our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Vyper

SECURITY REVIEW

Date: 7 November 2024

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, Rust, Go, Vyper, Move and Cairo.

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 Vyper

Powering a Lightning-Fast deflation with Electrifying buy and burn

Volt, built on TitanX, is a hyper-deflationary token with a unique auction system. It features a capped supply with all tokens distributed in the first 10 days, triggering full deflation afterward. Volt utilizes 80% of system value to buy tokens + 8% of value for growing bonded liquidity growth. Volt enters deflation quickly with a massive buy and burn.

Learn more about Volt's concept and the technicalities behind it here.

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 lasted 7 days with a total of 224 hours dedicated by 4 researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying two Medium and several Low-severity issues, including the absence of a refund mechanism, incorrect call sequences that lead to miscalculations, improper downcasting, and insufficient input validation and others.

The Vyper team has done an excellent job with their security approach, conducting four security reviews with various companies and solo researchers.

5.1 Protocol Summary

Project Name	**Vyper**
Repository	vyper
Type of Project	DeFi, Staking
Audit Timeline	7 days
Review Commit Hash	9d03eb6ac3119460650d7fb3e656e4dc59cd0b5f
Fixes Review Commit Hash	cfc0f0a7618411aa326b1256ccf207d37d65b4fe

5.2 Scope

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

File	**nSLOC**
src/actions/SwapActions.sol	77
src/nexus/VyperDragonXNexus.sol	162
src/nexus/DragonXVoltNexus.sol	164
src/nexus/DragonXVoltInput.sol	161
src/nexus/VoltVyperNexus.sol	171
src/src/Auction.sol	197
src/const/Constants.sol	19
src/Vyper.sol	91
src/VyperTreasury.sol	28
Total	**1070**

6. Findings Summary

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

- **Medium** issues: **2**
- **Low** issues: **8**

ID	Title	Severity	Status
[M-01]	Excess Liquidity Is Not Returned and Approval Is Not Revoked When Minting Positions in `Auction`	Medium	Fixed
[M-02]	`distributeDragonXForBurning()` Should Call Interval Update Before Transferring Tokens	Medium	Acknowledged
[L-01]	Improper Downcasting in `getDailyDragonXAllocation()`	Low	Fixed
[L-02]	Wrong Variable Name Used in `DragonXVoltNexus`	Low	Fixed
[L-03]	Deadline is Commented Out in `SwapActions.swapExactInput()`	Low	Fixed
[L-04]	`INTERVAL_TIME` Is Off By 1 Second	Low	Acknowledged
[L-05]	`lastIntervalNumber` Is Off By One	Low	Acknowledged
[L-06]	8% of Volt is Not Sent to Treasure Volt as Stated in the Documentation	Low	Acknowledged
[L-07]	The `startTimestamp` in `getCurrentInterval()` Is Not Checked	Low	Acknowledged
[L-08]	Users Should Not be Able to Get Vyper Until Initial Liquidity is Minted in The Auction	Low	Acknowledged

7. Findings

[M-01] Excess Liquidity Is Not Returned and Approval Is Not Revoked When Minting Positions in `Auction`

Severity

Medium Risk

Description

`Auction.addInitialLiquidity()` calls `INonfungiblePositionManager(v3PositionManager).mint()`, but does not revoke excess approvals and return extra funds:

```
(uint256 tokenId,,,) = INonfungiblePositionManager(v3PositionManager).
   mint(params);

LP memory newLP = LP({hasLP: true, tokenId: uint240(tokenId),
   isVyperToken0: sortedToken0 == address(vyper)});

if (isVoltPool) {
    lpPools.vyperVolt = newLP;
} else {
    lpPools.vyperDragonX = newLP;
}
```

When calling `nonfungiblePositionManager.mint()`, check whether there are excess funds left after minting, like how it is done in this example:

```
// Remove allowance and refund in both assets.
if (amount0 < amount0ToMint) {
    TransferHelper.safeApprove(
        DAI,
        address(nonfungiblePositionManager),
        0
    );
    uint refund0 = amount0ToMint - amount0;
    TransferHelper.safeTransfer(DAI, msg.sender, refund0);
}

if (amount1 < amount1ToMint) {
    TransferHelper.safeApprove(
        USDC,
        address(nonfungiblePositionManager),
        0
    );
    uint refund1 = amount1ToMint - amount1;
    TransferHelper.safeTransfer(USDC, msg.sender, refund1);
}
```

Another example.

