Vader Protocol contest Findings & Analysis Report

2022-05-03

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of the Vader Protocol smart contract system written in Solidity. The code contest took place between December 21—December 25 2021.

Wardens

21 Wardens contributed reports to the Vader Protocol contest:

1. TomFrenchBlockchain
2. leastwood
3. danb
4. cmichel
5. hyh
6. certora
7. gzeon
8. robee
9. defsec
10. pauliax
11. Dravee
12. GiveMeTestEther
13. 0x1f8b
14. cccz
15. Critical
16. Jujic
17. p4st13r4 (0x69e8 and 0xb4bb4)
18. AmitN

19. jayjonah8
20. MetaOxNull

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 37 unique vulnerabilities and 80 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 14 received a risk rating in the category of HIGH severity, 6 received a risk rating in the category of MEDIUM severity, and 17 received a risk rating in the category of LOW severity.

C4 analysis also identified 14 non-critical recommendations and 29 gas optimizations.

Scope

The code under review can be found within the C4 Vader Protocol contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 1542 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (14)

[H-01] `VaderPoolV2` minting synths & fungibles can be frontrun

Submitted by cmichel, also found by cccz, Critical, danb, leastwood, and TomFrenchBlockchain

The `VaderPoolV2` `mintFungible` and `mintSynth` functions perform an unsafe `nativeAsset.safeTransferFrom(from, address(this), nativeDeposit)` with a parameter-specified `from` address.

Note that these functions are not called by the Router, they are directly called on the pool. Therefore, users will usually be required to send two transactions, a first one approving the pool, and then a second one for the actual `mintSynth`.

An attacker can frontrun the `mintSynth(IERC20 foreignAsset, uint256 nativeDeposit, address from, address to)` function, use the same `from=victim` parameter but change the `to` parameter to the attacker.

Impact

It's possible to frontrun victims stealing their native token deposits and receiving synths / fungible tokens.

Recommended Mitigation Steps

Remove the `from` parameter and always perform the `safeTransferFrom` call with `from=msg.sender`.

SamSteinGG (Vader) acknowledged

[H-02] `VaderPoolV2` owner can steal all user assets which are approved `VaderPoolV2`

Submitted by TomFrenchBlockchain

Possible theft of all user assets with an ERC20 approval on VaderPoolV2.

Proof of Concept

The owner of `VaderPoolV2` can call the `setTokenSupport` function which allows the caller to supply any address from which to take the assets to provide the initial liquidity, the owner can also specify who shall receive the resulting LP NFT and so can take ownership over these assets. This call will succeed for any address which has an ERC20 approval on `VaderPoolV2` for USDV and `foreignAsset`.

https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/dex-v2/pool/VaderPoolV2.sol#L442-L474

This in effect gives custody over all assets in user wallets which are approved on `VaderPoolV2` to Vader Protocol governance. This is especially problematic in the case of Vader Protocol as there's a single entity (i.e. the Council) which can force through a proposal to steal these assets for themselves with only the timelock giving protection to users, for this reason I give this high severity.

Recommended Mitigation Steps

Enforce that the initial liquidity is provided by the VaderPoolV2 owner.

[H-03] Oracle doesn't calculate USDV/VADER price correctly

Submitted by TomFrenchBlockchain, also found by danb and leastwood

Invalid values returned from oracle for USDV and VADER prices in situations where the oracle uses more than one foreign asset.

Proof of Concept

The USDV price is calculated as so (for simplicity we'll consider a two pairs):

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/lbt/LiquidityBasedTWAP.sol#L393-L409

```
totalUSD =  (PriceForeign0InUSD * liquidityWeights[0] + PriceFor
```

`totalUSD` is then the average price of the foreign assets paired against USDV in terms of USD, weighted by the TVL of the relevant liquidity pool

```
totalUSDV =
  (pairData0
      .nativeTokenPriceAverage
      .mul(pairData0.foreignUnit)
      .decode144() * liquidityWeights[0] +
   pairData1
      .nativeTokenPriceAverage
      .mul(pairData1.foreignUnit)
      .decode144() * liquidityWeights[1]) /
  totalUSDVLiquidityWeight;

// in pseudocode for readability
totalUSDV = (USDVPriceInForeign0 * liquidityWeights[0] + USDVPri
```

`totalUSDV` is then the average price of USDV in terms of each of the foreign assets, weighted by the TVL of the relevant liquidity pool.

