Volt Protocol
Audit Report

September 23, 2024

Conducted by:

Bogo Cvetkov (b0g0), Independent Security Researcher

Table of Contents

1. About b0g0

Bogo Cvetkov (b0g0) is a smart contract security researcher with a proven track record of consistently uncovering vulnerabilities in a wide spectrum of DeFi protocols. Constantly pushing the limits of his expertise, he strives to be a superior security partner to any protocol & client he dedicates himself to!

2. About Volt

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. Volt enters deflation quickly with a massive buy and burn.

3. Risk Classification

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

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality
- **Low** - funds are not at risk

3.2. Likelihood

- **High** - almost certain to happen, easy to perform, or highly incentivized
- **Medium** - only conditionally possible, but still relatively likely
- **Low** - requires specific state or little-to-no incentive

3.3. Handling severity levels

- **Critical** - Must fix as soon as possible (if already deployed)
- **High** - Must fix (before deployment if not already deployed)
- **Medium** - Should fix
- **Low** - Could fix
- **Governance** - Could fix

4. Executive Summary

For the duration of 5 days **b0g0** has invested his expertise as a security researcher to analyze the smart contracts of **Volt** protocol and assess the state of its security. For that time a total of 13 issues have been detected, out of which **1** has been assigned a severity level of **High** and **3** a severity level of **Medium**.

5. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This effort is limited by time, resources, and expertise. My evaluation of the codebase aims to uncover as many vulnerabilities as possible, given the above limitations! Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended!

Overview

Project	Volt
URL	https://docs.volt.win/
Platform	Ethereum
Language	Solidity
Repo	https://github.com/Kuker-Labs/volt-contracts
Commit Hash	6d0e6c235371afaa33fbb62e88fe04d94bd46037
Mitigation	2206210e2c465a500b2f13ea32d20bf459c72fd8
Dates	18 September - 22 September 2024

Scope

Contract	Address
TheVolt.sol	-
Volt.sol	-
VoltAuction.sol	-
VoltBuyAndBurn.sol	-
OracleLibrary	-

Issue Statistic

Severity	Count
High	1
Medium	3
Low/Informational	9
Total	**13**

6. Findings

6.1. High Severity

6.1.1. TitanX depositors receive less volt tokens than they should

Context: VoltAuction.sol

Description:

The VoltAuction contract receives deposits from titanX holders. Based on the deposited amount relative to the total deposited for that day, the depositors can claim volt tokens as rewards:

```
function claim(uint32 _day) public {
       ....
        uint256 toClaim = amountToClaim(msg.sender, _day);

        if (toClaim == 0) revert VoltAuction__NothingToClaim();
       ...

        volt.safeTransfer(msg.sender, toClaim);
        ...
    }
```

The problem here is how the toClaim is calculated inside amountToClaim():

```
function amountToClaim(
        address _user,
        uint32 _day
    ) public view returns (uint256 toClaim) {
        uint256 depositAmount = depositOf[_user][_day];
        DailyStatistic memory stats = dailyStats[_day];

        //@audit - this rounds down once
        uint256 voltPerTitanX = wdiv(stats.voltEmitted, stats.titanXDeposited);

        //@audit - and then the round down is multiplied by the token amount
        return wmul(depositAmount, voltPerTitanX);
    }
```

First the total emitted volt tokens are distributed among the total deposited tokens for that day to calculate how much volt tokens should be claimed for each deposited titanX. However the wdiv() function rounds down. This is normal if it happens once, however the already calculated voltPerTitanX amount (rounded down) is multiplied by the number of deposited tokens, which respectively multiplies the effect of rounding down.

