JAN.24

SECURITY REVIEW REPORT FOR SPOOL

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

The audit conducted focused on the new Convex StFrxEth strategy smart contract for the Spool Protocol V2. The Spool Protocol V2 acts as a DeFi middleware that enables users to participate in a specific set of yield-generating strategies. In the Convex StFrxEth strategy, ETH is swapped for stETH and frxETH tokens, which are subsequently added to the Curve st-frxEth pool. Additionally, LP tokens are staked on Convex.

Our security assessment involved a comprehensive review of the strategy smart contract, spanning a total of 1 week. During this audit, we identified one high-severity vulnerability, one low-severity vulnerability, and a few informational issues.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality of the project have increased after the completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/SpoolFi/spool-v2-core/pull/11/commits/1a7939f8380ed62235b44e2af72c528f7898ec37

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

AUDITING DETAILS

STARTED

22.01.2024

DELIVERED

26.01.2024

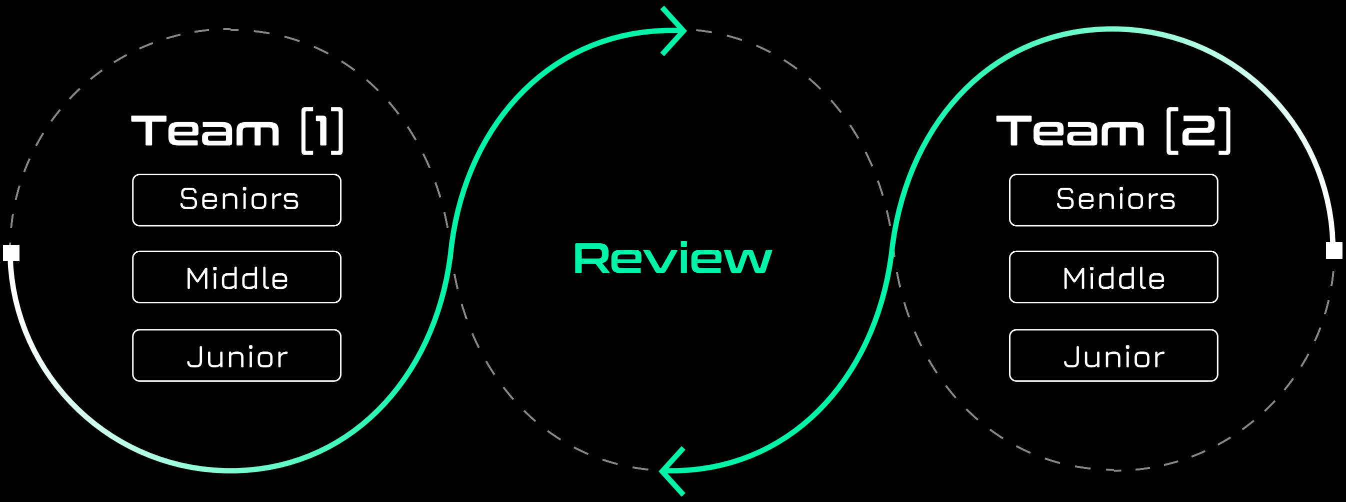
Review Led by

MIKHAIL EGOROV

Lead Smart Contract Auditor | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

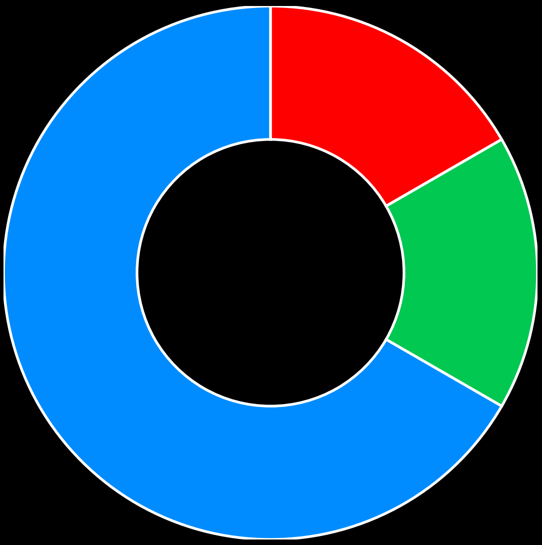
ISSUE SYMBOLIC CODES

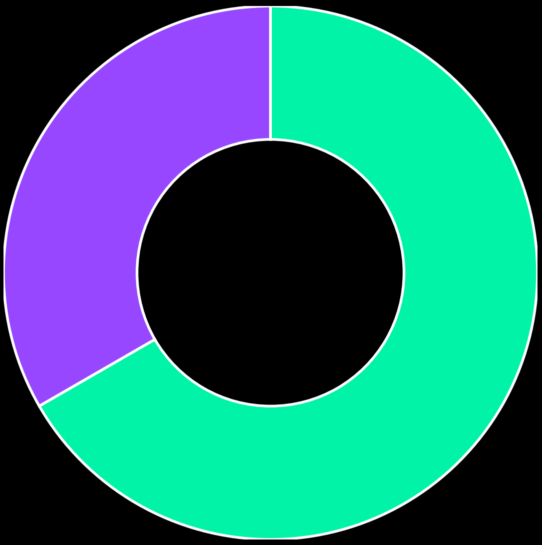
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	0
Low	1
Informational	4

Total: 6





WEAKNESSES

This section contains the list of discovered weaknesses.