It should be fairly clear that this is the incorrect calculation as all the terms in `totalUSDV` are in different units - you can't average the price of USDV in ETH with the price of USDV in BTC and get a meaningful result.

It appears that the VADER team intended to calculate the price of USDV in terms of USD through a number of different paired assets and then average them at the end based on the liquidity in each pair but have started averaging too early.

High severity issue as the oracle is crucial for determining the exchange rate between VADER and USDV to be used for IL protection and minting/burning of

USDV - an incorrect value will result in the protocol losing significant funds.

Recommended Mitigation Steps

Review the algorithm used for calculating the prices of assets and ensure that it's calculating what you expect.

SamSteinGG (Vader) acknowledged

[H-04] Vader TWAP averages wrong

Submitted by cmichel

The vader price in `LiquidityBasedTWAP.getVaderPrice` is computed using the `pastLiquidityWeights` and `pastTotalLiquidityWeight` return values of the `syncVaderPrice`.

The `syncVaderPrice` function does not initialize all weights and the total liquidity weight does not equal the sum of the individual weights because it skips initializing the pair with the previous data if the TWAP update window has not been reached yet:

```
    function syncVaderPrice()
        public
        override
        returns (
            uint256[] memory pastLiquidityWeights,
            uint256 pastTotalLiquidityWeight
        )
    {
        uint256 _totalLiquidityWeight;
        uint256 totalPairs = vaderPairs.length;
        pastLiquidityWeights = new uint256[](totalPairs);
        pastTotalLiquidityWeight = totalLiquidityWeight[uint256(Path

        for (uint256 i; i < totalPairs; ++i) {
            IUniswapV2Pair pair = vaderPairs[i];
            ExchangePair storage pairData = twapData[address(pair)];
            // @audit-info lastMeasurement is set in _updateVaderPri
            uint256 timeElapsed = block.timestamp - pairData.lastMea
```

```
            // @audit-info update period depends on pair
            // @audit-issue if update period not reached => does not
            if (timeElapsed < pairData.updatePeriod) continue;

            uint256 pastLiquidityEvaluation = pairData.pastLiquidity
            uint256 currentLiquidityEvaluation = _updateVaderPrice(
                pair,
                pairData,
                timeElapsed
            );

            pastLiquidityWeights[i] = pastLiquidityEvaluation;

            pairData.pastLiquidityEvaluation = currentLiquidityEvalu

            _totalLiquidityWeight += currentLiquidityEvaluation;
        }

        totalLiquidityWeight[uint256(Paths.VADER)] = _totalLiquidity
    }
```

POC

This bug leads to several different issues. A big one is that an attacker can break the price functions and make them revert. Observe what happens if an attacker calls `syncVaderPrice` twice in the same block:

- The first time any pairs that need to be updated are updated
- On the second call `_totalLiquidityWeight` is initialized to zero and all pairs have already been updated and thus skipped. `_totalLiquidityWeight` never increases and the storage variable `totalLiquidityWeight[uint256(Paths.VADER)] = _totalLiquidityWeight = 0;` is set to zero.
- DoS because calls to `getStaleVaderPrice` / `getVaderPrice` will revert in `_calculateVaderPrice` which divides by `totalLiquidityWeight = 0`.

Attacker keeps double-calling `syncVaderPrice` every time an update window of one of the pairs becomes eligible to be updated.

Impact

This bug leads to using wrong averaging and ignoring entire pairs due to their weights being initialized to zero and never being changed if the update window is not met. This in turn makes it easier to manipulate the price as potentially only a single pair needs to be price-manipulated.

It's also possible to always set the `totalLiquidityWeight` to zero by calling `syncVaderPrice` twice which in turn reverts all transactions making use of the price because of a division by zero in `_caluclateVaderPrice`. An attacker can break the `USDV.mint` minting forever and any router calls to `VaderReserve.reimburseImpermanentLoss` also fail as they perform a call to the reverting price function.

Recommended Mitigation Steps

Even if `timeElapsed < pairData.updatePeriod`, the old pair weight should still contribute to the total liquidity weight and be set in `pastLiquidityWeights`. Move the `_totalLiquidityWeight += currentLiquidityEvaluation` and the `pastLiquidityWeights[i] = pastLiquidityEvaluation` assignments before the `continue`.

