: "Unless there is a compelling reason, abi.encode should be preferred"
- This change would eliminate the hash collision vulnerability entirely

Example snippet

```
function _getWrapperSalt(address vault, address unitOfAccount, address
poolAddress)
    internal
    view
    returns (bytes32)
{
    return keccak256(abi.encode(evc, nonFungiblePositionManager, vault,
unitOfAccount, poolAddress));
}
```

Status

Acknowledged - The development team acknowledged the mention of this but has chosen to maintain the current implementation using `abi.encodePacked()` for gas optimization and other reasons. The team notes that all vault wrappers are clones with immutable args and they use `abi.encodePacked` consistently throughout the codebase to save gas. Since only static types are used in the salt generation, the collision risk doesn't pose a threat.

[L-02] Deterministic Vault Creation Vulnerable to Front-Running Attacks

Description

The vault creation functions in the codebase are completely deterministic and do not include `msg.sender` in the salt generation. This makes them vulnerable to front-running attacks where malicious actors can monitor the mempool for vault creation transactions and deploy the same vault with identical parameters before the legitimate user, causing their transaction to fail.

The salt generation functions only depend on vault parameters, not the user creating them, allowing anyone to compute the exact address where a vault will be deployed.

Snippet

```
// ERC4626VaultWrapperFactory.sol - No user-specific uniqueness in salt
function _generateSalt(address tokenA, address tokenB, uint24 fee, int24
tickSpacing)
    internal
    pure
    returns (bytes32)
{
    return keccak256(abi.encodePacked(tokenA, tokenB, fee,
tickSpacing));
}
```

Impact

- **Low Impact**: Malicious actors can front-run vault creation transactions, causing legitimate users' transactions to fail
- **Low Likelihood**: While possible, the attack has limited impact as the vault still gets created with the same parameters

Recommendations

- Include `msg.sender` in the salt generation to ensure user-specific uniqueness
- Allow users to provide custom salts as an optional parameter
- Add documentation for integrators about checking if pools are already initialized

Example snippet

```
// Include creator address in salt for uniqueness
function _generateSalt(address tokenA, address tokenB, uint24 fee, int24
tickSpacing)
    internal
    pure
    returns (bytes32)
{
    return keccak256(abi.encodePacked(tokenA, tokenB, fee, tickSpacing,
msg.sender));
}
```

Status

Acknowledged - The development team clarified that protection against front-running attacks already existed in the current contracts through the `beforeInitialize` hook that fails if the sender is anyone other than the factory, guaranteeing that front-running is not possible in the core contracts. The team added a comment for integrators about checking if pools are already initialized before calling create methods.

Conclusion

This security audit of the VII Finance Yield Harvesting Hook protocol identified two low severity issues: one related to hash collision vulnerability in salt generation using `abi.encodePacked()` and another related to deterministic vault creation vulnerable to front-running attacks. The development team has acknowledged both issues, choosing to maintain the current implementations for gas optimization and architectural reasons, while adding documentation for integrators. The protocol demonstrates good security practices overall with minimal vulnerabilities identified.

This report was prepared by 33Audits & Co and represents our independent security assessment of the VII Finance Yield Harvesting Hook protocol smart contracts.