SPOOL2-2

DISCREPANCIES IN _REDEEMFROMPROTOCOL FUNCTION IMPLEMENTATION

SEVERITY: High

PATH:

src/strategies/convex/ConvexStFrxEthStrategy.sol

REMEDIATION:

refine the _redeemFromProtocol() implementation to be consistent with the specification

STATUS: Fixed

DESCRIPTION:

The implementation of the **_redeemFromProtocol()** function does not adhere to the documentation on how slippage entries should be packed into the **slippages** array.

According to the breakdown in the documentation, the reallocation requires the use of four slippage values from withdrawalSlippages within the **_redeemFromProtocol()** function.

```
withdrawalSlippages
----------------
1: ssts check
2, 3: removeLiquidity: slippages
4, 5: steth/frxeth: slippages
```

However, the **_redeemFromProtocol()** function uses **0** slippages for removing liquidity by specifying an uninitialized array instead of an array with the 2nd and 3rd elements from the **slippages** array.

```
uint256[] memory amounts = _redeemInner(ssts, new uint256[](2));
```

As a result, incorrect slippage values **slippages**[2] and **slippages**[3] are used for unwrapping **stETH** and **frxETH** tokens into **ETH** instead of **slippages**[4] and **slippages**[5].

```
uint256 bought = _assetGroupUnwrap(amounts, slippages, slippageOffset);
```

This incorrect implementation may lead to the absence of slippage protection for **stETH** and **frxETH** unwrapping in the **_redeemFromProtocol()** function if the remove liquidity slippages are deliberately set to 0.

Additionally, according to the documentation, the withdrawal utilizes only the **slippages**[6] element.

```
-- breakdown --
- 1: amount check (eth)
- 2, 3: balance checks (pool)
- 4: ssts check (beforeRedeemalCheck)
- 5: compound
- 6: weth output: slippage
```

However, the _assetGroupUnwrap() function consumes two elements from the slippages array starting from index 6, since the strategy utilizes two tokens, frxETH and stETH. As a result, the call will revert either 0 slippage is used for frxETH unwrapping.

```solidity
function  _redeemFromProtocol(address[]  calldata,  uint256  ssts,  uint256[]
calldata slippages) internal override {
    uint256 slippageOffset;
    if (slippages[0] == 1) {
        slippageOffset = 6;
    } else if (slippages[0] == 2) {
        slippageOffset = 2;
    } else if (slippages[0] == 3) {
        slippageOffset = 1;
    } else if (slippages[0] == 0 && _isViewExecution()) {
        slippageOffset = 6;
    } else {
        revert ConvexStFrxEthRedeemSlippagesFailed();
    }

    uint256[] memory amounts = _redeemInner(ssts, new uint256[](2));
    uint256 bought = _assetGroupUnwrap(amounts, slippages, slippageOffset);

    if (_isViewExecution()) {
        emit Slippages(false, bought, "");
    }
}
```

SPOOL2-3

CONVEXSTFRXETHSTRATEGY CALCULATES INCORRECT USD WORTH AMOUNT

SEVERITY: Low

PATH:

src/strategies/convex/ConvexStFrxEthStrategy.sol

REMEDIATION:

apply separate exchange rates for stETH and frxETH when calculating _getUsdWorth()

STATUS: Acknowledged, see commentary

DESCRIPTION:

The **ConvexStFrxEthStrategy** strategy swaps **WETH** into **stETH** and **frxETH** tokens and further utilizes them by adding liquidity in the **stETH-frxETH** Curve.fi pool.

Within the **_getTokenWorth()** function, the amounts of **stETH** and **frxETH** belonging to the strategy are summed up together into the amount variable.

```
function _getTokenWorth() internal view returns (uint256[] memory amount) {
    amount = new uint256[](1);

    // convex
    uint256 lpTokenAmount = _crvRewards.balanceOf(address(this));

    // curve base
    uint256 lpTokenTotalSupply = IERC20(_pool).totalSupply();

    amount[0] = _balances(0) * lpTokenAmount / lpTokenTotalSupply;
    amount[0] += _balances(1) * lpTokenAmount / lpTokenTotalSupply;

    return amount;
}
```

Furthermore, inside the **_getUsdWorth()** function, the USD worth amount is calculated by applying a single exchange rate from **exchangeRates[0]**, assuming that **stETH** and **frxETH** have the same price in USD. However, this assumption is incorrect.

```
function _getUsdWorth(uint256[] memory exchangeRates, IUsdPriceFeedManager
priceFeedManager)
    internal
    view
    override
    returns (uint256)
{
        return priceFeedManager.assetToUsdCustomPrice(weth, _getTokenWorth()[0],
exchangeRates[0]);
}
```

As a result, the _getUsdWorth() function returns an incorrect value when called from Strategy.doHardWork().