SamSteinGG (Vader) confirmed

[H-05] Oracle returns an improperly scaled USDV/VADER price

Submitted by TomFrenchBlockchain

Invalid values returned from oracle in vast majority of situations.

Proof of Concept

The LBT oracle does not properly scale values when calculating prices for VADER or USDV. To show this we consider the simplest case where we expect USDV to return a value of $1 and show that the oracle does not return this value.

Consider the case of the LBT oracle tracking a single USDV-DAI pair where USDV trades 1:1 for DAI and Chainlink reports that DAI is exactly $1. We then work through

the lines linked below:

https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/lbt/LiquidityBasedTWAP.sol#L393-L409

For L397 we get a value of 1e8 as Chainlink reports the price of DAI with 8 decimals of accuracy.

```
foreignPrice = getChainlinkPrice(address(foreignAsset));
foreignPrice = 1e8
```

We can set `liquidityWeights[i]` and `totalUSDVLiquidityWeight` both to 1 as we only consider a single pair so L399-401 becomes

```
totalUSD = foreignPrice;
totalUSD = 1e8;
```

L403-408 is slightly more complex but from looking at the links below we can calculate `totalUSDV` as shown https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/dex-v2/pool/VaderPoolV2.sol#L81-L90 https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/external/libraries/FixedPoint.sol#L137-L160

```
totalUSDV = pairData
    .nativeTokenPriceAverage
    .mul(pairData.foreignUnit)
    .decode144()
// pairData.nativeTokenPriceAverage == 2**112
// pairData.foreignUnit = 10**18
// decode144(x) = x >> 112
totalUSDV = (2**112).mul(10**18).decode144()
totalUSDV = 10**18
```

Using `totalUSD` and `totalUSDV` we can then calculate the return value of `_calculateUSDVPrice`

```
    returnValue = (totalUSD * 1 ether) / totalUSDV;

    returnValue = 1e8 * 1e18 / 1e18

    returnValue = 1e8
```

For the oracle implementation to be correct we then expect that the Vader protocol to treat values of 1e8 from the oracle to mean USDV is worth $1. However from the lines of code linked below we can safely assume that it is intended to be that values of 1e18 represent$1 rather than 1e8.

https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/tokens/USDV.sol#L76 https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/tokens/USDV.sol#L109

High severity issue as the oracle is crucial for determining the exchange rate between VADER and USDV to be used for IL protection and minting/burning of USDV - an incorrect value will result in the protocol losing significant funds.

Recommended Mitigation Steps

Go over oracle calculation again to ensure that various scale factors are properly accounted for. Some handling of the difference in the number of decimals between the chainlink oracle and the foreign asset should be added.

Build a test suite to ensure that the oracle returns the expected values for simple situations.

SamSteinGG (Vader) confirmed

[H-06] LPs of `VaderPoolV2` can manipulate pool reserves to extract funds from the reserve.

Submitted by TomFrenchBlockchain, also found by hyh

Impermanent loss protection can be exploited to drain the reserve.

Proof of Concept

In `VaderPoolV2.burn` we calculate the current losses that the LP has made to impermanent loss.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex-v2/pool/VaderPoolV2.sol#L265-L296

These losses are then refunded to the LP in VADER tokens from the reserve.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex-v2/router/VaderRouterV2.sol#L220

This loss is calculated by the current reserves of the pool so if an LP can manipulate the pool's reserves they can artificially engineer a huge amount of IL in order to qualify for a payout up to the size of their LP position.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex/math/VaderMath.sol#L72-L92

The attack is then as follows.

1. Be an LP for a reasonable period of time (IL protection scales linearly up to 100% after a year)
2. Flashloan a huge amount of one of the pool's assets.
3. Trade against the pool with the flashloaned funds to unbalance it such that your LP position has huge IL.

4. Remove your liquidity and receive compensation from the reserve for the IL you have engineered.
5. Re-add your liquidity back to the pool.
6. Trade against the pool to bring it back into balance.

The attacker now holds the majority of their flashloaned funds (minus slippage/swap fees) along with a large fraction of the value of their LP position in VADER paid out from the reserve. The value of their LP position is unchanged. Given a large enough LP position, the IL protection funds extracted from the reserve will exceed the funds lost to swap fees and the attacker will be able to repay their flashloan with a profit.