Example:

- depositAmount - 900_000_00e18 (10%)(~**70$**)
- titanXDeposited - 900_000_000e18
- voltEmitted - 100_000_000e18
- voltPerTitanX -111111111111111111

Due to rounding down being multiplied by each token the total amount lost will be **1e7**

Recommendation:

Use a more simple formula, which is the de facto standard when calculating share of something. It will round down only once (1 wei) and prevent the above scenario from happening:

```
uint256 toClaim = (depositAmount * voltEmitted) / titanXDeposited;
```

Resolution:

Fixed

6.2. Medium Severity

6.2.1. Vault and titanX tokens can get locked in the VoltAuction contract

Context: VoltAuction.sol

Description :

The addLiquidityToVoltTitanxPool() function is used to provide the initial liquidity of volt & titanX tokens (INITIAL_VOLT_FOR_LP & INITIAL_TITAN_X_FOR_LIQ) to the newly created uniswap pool. The default slippage used for the provision is 20%. Problem is that after the position is minted, no check is made to see if all of the tokens have been deposited. As result any tokens (up to 20%) that have not been deposited into the pool will just remain stuck in the contract.

Recommendation

Use the amount0 & amount1 parameters returned from the INonfungiblePositionManager().mint() call and subtract them from the initial amounts. If any amounts are left, transfer them to the appropriate addresses.

Resolution:

Acknowledged - the team will closely monitor pool and manually add the required liquidity to keep the ratios in check and prevent slippage

6.2.2. Possibility of Volt tokens not getting minted in full

Context: VoltAuction.sol

Description :

The _updateAuction() function of the VoltAuction contract is used to calculate and distribute the initial emissions of volt tokens like so:

```
function _updateAuction() internal {
        uint32 daySinceStart = Time.daysSince(startTimestamp) + 1;

        if (dailyStats[daySinceStart].voltEmitted != 0) return;

        uint256 emitted = daySinceStart <= 10
            ? volt.emitForAuction()
            : theVolt.emitForAuction();

        dailyStats[daySinceStart].voltEmitted = uint128(emitted);
    }
```

It calculates the days passed since start and for the first 10 days it mints 100M volt tokens from the Volt contract. The function is triggered on each deposit() call.

If for some reason in the first 10 days there is a day without a single deposit call(e.g 100M emission is being skipped) there is no mechanism to compensate for it.

The deposit() function is public and anyone can call it. Assuming there would be an automated node tasked with calling the function every day, there is still a chance the problem will occur. For example the transaction can be sent properly during the 24 hour cycle, but due to high gas prices (especially relevant on mainnet) it can get stuck in the mem-pool and execute later (a couple of hours), going into the next day and skipping the current one. Another less probable, but still possible scenario is block stuffing (malicious scenario)

Recommendation

One possible approach might be to add an admin-restricted function, that can be called for the days between 1 & 10 and emit any skipped emissions for the days that have passed. This will act as a safety mechanism to make sure the target emission can always get distributed at 100% in case something goes wrong.

Resolution:

Acknowledged - the team will closely monitor the contract and make sure distributions will happen on a daily basis

6.2.3. Deposits should not be allowed in case of an empty treasure volt

Context: VoltAuction.sol

Description:

After the 10th day has passed, the Auction contract distributes tokens from theVolt (treasure volt). The logic for this is handled in the _updateAuction() function:

```
function _updateAuction() internal {
        uint32 daySinceStart = Time.daysSince(startTimestamp) + 1;

        if (dailyStats[daySinceStart].voltEmitted != 0) return;

        uint256 emitted = daySinceStart <= 10
            ? volt.emitForAuction()
            : theVolt.emitForAuction();

        dailyStats[daySinceStart].voltEmitted = uint128(emitted);
    }
```

In case theVolt is empty the emitted amount would be 0. However the deposit() &_updateAuction() functions will not revert in such scenarios, allowing deposits of titanX tokens - which won't get anything to claim since the emission for that day would be 0.

Some cases where this might occur are the following:

- VoltBuyAndBurn.swapTitanXForVoltAndBurn() is not called in the first 10 days after _auctionStartTime
- In the first 10 days the required INITIAL_TITAN_X_FOR_LIQ has not been accumulated, hence no titanX was send to theVault

The possibility of the above to occur is low, however the effect is more serious because the depositor will not get any volt tokens.

