shieldify

Stax

SECURITY REVIEW

Date: 14 March 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Stax

Stax is the latest addition to the Element 280 "Easy Button" Ecosystem, designed to simplify and enhance participation in the TitanX staking landscape.

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review was conducted in the span of 7 days and was part of Shieldify's Private Pool competitions.

Overall, the code is written to very high standards and implements very solid security practices. A total of 27 researchers participated in the Shieldify Private Pool competition, identifying two low-severity issues related to incorrect swapping paths and hardcoded limits, along with two informational findings.

Shieldify's team extends their gratitude to Stax's team for their exceptional work, communication and responsiveness.

5.1 Protocol Summary

Project Name	**Stax**
Repository	Stax
Type of Project	DeFi, Staking
Audit Timeline	7 days
Review Commit Hash	c9a4ac0a7a361719c01de6af43562d3b4bb639da
Fixes Review Commit Hash	83f1ed3315f7e933819a780a523b27c179836e90

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/OrxStax/OrxStax.sol	307
Total	**307**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Low** issues: **2**
- **Info** issues: **2**

ID	Title	Severity	Status
[L-01]	`_swapUsdxToTitanX()` Is Incapable of Swapping USDx to TitanX Due to Incorrect Path	Low	Fixed
[L-02]	Pool Fee Tiers Hardcoded to `10000` Limiting Swap Flexibility	Low	Fixed
[I-01]	`OrxStax.setSecondAgo()` Function Should Only Allow Values Greater Than +N Block Time	Info	Acknowledged
[I-02]	`amountIn` is Cast As `uint128` While OracleLibrary Expects to Receives It As `uint256`	Info	Fixed

7. Findings

[L-01] `_swapUsdxToTitanX()` Is Incapable of Swapping USDx to TitanX Due to Incorrect Path

Severity

Low Risk

Description

When `distributeUsdx()` is called to send the USDx in the contract to the `Stax Vaul` contract address, it uses `_swapUsdxToTitanX()` to multihop swap from USDx to TitanX using USDC and WETH. The problem is that the hardcoded path for this swap is incorrect, as some of the pools do not exist, so the swap will always fail. The path is hardcoded to look for pools with a 1% fee, but some of the pools only have 0.05% fee for that token pair, for example.

For USDX to USDC: Uniswap only has a 0.05% V3 pool. For USDC to WETH: Uniswap only has 0.01%, 0.3%, 0.05%. For WETH to TitanX: Uniswap actually has a 1% V3 pool.

Looking at this and comparing with the hardcoded `POOL_FEE_1PERCENT` enforced for all the swaps in the path, the swap will always fail.

Location of Affected Code

File: contracts/OrxStax/OrxStax.sol#L359

```
function _swapUsdxToTitanX(uint256 amountIn, uint256 minAmountOut,
   uint256 deadline) internal {
    bytes memory path = abi.encodePacked(USDX, POOL_FEE_1PERCENT, USDC,
       POOL_FEE_1PERCENT, WETH9, POOL_FEE_1PERCENT, TITANX);

    ISwapRouter.ExactInputParams memory params = ISwapRouter.
       ExactInputParams({
        path: path,
        recipient: STAX_VAULT,
        deadline: deadline,
        amountIn: amountIn,
        amountOutMinimum: minAmountOut
    });
    IERC20(USDX).safeIncreaseAllowance(UNISWAP_V3_ROUTER, amountIn);
    ISwapRouter(UNISWAP_V3_ROUTER).exactInput(params);
}
```

Impact

The function `_swapUsdxToTitanX()` will always fail, making it impossible to send staking rewards to the Stax Vault. Therefore, staking rewards will be stuck in the `OrxStax` contract.

Recommendation

Consider applying the following changes: - Replace `POOL_FEE_1PERCENT` with `500` for 0.05% for USDX to USDC. - Then replace `POOL_FEE_1PERCENT` with `100` for 0.01% or `3000` 0.3% or `500` 0.05% for USDC to WETH.

Team Response

Fixed.

[L-02] Pool Fee Tiers Hardcoded to `10000` Limiting Swap Flexibility

Severity

Low Risk

Description

The `_swapTitanXToOrx()` function hardcodes the Uniswap V3 pool fee to 1000 (1%) in the ExactInputSingleParams struct. While 1% is a common fee tier, hardcoding this value restricts the contract's ability to utilize pools with different fee tiers like 0.3% (300) or 0.05% (500). This design choice limits the contract's versatility in interacting with various Uniswap V3 liquidity pools.