This is a high risk issue as after a year any large LP is incentivised and able to perform this attack and drain reserve funds.

Recommended Mitigation Steps

Use a manipulation resistant oracle for the relative prices of the pool's assets (TWAP, etc.)

[H-07] Redemption value of synths can be manipulated to drain `VaderPoolV2` of all native assets in the associated pair

Submitted by TomFrenchBlockchain, also found by certora

Draining of funds from `VaderPoolV2`.

Proof of Concept

See the `VaderPool.mintSynth` function: https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex-v2/pool/VaderPoolV2.sol#L153-L194

As the pool's reserves can be manipulated through flashloans similar to on UniswapV2 (the slip mechanism can be mitigated by splitting the manipulation over a number of trades), an attacker may set the exchange rate between `nativeAsset` and synths (calculated from the reserves). An attacker can exploit this to drain funds from the pool.

1. The attacker first flashloans and sells a huge amount of `foreignAsset` to the pool. The pool now thinks `nativeAsset` is extremely valuable.
2. The attacker now uses a relatively small amount of `nativeAsset` to mint synths using `VaderPool.mintSynth`. As the pool thinks `nativeAsset` is very valuable the attacker will receive a huge amount of synths.
3. The attacker can now manipulate the pool in the opposite direction by buying up the `foreignAsset` they sold to the pool. `nativeAsset` is now back at its normal price, or perhaps artificially low if the attacker wishes.
4. The attacker now burns all of their synths. As `nativeAsset` is considered much less valuable than at the point the synths were minted it takes a lot more of `nativeAsset` in order to pay out for the burned synths.

For the price of a flashloan and some swap fees, the attacker has now managed to extract a large amount of `nativeAsset` from the pool. This process can be repeated as long as it is profitable.

Recommended Mitigation Steps

Tie the exchange rate use for minting/burning synths to a manipulation resistant oracle.

SamSteinGG (Vader) acknowledged

[H-08] Reserve does not properly apply prices of VADER and USDV tokens

Submitted by TomFrenchBlockchain

Reserve pays out vastly higher (or lower) IL protection than it should.

Proof of Concept

Consider the lines 98 and 102 as shown on the link below:

https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/reserve/VaderReserve.sol#L95-L103

Here we multiply the IL experienced by the LP by a price for USDV or VADER as returned by the LBT. However the price from the oracle is a fixed point number (scaled up by 1e8 or 1e18 depending on the resolution of finding “Oracle returns an improperly scaled USDV/VADER price”) and so a fixed scaling factor should be applied to convert back from a fixed point number to a standard integer.

As it stands depending on the branch which is executed, the amount to be reimbursed will be 1e18 times too large or too small. Should the “else” branch be executed the reserve will pay out much in terms of IL protection resulting in severe loss of funds. High severity.

Recommended Mitigation Steps

Apply similar logic to as displayed here:

https://github.com/code-423n4/2021-12-vader/blob/00ed84015d4116da2f9db0c68db6742c89e73f65/contracts/tokens/USDV.sol#L109

[H-09] `USDV.sol` Mint and Burn Amounts Are Incorrect

Submitted by leastwood, also found by TomFrenchBlockchain

The `USDV.mint` function queries the price of `Vader` from the `LiquidityBasedTwap` contract. The calculation to determine `uAmount` in `mint` is actually performed incorrectly. `uAmount = (vPrice * vAmount) / 1e18;` will return the `USD` amount for the provided `Vader` as `vPrice` is denominated in `USD/Vader`. This `uAmount` is subsequently used when minting tokens for the user (locked for a period of time) and fee to the contract owner.

This same issue also applies to how `vAmount = (uPrice * uAmount) / 1e18;` is calculated in `USDV.burn`.

This is a severe issue, as the `mint` and `burn` functions will always use an incorrect amount of tokens, leading to certain loss by either the protocol (if the user profits) or the user (if the user does not profit).