Impact

Funds left after minting a position will be stuck in the contract.

Recommendation

It's recommended to revoke approvals and refund excess tokens:

Sample example:

```
(uint256 tokenId,,amount0,amount1) = INonfungiblePositionManager(
   v3PositionManager).mint(params);
IERC20(sortedToken0).approve(address(v3PositionManager), 0);
IERC20(sortedToken1).approve(address(v3PositionManager), 0);

if(amount0 < amount0Sorted){
     IERC20(sortedToken0).transfer(owner, amount0Sorted - amount0)
}
if(amount1 < amount0Sorted){
     IERC20(sortedToken1).transfer(owner, amount1Sorted - amount1)
}
```

Team Response

Fixed.

[M-02] `distributeDragonXForBurning()` Should Call Interval Update Before Transferring Tokens

Severity

Medium Risk

Description

In `DragonXVoltInput.distributeDragonXForBurning()`, dragonX is first transferred into the contract before `_intervalUpdate()` is called. In the nexus contracts, `_intervalUpdate()` is called first before transferring tokens.

The comments also mention that the accumulated allocation should be updated before depositing new DragonX.

```
function distributeDragonXForBurning(uint256 _amount) external notAmount0
   (_amount) {
///@dev - If there are some missed intervals update the accumulated
   allocation before depositing new DragonX

    dragonX.safeTransferFrom(msg.sender, address(this), _amount);

    if (Time.blockTs() > startTimeStamp && Time.blockTs() -
       lastBurnedIntervalStartTimestamp > INTERVAL_TIME) {
        _intervalUpdate();
    }
}
```

Impact

The calculation of `_totalAmountForInterval` will not be as intended, especially when `(_totalAmountForInterval + additional > dragonX.balanceOf(address(this)))` in line 272.

Recommendation

Consider switching the transfer line:

```
function distributeDragonXForBurning(uint256 _amount) external notAmount0
   (_amount) {
    ///@dev - If there are some missed intervals update the accumulated
       allocation before depositing new DragonX

-   dragonX.safeTransferFrom(msg.sender, address(this), _amount);

    if (Time.blockTs() > startTimeStamp && Time.blockTs() -
       lastBurnedIntervalStartTimestamp > INTERVAL_TIME) {
        _intervalUpdate();
    }
+   dragonX.safeTransferFrom(msg.sender, address(this), _amount);
}
```

Team Response

Acknowledged.

[L-01] Improper Downcasting in `getDailyDragonXAllocation()`

Severity

Low Risk

Description

The `DragonXVoltInput.getDailyDragonXAllocation()` function has an input parameter of uint32.

```
function getDailyDragonXAllocation(uint32 t) public view returns (uint256
    dailyWadAllocation) {
    uint256 STARTING_ALOCATION = 0.24e18;
    uint256 MIN_ALOCATION = 0.15e18;
    uint256 daysSinceStart = Time.dayGap(startTimeStamp, t);
```

This value is used in Time.dayGap, which uses uint256 instead.

```
function dayGap(uint32 start, uint256 end) public pure returns (uint32
   daysPassed) {
      assembly {
          daysPassed := div(sub(end, start), SECONDS_PER_DAY)
      }
  }
```

Although the `t` value here refers to time, and uint32 is adequate (for +80years more) or so, it is best to use the same data type for consistency.

Recommendation

Use `uint256` in `getDailyDragonXAllocation()` for consistency.

Note: there are other places with inconsistent downcasting as well, like `Auction.deposit()` where `_amount` is downcasted from uint192 to uint128.

Team Response

Fixed.

[L-02] Wrong Variable Name Used in `DragonXVoltNexus`

Severity

Low Risk

Description

The Nexus contracts are similar in code; they just use different tokens to buy and burn.

