CD SECURITY

AUDIT REPORT

Vyper Boost

December 2024

Prepared by
0xRuhum
dimulski
Pelz

Introduction

A time-boxed security review of the **Vyper Boost** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

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 problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Vyper Boost**

Vyper Boost allows users to deposit DragonX ERC20 tokens and in return, they receive Vyper ERC20 tokens as rewards. A big portion of the DragonX tokens are swapped for Vyper tokens via an UniswapV3 pool. The Vyper tokens received from the swap are burned. The Vyper Boost protocol is the revolutionary perpetual system that makes the Vyper ERC20 token hyper deflationary.

1. Users deposit DragonX ERC20 tokens to the Auction.sol contract, and in return receive rewards in Vyper.ERC20 tokens.
2. The deposited DragonX tokens are distributed to different addresses, one of which is the BuyAndBurn.sol contract.
3. The BuyAndBurn.sol contract allocates a certain amount of DragonX tokens to 5-minute intervals. In each interval, users can swap the DragonX tokens allocated for the latest interval/intervals for Vyper ERC20 tokens via an UniswapV3 Pool. As an incentive, they receive a percentage of the amount of DragonX tokens; the received Vyper tokens from the swap are burned.

Severity classification

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 of a successful attack

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

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 005554b17b2337932cd473603562ef4d4b99606a**

Scope

The following folders were in scope of the audit:

- `src/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 3 issues
- Low & Info: 4 issues

Findings Summary

ID	Title	Severity	Status
[M-01]	Tokens for one additional period are distributed when they shouldn't be	Medium	Acknowledged
[M-02]	`BuyAndBurn` will allocate more tokens than it should if the first interval update happens after more than day	Medium	Acknowledged
[M-03]	Different amounts of tokens will be distributed per interval, depending on how often the _intervalUpdate() function is invoked	Medium	Acknowledged
[L-01]	`startTimestamp` can be set to a block.timestamp in the past	Low	Acknowledged
[L-02]	Incorrect Distribution Amounts in `Constants.sol`	Low	Fixed
[L-03]	The use of `msg.sender == tx.origin` invariant may be broken in future	Low	Acknowledged
[I-01]	Unused Return Value Declarations in `Auction.sol` and `BuyAndBurn.sol`	Informational	Acknowledged

Detailed Findings

[M-01] Tokens for one additional period are distributed when they shouldn't be

Severity

Impact: Low

Likelihood: High

Description

The BuyAndBurn::distributeDragonXForBurning() function is permissionless, mainly expected to be called by the Auction::_distribute() function. According to the deployment file, the **startTimeStamp** of the BuyAndBurn.sol contract, will be exactly 1 day after the, **startTimestamp** of the Auction.sol contract. During this time DragonX tokens will be accumulated within the BuyAndBurn contract. However, the implementation of the `BuyAndBurn.sol` contract distributes funds for one more period than it should.

From the Constants.sol contract we see the following:

```
uint16 constant INTERVAL_TIME = 5 minutes;
uint16 constant INTERVALS_PER_DAY = uint16(24 hours / INTERVAL_TIME);
```

INTERVALS_PER_DAY = 288, so for every 24 hours there should be 288 interval distributions. Consider the following example:

- The BuyAndBurn.sol **startTimeStamp** is set to 2 days and 14 hours.
- The DragonX tokens balance of the BuyAndBurn.sol contract is 100_000e18
- At 2 days 14 hours + 12 sec the BuyAndBurn::distributeDragonXForBurning() function is called, which will internally call the BuyAndBurn::_intervalUpdate() function.
- As we can see from the BuyAndBurn::_calculateIntervals() function:

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

        _lastIntervalNumber = lastIntervalNumber + missedIntervals + 1;

        uint32 currentDay = Time.dayGap(startTimeStamp,
uint32(block.timestamp));

        uint32 dayOfLastInterval = lastBurnedIntervalStartTimestamp == 0
            ? currentDay
            : Time.dayGap(startTimeStamp,
lastBurnedIntervalStartTimestamp);

        if (currentDay == dayOfLastInterval) {
```

```
            uint256 dailyAllocation = wmul(totalDragonXDistributed,
getDailyDragonXAllocation());
            uint128 _amountPerInterval = uint128(dailyAllocation /
INTERVALS_PER_DAY);

            uint128 additionalAmount = _amountPerInterval *
missedIntervals;

            _totalAmountForInterval = _amountPerInterval +
additionalAmount;
        }
        ...
        //@note - If the last interval was only updated, but not burned
add its allocation to the next one.
        uint128 additional = prevInt.amountBurned == 0 ?
prevInt.amountAllocated : 0;

        if (_totalAmountForInterval + additional >
dragonX.balanceOf(address(this))) {
            _totalAmountForInterval =
uint128(dragonX.balanceOf(address(this)));
        } else {
            _totalAmountForInterval += additional;
        }
    }
```

- The **missedIntervals** will be 0, however, the _totalAmountForInterval will be equal to the distribution for one period.
- The **dailyAllocation** will be 15_000e18
- The **_amountPerInterval** will be ~52e18
- The **_totalAmountForInterval** will be ~52e18
- Later on in the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to the **startTimeStamp** which is 2 days 14 hours
- Now 12 more hours pass, the current **block.timestamp** is 3 days 2 hours and 12 seconds and the BuyAndBurn::distributeDragonXForBurning() function is called again.
- When the BuyAndBurn::_intervalUpdate() function is called, which in turn calls the BuyAndBurn::_calculateIntervals() function internally, we get the following calculations:
- The **missedIntervals** = 143
- The **_totalAmountForInterval** will be ~7_500e18 (this amount is for 144 intervals)
- In the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to 3 days and 2 hours
- 12 more hours pass and the current **block.timestamp** is 3 days 14 hours and 12 seconds, the current balance of DragonX tokens of the BuyAndBurn.sol contract is 200_000e18
- This time we will enter the else statement of the BuyAndBurn::_calculateIntervals() function:

```
    function _calculateIntervals(uint256 timeElapsedSince)
        internal
        view
        returns (
```

```
            uint32 _lastIntervalNumber,
            uint128 _totalAmountForInterval,
            uint16 missedIntervals,
            uint256 beforeCurrDay
        )
    {
        missedIntervals = _calculateMissedIntervals(timeElapsedSince);

        _lastIntervalNumber = lastIntervalNumber + missedIntervals + 1;

        uint32 currentDay = Time.dayGap(startTimeStamp,
uint32(block.timestamp));

        uint32 dayOfLastInterval = lastBurnedIntervalStartTimestamp == 0
            ? currentDay
            : Time.dayGap(startTimeStamp,
lastBurnedIntervalStartTimestamp);

         ...
          else {
            uint32 _lastBurnedIntervalStartTimestamp =
lastBurnedIntervalStartTimestamp;

            uint32 theEndOfTheDay =
Time.getDayEnd(_lastBurnedIntervalStartTimestamp);

            uint256 balanceOf = dragonX.balanceOf(address(this));

            while (currentDay >= dayOfLastInterval) {
                uint32 end = uint32(Time.blockTs() < theEndOfTheDay ?
Time.blockTs() : theEndOfTheDay - 1);

                uint32 accumulatedIntervalsForTheDay = (end -
_lastBurnedIntervalStartTimestamp) / INTERVAL_TIME;

                uint256 diff = balanceOf > _totalAmountForInterval ?
balanceOf - _totalAmountForInterval : 0;

                //@note - If the day we are looping over the same day as
the last interval's use the cached allocation, otherwise use the current
balance
                uint256 forAllocation = Time.dayGap(startTimeStamp,
lastBurnedIntervalStartTimestamp)
                    == dayOfLastInterval
                    ? totalDragonXDistributed
                    : balanceOf >= _totalAmountForInterval + wmul(diff,
getDailyDragonXAllocation()) ? diff : 0;

                uint256 dailyAllocation = wmul(forAllocation,
getDailyDragonXAllocation());