Proof of Concept

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/tokens/USDV.sol#L66-L98

```
    function mint(uint256 vAmount)
        external
        onlyWhenNotLocked
        returns (uint256 uAmount)
    {
        uint256 vPrice = lbt.getVaderPrice();

        vader.transferFrom(msg.sender, address(this), vAmount);
        vader.burn(vAmount);

        uAmount = (vPrice * vAmount) / 1e18;

        if (cycleTimestamp <= block.timestamp) {
            cycleTimestamp = block.timestamp + 24 hours;
            cycleMints = uAmount;
        } else {
            cycleMints += uAmount;
            require(
                cycleMints <= dailyLimit,
                "USDV::mint: 24 Hour Limit Reached"
            );
        }

        if (exchangeFee != 0) {
            uint256 fee = (uAmount * exchangeFee) / _MAX_BASIS_POINT
            uAmount = uAmount - fee;
            _mint(owner(), fee);
        }

        _mint(address(this), uAmount);

        _createLock(LockTypes.USDV, uAmount);
    }
```

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/tokens/USDV.sol#L100-L120

```
function burn(uint256 uAmount)
    external
    onlyWhenNotLocked
    returns (uint256 vAmount)
{
    uint256 uPrice = lbt.getUSDVPrice();

    _burn(msg.sender, uAmount);

    vAmount = (uPrice * uAmount) / 1e18;

    if (exchangeFee != 0) {
        uint256 fee = (vAmount * exchangeFee) / _MAX_BASIS_POINT
        vAmount = vAmount - fee;
        vader.mint(owner(), fee);
    }

    vader.mint(address(this), vAmount);

    _createLock(LockTypes.VADER, vAmount);
}
```

Recommended Mitigation Steps

Consider utilising both `getVaderPrice` and `getUSDVPrice` when calculating the expected `uAmount` and `vAmount` to mint or burn. To calculate `uAmount` in `mint`, `vPrice` should be denominated in `USDV/Vader`. To calculate `vAmount` in `burn`, `uPrice` should be denominated in `Vader/USDV`. It would be useful to add unit tests to test this explicitly as it is expected that users will interact with the `USDV.sol` contract frequently.

Oxstormtrooper (Vader) disputed and commented:

> Mint / burn calculation with USD is intentional, modeled after LUNA / UST.

> Mint USDV
> 1 USD worth of Vader should mint 1 USDV

> Burn USDV
> 1 USDV should mint 1 USD worth of Vader

> https://docs.terra.money/Concepts/Protocol.html#expansion-and-contraction

[H-10] `previousPrices` Is Never Updated Upon Syncing Token Price

Submitted by leastwood

The `LiquidityBasedTWAP` contract attempts to accurately track the price of `VADER` and `USDV` while still being resistant to flash loan manipulation and short-term volatility. The `previousPrices` array is meant to track the last queried price for the two available paths, namely `VADER` and `USDV`.

The `setupVader` function configures the `VADER` token by setting `previousPrices` and adding a token pair. However, `syncVaderPrice` does not update `previousPrices` after syncing, causing `currentLiquidityEvaluation` to be dependent on the initial price for `VADER`. As a result, liquidity weightings do not accurately reflect the current and most up to date price for `VADER`.

This same issue also affects how `USDV` calculates `currentLiquidityEvaluation`.

This issue is of high risk and heavily impacts the accuracy of the TWAP implementation as the set price for `VADER/USDV` diverges from current market prices. For example, as the Chainlink oracle price and initial price for `VADER` diverge, `currentLiquidityEvaluation` will begin to favour either on-chain or off-chain price data depending on which price result is greater. The following calculation for `currentLiquidityEvaluation` outlines this behaviour.

```
    currentLiquidityEvaluation =
        (reserveNative * previousPrices[uint256(Paths.VADER)]) +
        (reserveForeign * getChainlinkPrice(pairData.foreignAsset));
```

Proof of Concept

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/lbt/LiquidityBasedTWAP.sol#L150-L189

```
function _updateVaderPrice(
    IUniswapV2Pair pair,
    ExchangePair storage pairData,
    uint256 timeElapsed
) internal returns (uint256 currentLiquidityEvaluation) {
    bool isFirst = pair.token0() == vader;

    (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();

    (uint256 reserveNative, uint256 reserveForeign) = isFirst
        ? (reserve0, reserve1)
        : (reserve1, reserve0);

    (
        uint256 price0Cumulative,
        uint256 price1Cumulative,
        uint256 currentMeasurement
    ) = UniswapV2OracleLibrary.currentCumulativePrices(address(p

    uint256 nativeTokenPriceCumulative = isFirst
        ? price0Cumulative
        : price1Cumulative;

    unchecked {
        pairData.nativeTokenPriceAverage = FixedPoint.uq112x112(
            uint224(
                (nativeTokenPriceCumulative -
                    pairData.nativeTokenPriceCumulative) / timeE
            )
        );
    }

    pairData.nativeTokenPriceCumulative = nativeTokenPriceCumula

    pairData.lastMeasurement = currentMeasurement;

    currentLiquidityEvaluation =
        (reserveNative * previousPrices[uint256(Paths.VADER)]) +
        (reserveForeign * getChainlinkPrice(pairData.foreignAsse
}
```