In `DragonXVoltNexus._calculateInterval()` , the variable `_totalVoltDistributed` is erroneously introduced. The `_calculateIntervals()` function creates a new variable `_totalVoltDistributed` , but it should be `_totalDragonXDistributed` instead.

```
uint256 _totalVoltDistributed = totalDragonXDistributed:

if (currentDay == dayOfLastInterval) {
    uint128 _amountPerInterval = uint128(_totalVoltDistributed /
       INTERVALS_PER_DAY);
    ...
```

In VyperDragonXNexus, `_totalVyperDistributed` is used for totalVyperDistributed:

```
uint256 _totalVyperDistributed = totalVyperDistributed;

if (currentDay == dayOfLastInterval) {
    uint128 _amountPerInterval = uint128(_totalVyperDistributed /
       INTERVALS_PER_DAY);
    ...
```

Recommendation

The `_totalVoltDistributed` should be `_totalDragonXDistributed` instead.

Team Response

Fixed.

[L-03] Deadline is Commented Out in `SwapActions.swapExactInput()`

Severity

Low Risk

Description

The deadline parameter is not used in `SwapActions.swapExactInput` is commented out:

```
 ISwapRouter.ExactInputParams memory params = ISwapRouter.
    ExactInputParams({
        path: path,
        recipient: address(this),
       // deadline: deadline,
        amountIn: tokenInAmount,
        amountOutMinimum: minAmount
  });
```

In some contracts which call `swapExactInput()` like `Auction`, the deadline is checked with the `notExpired` modifier in `Errors`. In other contracts like `VoltVyperNexus.swapVoltToVyperAndDistribute()`, the deadline is not checked internally.

Impact

Without a deadline parameter, the transaction may sit in the mempool and be executed at a much later time potentially resulting in a worse price for the user.

Recommendation

In all functions that call `swapExactInput()`, check the deadline using the `notExpired` modifier. Alternatively, uncomment the deadline in `SwapActions.swapExactInput()`.

Team Response

Fixed.

[L-04] `INTERVAL_TIME` is Off By 1 Second

Severity

Low Risk

Description

The interval time is set to 5 minutes but it can only be called after 5 minutes and 1 second to set the next interval. This issue only exists if the interval is called between 5min - 10min.

At 10 mins, the protocol calculates 2 intervals but at 5 minute, the protocol calculates as 0 intervals.

Recommendation

For more precise calculation, change `>` to `>=` in `getCurrentInterval()`:

```
if (lastBurnedIntervalStartTimestamp == 0 || timeElapseSinceLastBurn >=
   INTERVAL_TIME) {
      (_lastInterval, _amountAllocated, _missedIntervals, beforeCurrday)
         =
      _calculateIntervals(timeElapseSinceLastBurn);

      _lastIntervalStartTimestamp = startPoint;
    _missedIntervals += timeElapseSinceLastBurn >= INTERVAL_TIME &&
       lastBurnedIntervalStartTimestamp != 0 ? 1 : 0;
      updated = true;
  }
```

Team Response

Acknowledged.

[L-05] `lastIntervalNumber` Is Off By One

Severity

Low Risk

Description

When the protocol starts, `startTimestamp` is saved. At time t (startTimestamp) to t+5, the interval should be zero and at time t+5 - t+10, the interval should be 1.

```
Supposed interval count
t - t+5, Interval = 0
t+5 - t+10, Interval = 1
```

The protocol sets it in such a way that in `t - t+5`, the interval is 1 and at `t+5 - t+10`, the interval is 2.

```
Protocol's interval count
t - t+5, Interval = 1
t+5 - t+10, Interval = 2
```

Recommendation

If this calculation is not intended, subtract `_lastIntervalNumber` by one if it's the first interval update.

```
function _calculateIntervals(uint256 timeElapsedSince)
    internal
    view
    returns (
        uint32 _lastIntervalNumber,
        uint128 _totalAmountForInterval,
        uint16 missedIntervals,
        uint256 beforeCurrDay
    )
{
    missedIntervals = _calculateMissedIntervals(timeElapsedSince);

    _lastIntervalNumber = lastIntervalNumber + missedIntervals + 1;
  if(lastBurnedIntervalStartTimestamp == 0) _lastIntervalNumber--;
```

Team Response

Acknowledged.