                ///@notice ->  minus INTERVAL_TIME minutes since, at the
end of the day the new epoch with new allocation
                _lastBurnedIntervalStartTimestamp = theEndOfTheDay -
```

```
INTERVAL_TIME;

                ///@notice ->  plus INTERVAL_TIME minutes to flip into the
next day
                theEndOfTheDay =
Time.getDayEnd(_lastBurnedIntervalStartTimestamp + INTERVAL_TIME);

                if (dayOfLastInterval == currentDay) beforeCurrDay =
_totalAmountForInterval;

                _totalAmountForInterval +=
                    uint128((dailyAllocation *
accumulatedIntervalsForTheDay) / INTERVALS_PER_DAY);

                dayOfLastInterval++;
            }
        }

        Interval memory prevInt = intervals[lastIntervalNumber];

        //@note - If the last interval was only updated, but not burned
add its allocation to the next one.
        uint128 additional = prevInt.amountBurned == 0 ?
prevInt.amountAllocated : 0;

        if (_totalAmountForInterval + additional >
dragonX.balanceOf(address(this))) {
            _totalAmountForInterval =
uint128(dragonX.balanceOf(address(this)));
        } else {
            _totalAmountForInterval += additional;
        }
    }
```

- The **missedIntervals** will be 143
- The **theEndOfTheDay** will be 3 days and 14 hours
- The **end** will be 3 days and 14 hours - 1 sec
- The **accumulatedIntervalsForTheDay** will be 143
- The **dailyAllocation** will be 15_000e18
- The **_lastBurnedIntervalStartTimestamp** will be 3 days & 14 hours - 5 min
- The **theEndOfTheDay** will be 4 days & 14 hours
- The **_totalAmountForInterval** will be ~7_448e18
- When we enter the next while loop iteration we get the following:
- The **end** will be equal to the current **block.timestamp** which is 3 days & 14 hours & 12 seconds
- The **accumulatedIntervalsForTheDay** will be 1
- The **diff** will be ~192_552e18
- The **dailyAllocation** will be ~28_882e18
- The **_totalAmountForInterval** will be ~7_448e18 + ~100e18 = ~7_548e18
- In the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to 3 days and 14 hours

As can be seen from the above example, a total of 15_100e18 tokens will be distributed(made available for burning) for a period of 1 day, when it should have been only 15_000e18 which is equal to the 288 Intervals that should occur during 24 hours.

Recommendations

Consider adding a functionality that adds 1 interval(5 mins) to the **lastBurnedIntervalStartTimestamp** the first time the _intervalUpdate() function is invoked.

[M-02] BuyAndBurn will allocate more tokens than it should if the first interval update happens after more than day

Severity

Impact: High

Likelihood: Low

Description

If the first interval update happens more than a day after the start time, the elapsed time will count as the same day no matter how long it is.

Meaning, if the update happens 48 hours after the start, it'll allocate 15% of the initial DragonX deposit twice, so 30% in total.

Here's a PoC:

```
    function test_allocates_more_than_it_should() public {
        // This test shows how the `_calculateIntervals` function sets the
`_lastBurnedIntervalStartTimestamp` to the end of the day
        // even if the current timestamp is in the middle of the day

        // setup
        address user = makeAddr("user");
        MockERC20 dragonX = new MockERC20("DragonX", "DRGNX");
        SwapActionParams memory params = SwapActionParams(address(0),
address(0), address(this));
        // start time to is beginning of the current day
        uint32 startTime = uint32(1735135200) / 1 days * 1 days;
        VyperBoostBuyAndBurn bnb = new VyperBoostBuyAndBurn(startTime,
address(dragonX), address(0), params);

        deal(address(dragonX), user, 100e18);
        vm.startPrank(user);
        dragonX.approve(address(bnb), type(uint256).max);
        bnb.distributeDragonXForBurning(100e18);
        vm.stopPrank();
```