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/lbt/LiquidityBasedTWAP.sol#L221-L235

```
    function setupVader(
        IUniswapV2Pair pair,
        IAggregatorV3 oracle,
        uint256 updatePeriod,
        uint256 vaderPrice
    ) external onlyOwner {
        require(
            previousPrices[uint256(Paths.VADER)] == 0,
            "LBTWAP::setupVader: Already Initialized"
        );

        previousPrices[uint256(Paths.VADER)] = vaderPrice;

        _addVaderPair(pair, oracle, updatePeriod);
    }
```

Recommended Mitigation Steps

Consider updating `previousPrices[uint256(Paths.VADER)]` and `previousPrices[uint256(Paths.USDV)]` after syncing the respective prices for the two tokens. This will ensure the most up to date price is used when evaluating liquidity for all available token pairs.

SamSteinGG (Vader) acknowledged

[H-11] `totalLiquidityWeight` Is Updated When Adding New Token Pairs Which Skews Price Data For `getVaderPrice` and `getUSDVPrice`

Submitted by leastwood

The `_addVaderPair` function is called by the `onlyOwner` role. The relevant data in the `twapData` mapping is set by querying the respective liquidity pool and Chainlink oracle. `totalLiquidityWeight` for the `VADER` path is also incremented by the `pairLiquidityEvaluation` amount (calculated within `_addVaderPair`). If a user then calls `syncVaderPrice`, the recently updated `totalLiquidityWeight` will be taken into consideration when iterating through all token pairs eligible for price

updates to calculate the liquidity weight for each token pair. This data is stored in `pastTotalLiquidityWeight` and `pastLiquidityWeights` respectively.

As a result, newly added token pairs will increase `pastTotalLiquidityWeight` while leaving `pastLiquidityWeights` underrepresented. This only occurs if `syncVaderPrice` is called before the update period for the new token has not been passed.

This issue also affects how the price for `USDV` is synced.

Proof of Concept

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/lbt/LiquidityBasedTWAP.sol#L299

```
    function _addVaderPair(
        IUniswapV2Pair pair,
        IAggregatorV3 oracle,
        uint256 updatePeriod
    ) internal {
        require(
            updatePeriod != 0,
            "LBTWAP::addVaderPair: Incorrect Update Period"
        );

        require(oracle.decimals() == 8, "LBTWAP::addVaderPair: Non-U

        ExchangePair storage pairData = twapData[address(pair)];

        bool isFirst = pair.token0() == vader;

        (address nativeAsset, address foreignAsset) = isFirst
            ? (pair.token0(), pair.token1())
            : (pair.token1(), pair.token0());

        oracles[foreignAsset] = oracle;

        require(nativeAsset == vader, "LBTWAP::addVaderPair: Unsuppo

        pairData.foreignAsset = foreignAsset;
        pairData.foreignUnit = uint96(
            10**uint256(IERC20Metadata(foreignAsset).decimals())
```

```
        );

        pairData.updatePeriod = updatePeriod;
        pairData.lastMeasurement = block.timestamp;

        pairData.nativeTokenPriceCumulative = isFirst
            ? pair.price0CumulativeLast()
            : pair.price1CumulativeLast();

        (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();

        (uint256 reserveNative, uint256 reserveForeign) = isFirst
            ? (reserve0, reserve1)
            : (reserve1, reserve0);

        uint256 pairLiquidityEvaluation = (reserveNative *
            previousPrices[uint256(Paths.VADER)]) +
            (reserveForeign * getChainlinkPrice(foreignAsset));

        pairData.pastLiquidityEvaluation = pairLiquidityEvaluation;

        totalLiquidityWeight[uint256(Paths.VADER)] += pairLiquidityE

        vaderPairs.push(pair);

        if (maxUpdateWindow < updatePeriod) maxUpdateWindow = update
    }
```