Location of Affected Code

File: contracts/OrxStax/OrxStax.sol#L343

```
function _swapTitanXToOrx(uint256 amountIn, uint256 minAmountOut, uint256
    deadline) internal returns (uint256) {
    _twapCheck(TITANX, ORX, amountIn, minAmountOut, ORX_TITANX_POOL,
       orxDeviation);
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter.
       ExactInputSingleParams({
        tokenIn: TITANX,
        tokenOut: ORX,
        fee: POOL_FEE_1PERCENT, // @audit
        recipient: address(this),
        deadline: deadline,
        amountIn: amountIn,
        amountOutMinimum: minAmountOut,
        sqrtPriceLimitX96: 0
    });
    IERC20(TITANX).safeIncreaseAllowance(UNISWAP_V3_ROUTER, amountIn);
    return ISwapRouter(UNISWAP_V3_ROUTER).exactInputSingle(params);
}
```

Impact

Users are restricted to pools with 1% fees only, preventing access to potentially more cost-efficient pools or pools that only exist in other fee tiers. This could result in higher transaction costs or failed transactions if the desired trading pair isn't available in the 1% fee tier.

Recommendation

Make the fee parameter configurable by modifying the function to accept a fee parameter:

```
- function _swapTitanXToOrx(uint256 amountIn, uint256 minAmountOut,
   uint256 deadline) internal returns (uint256) {
+ function _swapTitanXToOrx(uint256 amountIn, uint256 minAmountOut,
   uint256 deadline, uint24 _poolFee) internal returns (uint256) {
    _twapCheck(TITANX, ORX, amountIn, minAmountOut, ORX_TITANX_POOL,
       orxDeviation);
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter.
       ExactInputSingleParams({
        tokenIn: TITANX,
        tokenOut: ORX,
-       fee: POOL_FEE_1PERCENT,
+       fee: _poolFee,
        recipient: address(this),
        deadline: deadline,
        amountIn: amountIn,
        amountOutMinimum: minAmountOut,
        sqrtPriceLimitX96: 0
    });
    IERC20(TITANX).safeIncreaseAllowance(UNISWAP_V3_ROUTER, amountIn);
    return ISwapRouter(UNISWAP_V3_ROUTER).exactInputSingle(params);
}
```

Team Response

Fixed.

[I-01] `OrxStax.setSecondAgo()` Function Should Only Allow Values Greater Than +N Block Time

Severity

Info

Description

The `OrxStax::setSecondAgo` allow admins to set the `secondsAgo` state variable, which defines the minimum time window that can be used when reading the TWAP price. Right now, any value above 0 is valid, but that means that a value less than a block time can be set, which defeats the principle of a TWAP. The lowest value should be at least 2 block time minimum, even though it is highly recommended to measure the price for a longer block time.

Location of Affected Code

File: contracts/OrxStax/OrxStax.sol

```
function setSecondsAgo(uint32 limit) external onlyOwner {
@>  if (limit == 0) revert ZeroInput();
    secondsAgo = limit;
}
```

```
function _twapCheck(address tokenIn, address tokenOut, uint256 amountIn,
   uint256 minAmountOut, address poolAddress, uint32 deviation) internal
   view {
@> uint32 _secondsAgo = secondsAgo;
    uint32 oldestObservation = OracleLibrary.
       getOldestObservationSecondsAgo(poolAddress);
    if (oldestObservation < _secondsAgo) {
        _secondsAgo = oldestObservation;
    }

@> (int24 arithmeticMeanTick,) = OracleLibrary.consult(poolAddress,
   _secondsAgo);
```

Impact

Parameter could allow TWAP price manipulation

Recommendation

Consider applying the following change:

```
// @notice Sets the number of seconds to look back for TWAP price
   calculations.
// @param limit The number of seconds to use for TWAP price lookback.
function setSecondsAgo(uint32 limit) external onlyOwner {
-    if (limit == 0) revert ZeroInput();
+    if (limit < minTimeWindow) revert ZeroInput();
     secondsAgo = limit;
}
```

Team Response

Acknowledged.

[I-02] `amountIn` is Cast As `uint128` While OracleLibrary Expects to Receives It As `uint256`

Severity

Info

Description

In the `_twapCheck()` the `amountIn` is passed into `OracleLibrary.getQuoteForSqrtRatioX96()` along with other variables to get `twapAmountOut`. The problem here is that before passing the amountIn to the `Oraclelibrary` it is cast as a `unit128` while it is a uint256 variable, The `OracleLibrary` is expecting to receive it as a uint256 variable in its contract. Therefore casting as a uint128 is pointless and could slightly affect the accuracy of the resulting `twapAmountOut`.

Location of Affected Code

File: contracts/OrxStax/OrxStax.sol#L383

```
uint256 twapAmountOut = OracleLibrary.getQuoteForSqrtRatioX96(
   sqrtPriceX96, uint128(amountIn), tokenIn, tokenOut);
```

File contracts/OrxStax/lib/OracleLibrary.sol#L157

```
function getQuoteForSqrtRatioX96(uint160 sqrtRatioX96, uint256 baseAmount
   , address baseToken, address quoteToken)
    internal
    pure
    returns (uint256 quoteAmount)
```

Impact

Accuracy of `twapAmountOut` could be affected, but risk is low as it won't pose a significant threat to the protocol.

Recommendation

Remove the `uint128(amountIn)` cast and leave it as `amountIn`.

Team Response

Fixed.

shieldify

Thank you!