```
        // we warp 48 hours into the future
        // so we'd expect the total allocated amount to be:
        // first day:
        // 100e18 * 15% = 15e18
        // second day:
        // 85e18 * 15% = 12.75e18
        // total allocation = 15e18 + 12.75e18 = 27.75e18

        vm.warp(startTime + 48 hours);
        deal(address(dragonX), user, 1);
        vm.prank(user);
        bnb.distributeDragonXForBurning(1);

        (uint128 allocated,) = bnb.intervals(bnb.lastIntervalNumber());

        // will fail
        // 30052083333333333141 != 27750000000000000000
        assertEq(allocated, 27.75e18);
    }
```

The issue is that `_calculateIntervals()` will calculate `dayOfLastInterval` as `currentDay` if `lastBurnedIntervalTimestamp == 0`.

```
    function _calculateIntervals(uint256 timeElapsedSince)
        internal
        view
        returns (
            uint32 _lastIntervalNumber,
            uint128 _totalAmountForInterval,
            uint16 missedIntervals,
            uint256 beforeCurrDay
        )
    {
        // ...
        uint32 currentDay = Time.dayGap(startTimeStamp,
uint32(block.timestamp));

        uint32 dayOfLastInterval = lastBurnedIntervalStartTimestamp == 0
            ? currentDay
            : Time.dayGap(startTimeStamp,
lastBurnedIntervalStartTimestamp);

