hexens × Spool

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 web3 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, Coinstats, 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.

AUDIT LED BY

MIKHAIL EGOROV

Lead Smart Contract Auditor | Hexens

Audit Starting Date
03.01.2024

Audit Completion Date
08.01.2024

---

hexens × Spool

METHODOLOGY

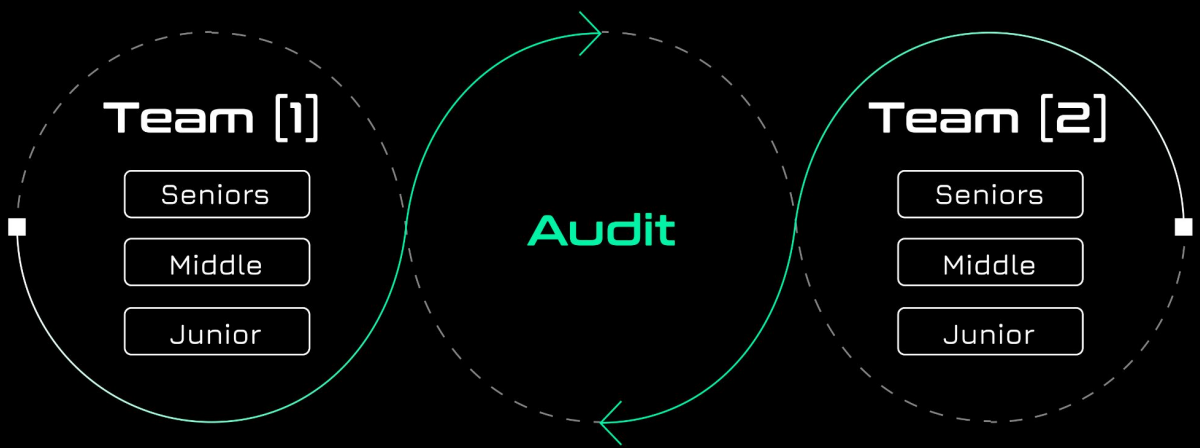
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

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.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of smart contracts. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

The audit conducted focused on the new OETH strategy smart contract for the Spool Protocol V2. The Spool Protocol V2 acts as a DeFi middleware enables users to participate in a specific set of yield-generating strategies. In the OETH strategy, ETH is staked in Origin Ether (OETH) and is then invested in various liquid staking derivative strategies.

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 and one informational issue. The high-severity vulnerability affected the calculation of the accrued yield percentage, leading to incorrect results.

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/tree/c99f190432981ee9a16a2359abded925f63e09bc

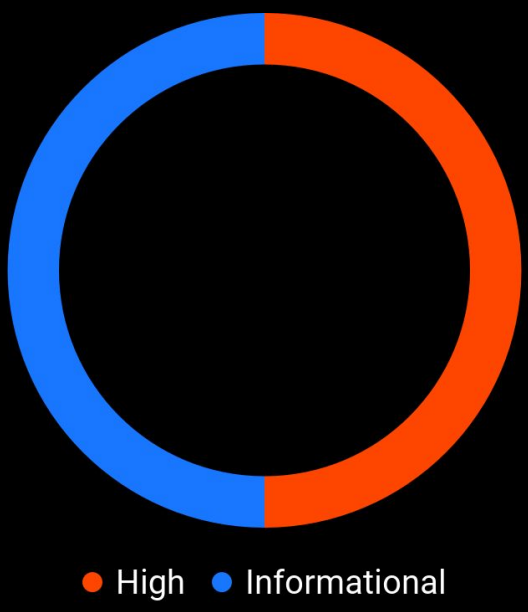
The issues described in this report were fixed in the following commit:
https://github.com/SpoolFi/spool-v2-core/commit/36336ca3db7a5eaf11c0dee9e90c61db3bc7b801

SUMMARY

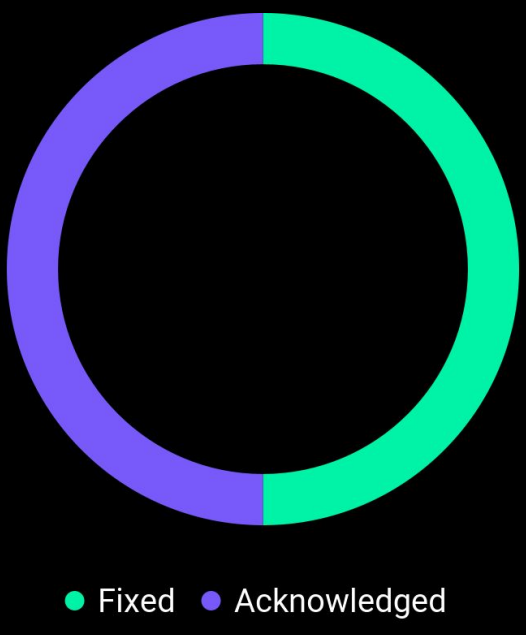
SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	1
MEDIUM	0
LOW	0
INFORMATIONAL	1

TOTAL: 2

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

SPOOL-2. _GETYIELDPERCENTAGE INCORRECTLY CALCULATES YIELD PERCENTAGE

SEVERITY: High

PATH: src/strategies/OEthHoldingStrategy.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The **_getYieldPercentage()** function returns a higher yield percentage than actual accrued yield for **OETH**. As a result, an excessive amount of reward is distributed inside **Strategy.doHardWork()**.