Recommendation:

Add validation inside the deposit function that will make sure the deposit will revert in case the emitted amount is zero - e.g there would be nothing to claim. It is a simple but important validation that will safeguard against such edge cases.

Resolution:

Fixed

6.3. Low Severity

6.3.1. Interval calculation can be distorted if the startTimestamp is not divisible by the interval time

Context: VoltBuyAndBurn.sol

Description:

If the startTimestamp of the VoltBuyAndBurn contract is set to a value that is not divisible by the interval time it might distort the calculations of passed intervals. For example the following logic in _calculateIntervals():

```
uint32 accumulatedIntervalsForTheDay = (end -
                   _lastBurnedIntervalStartTimestamp) / INTERVAL_TIME;
```

might cut off one interval (they are normally 96 per day) due to rounding down since _lastBurnedIntervalStartTimestamp will not be evenly divisible by interval if startTimestamp isn't. The main reason is that the amount of accumulated intervals is always added on top of the initial startTimestamp (which is un-even from the start):

```
if (updated) {
            lastBurnedIntervalStartTimestamp =
                _lastIntervalStartTimestamp +
                (uint32(_missedIntervals) * INTERVAL_TIME);
           ....
        }
```

The main effect of this is that it will affect accuracy when calculating the intervals that have passed.

Recommendation:

Consider adding the following validation in the constructor to make sure, the proper startTimestamp is set:

```
if ((_startTimestamp % INTERVAL_TIME) != 0)
                    revert Invalid_Timestamp();
```

Resolution:

Fixed

6.3.2. If intervals are not updated in the first day after start, distribution would be suboptimal

Context: VoltBuyAndBurn.sol

Description:

The _calculateIntervals() function uses the following logic:

```
uint32 dayOfLastInterval = lastBurnedIntervalStartTimestamp == 0
            ? currentDay
            : Time.dayCountByT(lastBurnedIntervalStartTimestamp);

        if (currentDay == dayOfLastInterval) {
                uint256 dailyAllocation = wmul(
                totalTitanXDistributed,
                getDailyTitanXAllocation(Time.blockTs())
            );

            uint128 _amountPerInterval = uint128(
                dailyAllocation / INTERVALS_PER_DAY
            );
         }
```

The first time _calculateIntervals() is run lastBurnedIntervalStartTimestamp is always 0, so dayOfLastInterval will be the current day, which means the first logic block is always executed on the first run after start.

And if we have a scenario where there has been a deposit in the first day(before start when intervals are not updated), 2 days pass and _calculateIntervals() is run for the first time, the getDailyTitanXAllocation(Time.blockTs()) call will calculate the distribution for the second day, although this is the deposit from the first day and distribute it at a lower percent.

Recommendation:

The scenario is less likely to happen, since probably there would be deposits on the start day which would activate interval updates.

Recommendation here is to make sure _calculateIntervals() is called on the same day after start of buyAndBurn, after that the interval update logic will kick in and handle distribution for previous days.

Resolution:

Acknowledged

6.3.3. Additional _amountPerInterval is added on the start of distribution

Context: VoltBuyAndBurn.sol

Description:

The _calculateIntervals() function uses the following logic:

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

            uint128 additionalAmount = _amountPerInterval * missedIntervals;

            //@audit-info - _amountPerInterval is added one more time
            _totalAmountForInterval = _amountPerInterval + additionalAmount;
```

The reason _amountPerInterval is added to additional amount is because after the first interval update the intervals are decremented by one and the above logic compensates for it

```
 _missedIntervals = uint16(timeElapsedSince / INTERVAL_TIME);

if (lastBurnedIntervalStartTimestamp != 0) _missedIntervals--;
```

However the intervals are not decremented on the first run when lastBurnedIntervalStartTimestamp is 0, so there is no need to add an additional interval there

Recommendation:

The following logic can be updated