        // ...
```

Recommendations

If `lastBurnedIntervalTimestamp == 0`, `dayOfLastInterval` should be `Time.dayGap(startTimeStamp, startTimestamp)`.

[M-03] Different amounts of tokens will be distributed per interval, depending on how often the _intervalUpdate() function is invoked

Severity

Impact: Medium

Likelihood: Medium

Description

The BuyAndBurn::distributeDragonXForBurning() function is permissionless, mainly expected to be called by the Auction::_distribute() function. However, depending on how often the _intervalUpdate() function is invoked, the **dailyAllocation** of DragonX token for each new day will be different.

According to the deployment file, the **startTimeStamp** of the BuyAndBurn.sol contract, will be exactly 1 day after the, **startTimestamp** of the Auction.sol contract.

Consider the following example:

- The BuyAndBurn.sol **startTimeStamp** is set to 2 days and 14 hours.
- The DragonX tokens balance of the BuyAndBurn.sol contract is 100_000e18
- At 2 days 14 hours + 12 sec the BuyAndBurn::distributeDragonXForBurning() function is called, which will internally call the BuyAndBurn::_intervalUpdate() function.
- We will get the following calculations from the BuyAndBurn::_calculateIntervals() function:
    1. The **missedIntervals** will be 0, however, the _totalAmountForInterval will be equal to the distribution for one period.
    2. The **dailyAllocation** will be 15_000e18
    3. The **_amountPerInterval** will be ~52e18
    4. The **_totalAmountForInterval** will be ~52e18
    5. Later on in the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to the **startTimeStamp** which is 2 days 14 hours
- Now 12 more hours pass, the current **block.timestamp** is 3 days 2 hours and 12 seconds and the BuyAndBurn::distributeDragonXForBurning() function is called again.
- When the BuyAndBurn::_intervalUpdate() function is called, which in turn calls the BuyAndBurn::_calculateIntervals() function internally, we get the following calculations:
    1. The **missedIntervals** = 143
    2. The **_totalAmountForInterval** will be ~7_500e18 (this amount is for 144 intervals)
    3. In the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to 3 days and 2 hours
- 12 more hours pass and the current **block.timestamp** is 3 days 14 hours and 12 seconds, the current balance of DragonX tokens of the BuyAndBurn.sol contract is 200_000e18
- This time we will enter the else statement of the BuyAndBurn::_calculateIntervals() function:
    1. The **missedIntervals** will be 143
    2. The **theEndOfTheDay** will be 3 days and 14 hours

3. The **end** will be 3 days and 14 hours - 1 sec
4. The **accumulatedIntervalsForTheDay** will be 143
5. The **dailyAllocation** will be 15_000e18
6. The **_lastBurnedIntervalStartTimestamp** will be 3 days & 14 hours - 5 min
7. The **theEndOfTheDay** will be 4 days & 14 hours
8. The **_totalAmountForInterval** will be ~7_448e18
9. When we enter the next while loop iteration we get the following:
10. The **end** will be equal to the current **block.timestamp** which is 3 days & 14 hours & 12 seconds
11. The **accumulatedIntervalsForTheDay** will be 1
12. The **diff** will be ~192_552e18
13. The **dailyAllocation** will be ~28_882e18
14. The **_totalAmountForInterval** will be ~7_448e18 + ~100e18 = ~7_548e18
15. In the _updateSnapshot() function, the **totalDragonXDistributed** will be set to ~192_552e18
16. In the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to 3 days and 14 hours.

Now let's consider a different scenario

- The BuyAndBurn.sol **startTimeStamp** is set to 2 days and 14 hours.
- The DragonX tokens balance of the BuyAndBurn.sol contract is 100_000e18
- At 2 days 14 hours + 287 * 5 mins + 12 sec the BuyAndBurn::distributeDragonXForBurning() function is called, which will internally call the BuyAndBurn::_intervalUpdate() function.
- We will get the following calculations from the BuyAndBurn::_calculateIntervals() function:
  1. The **missedIntervals** will be 287
  2. The **dailyAllocation** will be 15_000e18
  3. The **_amountPerInterval** will be ~52e18
  4. The **_totalAmountForInterval** will be ~15_000e18
  5. Later on in the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to the **startTimeStamp** which is 2 days 14 hours + 287 * 5 mins which will be equal to 3 days 14 hours - 5 min
- 5 more minutes pass and the current **block.timestamp** is 3 days 14 hours and 12 seconds, the current balance of DragonX tokens of the BuyAndBurn.sol contract is 200_000e18
- This time we will enter the else statement of the BuyAndBurn::_calculateIntervals() function:
  1. The **missedIntervals** will be 0
  2. The **theEndOfTheDay** will be 3 days and 14 hours
  3. The **end** will be 3 days and 14 hours - 1 sec
  4. The **accumulatedIntervalsForTheDay** will be 0
  5. The **_lastBurnedIntervalStartTimestamp** will be 3 days & 14 hours - 5 min
  6. The **theEndOfTheDay** will be 4 days & 14 hours
  7. The **_totalAmountForInterval** will be 0
  8. When we enter the next while loop iteration we get the following:
  9. The **end** will be equal to the current **block.timestamp** which is 3 days & 14 hours & 12 seconds
  10. The **accumulatedIntervalsForTheDay** will be 1
  11. The **diff** will be 200_000e18
  12. The **dailyAllocation** will be 30_000e18

13. The **_totalAmountForInterval** will be 0 + ~104e18 = ~104e18
14. In the _updateSnapshot() function, the **totalDragonXDistributed** will be set to 200_000e18
15. In the BuyAndBurn::_intervalUpdate() function, the **lastBurnedIntervalStartTimestamp** will be set to 3 days and 14 hours.

As we can see from the above examples in both of the cases the protocol distributes almost equal amounts for the same periods ~15_100e18. However in the first case, the **totalDragonXDistributed** will be ~192_552e18, and in the second 200_000e18. This is a significant difference because when the calculations for the next interval are performed the **dailyAllocation** in the first example will be 192_552e18 * 0.15e18 / 1e18 = 28_882e18, and in the second example the **dailyAllocation** will be 30_000e18 Based on the fact that the sole purpose of the `BuyAndBurn.sol` contract is to determine how much tokens should be distributed for burning, and thus incentives for the users calling the swapDragonXToVyperAndBurn() function, I believe the above described discrepancies warrant a medium impact.

Recommendations

Consider implementing an iterative system in the _calculateIntervals() function that guarantees each interval is calculated separately. Or implement an offchain oracle that guarantees the BuyAndBurn::_intervalUpdate() function is invoked in each period.

[L-01] `startTimestamp` can be set to a block.timestamp in the past

Both the `Auction.sol` and `BuyAndBurn.sol` contracts set a **startTimestamp** in their constructors. However, there are no checks to see whether the timestamp is in the future.

Recommendation

Consider checking whether the startTimestamp is in the future.

[L-02] Incorrect Distribution Amounts in `Constants.sol`

The distribution amounts defined in `Constants.sol` do not align with the specifications outlined in the project documentation. This discrepancy would lead to incorrect token allocations.

Code Snippet