[L-06] 8% of Volt is Not Sent to Treasure Volt as Stated in the Documentation

Severity

Low Risk

Description

In the documentation, it states

> The "DragonX input" bnb buys VOLT (8% sent to "Treasure Volt, 92% used the next day to buy VYPER)

The whitepaper shows a figure of the Nexus where DragonX is used to buy Volt and 8% is sent to liquidity.

Be it the `DragonXVoltInput` contract or `DragonXVoltNexus` contract, both functions do not send 8% of VOLT to the treasury when `swapDragonXToVoltAndDistribute()` is called.

Impact

Documentation differs from code, 100% of VOLT is sent to buy VYPER instead.

Recommendation

It's recommended to copy how the `VyperDragonXNexus` contract does the transfer and replicate it in the `DragonXVoltNexus` contract.

```
dragonX.transfer(LIQUIDITY_BONDING_ADDR, wmul(dragonXAmount, uint256
    (0.08e18)));

    {
     uint256 toNexus = wmul(dragonXAmount, uint256(0.92e18));
        dragonX.approve(address(vyper.dragonXVoltNexus()), toNexus);
        vyper.dragonXVoltNexus().distributeDragonXForBurning(toNexus);
    }

    emit BuyAndBurn(currInterval.amountAllocated - incentive,
       dragonXAmount, msg.sender);
```

Team Response

Acknowledged.

[L-07] The `startTimestamp` in `DragonXVoltInput.getCurrentInterval()` Is Not Checked

Severity

Low Risk

Description

In the nexus contracts `getCurrentInterval()` function, if `startTimeStamp` is greater than `block.timestamp`, it means that the buy and burn system has not started yet. The view function should return 0 values then.

File: VyperDragonXNexus.sol

```
function getCurrentInterval()
    public
    view
    returns (
        uint32 _lastInterval,
        uint128 _amountAllocated,
        uint16 _missedIntervals,
        uint32 _lastIntervalStartTimestamp,
        bool updated
    )
{
if (startTimeStamp > Time.blockTs()) return (0, 0, 0, 0, false);
```

This check is not present in DragonXVoltInput.sol. Users might call `getCurrentInterval` before the start and get the wrong output.

Impact

If the contract is created but `startTimeStamp > block.timestamp`, then inaccurate values will be returned.

Recommendation

Append this check in `DragonXVoltInput.sol` as well.

Team Response

Acknowledged.

[L-08] Users Should Not be Able to Get Vyper Until Initial Liquidity is Minted in The Auction

Severity

Low Risk

Description

Users can deposit dragonX in `Auction.sol` and get back vyper (new token) in return. The auction contract intends to mint the liquidity position once enough dragonX is in the contract, and it will create a dragonX/vyper pool and a volt/vyper pool with $2500 on each side.

```
    _addLiquidityToPool(address(vyper), address(volt),
        INITIAL_VYPER_FOR_LP, voltAmount, _deadline, true);

     // Add liquidity to VYPER/DRAGONX pool
    _addLiquidityToPool(
        address(vyper), address(dragonX), INITIAL_VYPER_FOR_LP,
            INITIAL_DRAGONX_FOR_LP, _deadline, false
    );
```

The issue is that the auction mints 100M vyper everyday, and if there's not enough dragonX deposited on the first day to create the liquidity pool, users can get vyper tokens according to how much dragonX they put in.

Once users get vyper, they can manipulate the uniswapV3 position of dragonX/vyper or volt/vyper. When the protocol eventually creates the liquidity pool then, the price ratio may not be the same anymore.

Impact

UniswapV3 pool can be manipulated because of small liquidity. The `addInitialLiquidity()` function may keep reverting as a result of the manipulation.

Recommendation

Recommend not allowing the users to call `Auction.claim()` if the pool has not been set up yet to prevent users from getting vyper and minting positions in the uniswapV3 pool.

```
 function claim(uint32 _day) public {
+   require(lpPools.vyperVolt.hasLP && lpPools.vyperDragonX.hasLP, "
   Liquidity not added yet");
    uint32 daySinceStart = Time.dayGap(startTimestamp, Time.blockTs()) +
       1;
}
```

Team Response

Acknowledged.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!