When the **OETH** token accrues interest and the rebasing happens, the value of **_rebasingCreditsPerToken** decreases, so holders have more **OETH** on their balances.

To calculate the percentage inside **_getYieldPercentage()**, the last and current low-resolution values of **_rebasingCreditsPerToken** are passed to **Strategy._calculateYieldPercentage()**. Notably, function arguments are swapped. So, the last value is passed as **currentValue** parameter, and the current value is passed as **previousValue**.

We can summarize the **_getYieldPercentage()** function as the following formula:

```
percentage = (last_rebasingCreditsPerToken - current_rebasingCreditsPerToken) /
current_rebasingCreditsPerToken
```

while it should be

```
percentage = (last_rebasingCreditsPerToken - current_rebasingCreditsPerToken) /
last_rebasingCreditsPerToken
```

Suppose there is a **20%** yield, and **_rebasingCreditsPerToken** changed by **20%** from **934619183513304865** to **74769323977070863**1.

The first formula gives **25%**

```
(934619183513304865-74769323977070863) / 74769323977070863 = 0.25
```

While the correct one gives **20%**

```
(934619183513304865-74769323977070863) / 934619183513304865 = 0.2
```

The **test_getYieldPercentage()** test contains an error. It assumes that **10e18 / 12e18** is a **20%** decrease while it's a **16.6%** decrease.

```
// - generate ~20% yield (reduce credits per token by 20%)
uint256 priceBefore = oEthVault.priceUnitMint(address(tokenWeth));
vm.record();
uint256 rebasingCreditsPerToken = oEthToken.rebasingCreditsPerTokenHighres();
(bytes32[] memory reads,) = vm.accesses(address(oEthToken));
vm.store(address(oEthToken), reads[1], bytes32(rebasingCreditsPerToken * 10e18 / 12e18));
```

Changing it to **8e18 / 10e18** fails the test.

```
function _getYieldPercentage(int256) internal override returns (int256 baseYieldPercentage) {
  uint256 rebasingCreditsPerTokenCurrent = oEthToken.rebasingCreditsPerToken();

  baseYieldPercentage = _calculateYieldPercentage(rebasingCreditsPerTokenCurrent,
_rebasingCreditsPerTokenLast);

  _rebasingCreditsPerTokenLast = rebasingCreditsPerTokenCurrent;
}
```

```
function _calculateYieldPercentage(uint256 previousValue, uint256 currentValue)
  internal
  pure
  returns (int256 yieldPercentage)
{
  if (currentValue > previousValue) {
     yieldPercentage = int256((currentValue - previousValue) * YIELD_FULL_PERCENT / previousValue);
  } else if (previousValue > currentValue) {
     yieldPercentage = -int256((previousValue - currentValue) * YIELD_FULL_PERCENT / previousValue);
  }
}
```

Change the following line inside the **_getYieldPercentage()** function.

```
- baseYieldPercentage = _calculateYieldPercentage(rebasingCreditsPerTokenCurrent,
_rebasingCreditsPerTokenLast);
+ baseYieldPercentage = -_calculateYieldPercentage(_rebasingCreditsPerTokenLast,
rebasingCreditsPerTokenCurrent);
```

SPOOL-1. CONSTANT AND IMMUTABLE VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: src/strategies/OEthHoldingStrategy.sol

REMEDIATION: see description

STATUS: acknowledged, see commentary

DESCRIPTION:

The parameters on lines 65-70 should be **private**. Setting constants and immutables to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the values outside of where it's used, and won't add another entry to the method ID table. The values can still be read from the verified contract source code if necessary.

```
int128 public constant CURVE_ETH_POOL_ETH_INDEX = 0;
int128 public constant CURVE_ETH_POOL_OETH_INDEX = 1;
IOEthToken public immutable oEthToken;
IVaultCore public immutable oEthVault;
ICurveEthPool public immutable curve;
```

```
-   int128 public constant CURVE_ETH_POOL_ETH_INDEX = 0;
-   int128 public constant CURVE_ETH_POOL_OETH_INDEX = 1;
-   IOEthToken public immutable oEthToken;
-   IVaultCore public immutable oEthVault;
-   ICurveEthPool public immutable curve;
+   int128 private constant CURVE_ETH_POOL_ETH_INDEX = 0;
+   int128 private constant CURVE_ETH_POOL_OETH_INDEX = 1;
+   IOEthToken private immutable oEthToken;
+   IVaultCore private immutable oEthVault;
+   ICurveEthPool private immutable curve;
```

Commentary from the client:

“ - We acknowledge that we are creating public variables here and that it increases the deployment cost, but we are prepared to absorb this relatively small cost. It makes integration for other parts of our stack easier, and also maintains consistency across other strategy implementations, which implement these values the same way.”