```
_totalAmountForInterval = additionalAmount;

if (lastBurnedIntervalStartTimestamp != 0) {
      _totalAmountForInterval += _amountPerInterval;
}
```

Resolution:

Not fixed

6.3.4. Accumulated intervals for the day might be inflated

Context: VoltBuyAndBurn.sol

Description:

The _calculateIntervals() function uses the following logic when more than 1 day since last update has passed:

```
 while (currentDay >= dayOfLastInterval) {
....
  uint32 accumulatedIntervalsForTheDay = (end -
                    _lastBurnedIntervalStartTimestamp) / INTERVAL_TIME;
....

 ///@notice ->  minus 15 minutes since, at the end of the day the new epoch with new
allocation
  _lastBurnedIntervalStartTimestamp =
  theEndOfTheDay - INTERVAL_TIME; //@audit - this adds extra interval

  ///@notice ->  plus 15 minutes to flip into the next day
  theEndOfTheDay = getDayEnd(
      _lastBurnedIntervalStartTimestamp + INTERVAL_TIME
  );
```

```
}
```

The getDayEnd() function calculates the end of the day for each passed day since the last update and uses it to calculate the passed intervals. The thing is that on each day the _lastBurnedIntervalStartTimestamp is decreased with INTERVAL_TIME - as a result the final calculated accumulatedIntervalsForTheDay is 24 hrs + INTERVAL_TIME.

Here is an example:

- 4 days have passed since last update
- on day 1 since last update:
  - _lastBurnedIntervalStartTimestamp - day 1
  - theEndOfTheDay - day 2 since update
  - accumulatedIntervalsForTheDay - INTERVALS_PER_DAY
- on day 2 since last update:
  - _lastBurnedIntervalStartTimestamp - (day 2 - INTERVAL_TIME)
  - theEndOfTheDay - day 3 since update
  - accumulatedIntervalsForTheDay - (INTERVALS_PER_DAY + INTERVAL_TIME)
- on day 3 since last update:
  - _lastBurnedIntervalStartTimestamp -(day 3 - INTERVAL_TIME)
  - theEndOfTheDay - day 4 since update
  - accumulatedIntervalsForTheDay - (INTERVALS_PER_DAY + INTERVAL_TIME)

As you can see for each next day an additional interval is added. This affects a couple of assumptions made during interval calculation:

- Code assumes INTERVALS_PER_DAY have passed, while in fact INTERVALS_PER_DAY + INTERVAL_TIME have passed for some days

```
uint128 _amountPerInterval = uint128(
                    dailyAllocation / INTERVALS_PER_DAY
                );
```

- As result the the amount per interval is accumulated for one additional interval

```
_totalAmountForInterval +=

                    _amountPerInterval *

                    accumulatedIntervalsForTheDay;
```

- The alreadyAllocated variable is incremented with dailyAllocation, although the actual allocated amount in _totalAmountForInterval was more than that

```
alreadyAllocated += dayOfLastInterval == currentDay
                   ? _amountPerInterval * accumulatedIntervalsForTheDay
                   : dailyAllocation;
```

As you can see the logic of some calculations and assumptions made in the code can be affected, leading to unexpected outcomes or distorted distributions.

Recommendation:

Most of the time discrepancies in calculated distribution amounts are handled in the final code blocks of the function, which make sure the allocations do not exceed the available balances:

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

Still it is preferred that the distribution logic works as expected. Consider if the reduction of _lastBurnedIntervalStartTimestamp by an INTERVAL_TIME is necessary and if remove it in order to fit in the expected intervals for a day

An additional recommendation is to update the alreadyAllocated variable to be always increased by the actual amount of accumulatedIntervalsForTheDay instead of assigning dailyAllocation, which might not always hold true

Resolution:

Not fixed

6.3.5. Current day allocation might be reduced during interval update

Context: VoltBuyAndBurn.sol

Description:

The _calculateIntervals() function calculates and returns a beforeCurrDay value. It is only used when more than 1 day since the last update has passed to reduce appropriately the available allocation amounts for the current day.

Based on the logic of the function if currentDay == dayOfLastInterval, nothing is deducted from the available balances. However if multiple days have passed, the following logic is used:

```
alreadyAllocated += dayOfLastInterval == currentDay
                    ? _amountPerInterval * accumulatedIntervalsForTheDay
                    : dailyAllocation;

                //@audit - reduces the allocation for the day
                if (dayOfLastInterval == currentDay)
                    beforeCurrDay = alreadyAllocated;
```

In the second case the allocated amounts for the current day are also reduced from the available amounts for that day (totalTitanXDistributed) . Here is a short example on the difference:

- 1st case - currentDay == dayOfLastInterval

        - 100 tokens available for distribution - 100/INTERVALS_PER_DAY - each interval for that day gets an equal cut
    - 2nd case - currentDay != dayOfLastInterval
        - If we assume 3 days have passed + 50% from the current day and we have 400 tokens to allocate - 300 are allocated for the first 3 days and 50 for the first half of the current day. totalTitanXDistributed is reduced by 350 and the other intervals from the current day would have dailyAllocation of 50/INTERVALS_PER_DAY, when it should be 100/INTERVALS_PER_DAY.

Recommendation:

Consider not including the allocations from the current day in the beforeCurrDay variable

Resolution:

Fixed

6.4. Governance

6.4.1. Governance Privileges

Context: VoltBuyAndBurn.sol

Description:

The contract owner has control over several variables that can impact the outcome of a transaction:

- slippageAdmin
- swapCap
- slippage, twapLookback

Recommendation:

Consider incorporating a Gnosis multi-signature contract as the owner and ensuring that the Gnosis participants are trusted entities

Resolution:

Acknowledged

6.5. Informational

6.5.1. Insufficient validation

Context: VoltAuction.sol, VoltBuyAndBurn.sol

Description:

All issues related to validation are collected here to keep the report focused and easy to read:

- Inside VoltAuction.claim() check that _day is not 0 and update == to >= daySinceStart to also make sure future dates are not accepted
- Inside VoltAuction.collectFees() add the following check

```
if (!lp.hasLP) revert VoltAuction__LiquidityNotAdded();
```

- Inside the constructor of Volt contract check that the owner, titanX & _dragonX are not address(0). It's a simple guard that would prevent potential mistakes during deployment
- Inside VoltBuyAndBurn.getCurrentInterval() consider checking if burning has not started yet and just return 0. It is a separate public function that can be called directly and it will revert due to an underflow.

Recommendation:

Consider implementing the above mentioned recommendations

Resolution:

Partially Fixed

6.5.2. Gas optimizations

Context: VoltAuction.sol, VoltBuyAndBurn.sol

Description:

All issues related to gas are collected here to keep the report focused and easy to read:

- Inside VoltAuction._distribute() move the titanX.balanceOf(address(this)) call inside the if block. It is not being used after pool liquidity is added so it will save gas to skip it for subsequent calls.
- Inside VoltBuyAndBurn._calculateIntervals() you can check if dailyAllocation is 0 and skip the next couple of lines of calculations and assignments in order to save some gas on further iterations

Recommendation:

Consider implementing the above mentioned recommendations

Resolution:

Not Fixed

6.5.3. Emit events on important state updates

Context: VoltBuyAndBurn.sol

Description:

It is a good practice to emit events when important state variables are being updated in a contract. Events should be added to the following functions:

- changeSlippageAdmin()
- setSwapCap()
- changeTitanXToVoltSlippage()

Recommendation:

Consider implementing the above mentioned recommendations

Resolution:

Not Fixed