Commentary from the client:

“ - On review of this issue, we have decided to not implement a change. The assumption that frxETH and stETH have equivalent value (or close to) to WETH is a strong enough assumption for the functioning of the protocol. Any significant deviation has a very low probability of occuring.

The other technical aspect of why we cannot change this is to do with the architecture of the doHardWork: WETH is the asset group of this strategy, and the way it is architected is that we can only pass the exchange rate for the asset group tokens to getUsdWorth. As a result we can only pass the WETH price, without significantly modifying core code. This is how we implement strategies SfrxEthHoldingStrategy and StEthHoldingStrategy; WETH is always the exchange rate used, as the asset group token is WETH.”

SPOOL2-6

RETURN VALUE OF LIDO.SUBMIT OF ISN'T CHECKED

SEVERITY: Informational

PATH:

src/strategies/libraries/EthStEthAssetGroupAdapter.sol

REMEDIATION:

check the returned value of lido.submit()

STATUS: Acknowledged, see commentary

DESCRIPTION:

In case **stETH** tokens are minted from the Lido contract, the **lido.submit()** function returns uint256 amount of **stETH** minted. The amount of **stETH** may vary since it represents shares in the contract.

However, the returned value isn't checked for slippage inside the **wrap()** function.

```
function wrap(uint256 amount, uint256 slippage) public returns (uint256
bought) {
if (slippage == type(uint256).max) {
_stake(amount);
return amount;
}
bought = _buyOnCurve(amount, slippage);
}
```

```
function _stake(uint256 amount) private {
lido.submit{value: amount}(address(this));
}
```

Commentary from the client:

" - On review of this issue, we have decided not to implement a change. While we don't explicitly check the result of lido.submit, the output value from this call is fed into add_liquidity, which we do check slippage on.

This pattern is repeated in the redeem flow, where we only check the output weth amount. Given this, we think that sufficient slippage is already in place."

SPOOL2-1

MISSING ADDRESS ZERO CHECK

SEVERITY: Informational

PATH:

ConvexStFrxEthStrategy.sol: constructor:L119-125

REMEDIATION:

consider adding address zero check

STATUS: Fixed

DESCRIPTION:

The constructor of **ConvexStFrxEthStrategy** lacks zero address validation since **_swapper** is used in other functions of the contract, an error in this state variable can lead to the redeployment of the contract.

```
constructor(
IAssetGroupRegistry assetGroupRegistry_,
ISpoolAccessControl accessControl_,
uint256 assetGroupId_,
ISwapper swapper_
)      Strategy(assetGroupRegistry_,      accessControl_,      assetGroupId_)
WethHelper(_weth) {
_swapper = swapper_;
}
```

SPOOL2-4

INTERNAL FUNCTION ISN'T USED

SEVERITY: Informational

PATH:

ConvexStFrxEthStrategy.sol:_getTokenWorth():L398

REMEDIATION:

consider using _lpTokenBalance()

STATUS: Fixed

DESCRIPTION:

The **_lpTokenBalance()** function is an internal function that retrieves the balance of LP tokens held by the contract. In the **_getTokenWorth** function, the LP token balance is obtained through a direct call.

It is recommended to follow a standardized approach for retrieving the LP token balance, as it improves code readability and maintainability.

```
function _getTokenWorth() internal view returns (uint256[] memory amount) {
amount = new uint256[](1);

// convex
uint256 lpTokenAmount = _crvRewards.balanceOf(address(this));
```

```
function _lpTokenBalance() internal view returns (uint256) {
        return _crvRewards.balanceOf(address(this));
    }
```

SPOOL2-5

MISSING CLAIM OF EXTRA CVX REWARD

SEVERITY: Informational

REMEDIATION:

declare _extraRewards as a variable instead of a constant. Make it adjustable through initialize()

STATUS: Fixed

DESCRIPTION:

The **_getRewards()** doesn't claim an extra reward in the Convex reward pool since the **_extraRewards** constant is **false**. As a result, the strategy might generate less yield if an additional reward in **CVX** is distributed in the pool.

```
bool private constant _extraRewards = false;
```

```
function _getRewards() private returns (address[] memory) {
// get CRV and extra rewards
_crvRewards.getReward(address(this), _extraRewards);

address[] memory rewardTokens = new address[](2);
rewardTokens[0] = _crvRewardToken;
rewardTokens[1] = _cvxRewardToken;

return rewardTokens;
}
```

hexens × Spool