```
uint64 constant DX_BURN = 0.05e18; // 5% @audit-issue should be 4%
according to docs
uint64 constant TO_LP = 0.1e18; // 10% @audit-issue should be 8% according
to docs
uint64 constant TO_VOLT_BURN = 0.05e18; // 5% @audit-issue should be 4%
according to docs
uint64 constant TO_BNB = 0.72e18; // 72% @audit-issue should be 76%
according to docs
```

> **Note:**
> Distribution of DragonX input as per documentation:
>
> - 7% Genesis
> - 1% Dev
> - 4% Volt burn
> - 4% Burnt
> - 8% Liquidity wallet
> - 76% Buy and burn

Identified Deviations

1. `DX_BURN` is set to 5% but should be 4%.
2. `TO_LP` is set to 10% but should be 8%.
3. `TO_VOLT_BURN` is set to 5% but should be 4%.
4. `TO_BNB` is set to 72% but should be 76%.

Recommendations

Update the constants in `Constants.sol` to match the values specified in the documentation:

```
uint64 constant DX_BURN = 0.04e18; // 4%
uint64 constant TO_LP = 0.08e18; // 8%
uint64 constant TO_VOLT_BURN = 0.04e18; // 4%
uint64 constant TO_BNB = 0.76e18; // 76%
```

[L-03] The use of msg.sender == tx.origin invariant may be broken in future

The `swapDragonXToVyperAndBurn` function in the **BuyAndBurn.sol** contract enforce a restriction using `msg.sender == tx.origin` to prevent interactions from other smart contracts:

```
require(msg.sender == tx.origin, OnlyEOA());
```

This check is currently valid because **EOAs** (Externally Owned Accounts) cannot contain contract code. However, with the introduction of **EIP-7702**, EOAs may hold contract code, enabling contracts to mimic EOAs (`msg.sender == tx.origin`).

The proposed change in Ethereum would:

1. Allow contracts to batch transactions on behalf of an EOA.
2. Break the assumption that `msg.sender == tx.origin` guarantees the transaction is initiated by an EOA.

You can read more about the eip here

Recommendation

Remove the check

[I-01] Unused Return Value Declarations in `Auction.sol` and `BuyAndBurn.sol`

The functions `amountToClaim` in `Auction.sol` and `getDailyDragonXAllocation` in `BuyAndBurn.sol` declare named return values, but these return values are never used within the function bodies.

Identified Functions

1. **`Auction.sol`**

```
function amountToClaim(address _user, uint32 _day) public view returns
(uint256 toClaim) { // @audit-issue Unused return value declaration
    // Function implementation...
}
```

2. **`BuyAndBurn.sol`**

```
function getDailyDragonXAllocation() public pure returns (uint256
dailyWadAllocation) { // @audit-issue Unused return value declaration
    return 0.15e18;
}
```

In both cases, the named return values (`toClaim` and `dailyWadAllocation`) are not referenced or utilized in the function body.

Recommendations

Update the function definitions to remove the unused named return values or implement them.