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/lbt/LiquidityBasedTWAP.sol#L113-L148

```
    function syncVaderPrice()
        public
        override
        returns (
            uint256[] memory pastLiquidityWeights,
            uint256 pastTotalLiquidityWeight
        )
    {
        uint256 _totalLiquidityWeight;
        uint256 totalPairs = vaderPairs.length;
        pastLiquidityWeights = new uint256[](totalPairs);
        pastTotalLiquidityWeight = totalLiquidityWeight[uint256(Path
```

```
        for (uint256 i; i < totalPairs; ++i) {
            IUniswapV2Pair pair = vaderPairs[i];
            ExchangePair storage pairData = twapData[address(pair)];
            uint256 timeElapsed = block.timestamp - pairData.lastMea

            if (timeElapsed < pairData.updatePeriod) continue;

            uint256 pastLiquidityEvaluation = pairData.pastLiquidity
            uint256 currentLiquidityEvaluation = _updateVaderPrice(
                pair,
                pairData,
                timeElapsed
            );

            pastLiquidityWeights[i] = pastLiquidityEvaluation;

            pairData.pastLiquidityEvaluation = currentLiquidityEvalu

            _totalLiquidityWeight += currentLiquidityEvaluation;
        }

        totalLiquidityWeight[uint256(Paths.VADER)] = _totalLiquidity
    }
```

As shown above, `pastTotalLiquidityWeight = totalLiquidityWeight[uint256(Paths.VADER)]` loads in the total liquidity weight which is updated when `_addVaderPair` is called. However, `pastLiquidityWeights` is calculated by iterating through each token pair that is eligible to be updated.

Recommended Mitigation Steps

Consider removing the line `totalLiquidityWeight[uint256(Paths.VADER)] += pairLiquidityEvaluation;` in `_addVaderPair` so that newly added tokens do not impact upcoming queries for `VADER/USDV` price data. This should ensure `syncVaderPrice` and `syncUSDVPrice` cannot be manipulated when adding new tokens.

SamSteinGG (Vader) confirmed

[H-12] Using single total native reserve variable for synth and non-synth reserves of `VaderPoolV2` can lead to losses for synth holders

Submitted by hyh, also found by certora

Users that mint synths do provide native assets, increasing native reserve pool, but do not get any liquidity shares issued. In the same time, an exit of non-synth liquidity provider yields releasing a proportion of all current reserves to him.

Whenever an exit of non-synth LP is substantial enough, the system will have much less native asset regarding the cumulative deposit of synth holders. That is, when a LP entered he provided a share of current reserves, both native and foreign, and got the corresponding liquidity shares in return. Suppose then big enough amounts of synths were minted, providing correspondingly big enough amount of native assets. If the LP now wants to exit, he will obtain a part of total native assets, including a part of the amount that was provided by synth minter. If the exit is big enough there will be substantially less native assets left to reimburse the synth minter than he initially provided. This is not reversible: the synth minters lost their native assets to LP that exited.

Proof of Concept

There are three types of mint/burn: NFT, fungible and synths. First two get LP shares, the latter gets synths. Whenever NFT or fungible LP exits, it gets a proportion of combined reserves. That is, some of native reserves were deposited by synth minters, but it is not accounted anyhow, only one total reserve number per asset is used.

Suppose the following scenario, Alice wants to provide liquidity, while Bob wants to mint synths:

1. Alice deposits both sides to a pool, 100 USDV and 100 foreign.
2. Bob deposit 100 USDV and mints some foreign Synth.
3. LP withdraws 95% of her liquidity shares. As she owns the pool liquidity, she gets 95% of USDV and foreign total reserves, 190 USDV and 95 foreign. Alice received almost all of her and Bob's USDV.

4. If Bob burn his synth and withdraw, he will get a tiny fraction of USDV he deposited (calculated by VaderMath.calculateSwap, we use its terms):

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex/math/VaderMath.sol#L98 x = 100, X = 0.05 * 200 = 10, Y = 0.05 * 100 = 5. Swap outcome, how much USDV will Bob get, is x * Y * X / (x + X) ^ 2 = 100 * 5 * 10 / (110^2) = 0.4 (rounded).

The issue is that synth provided and LP provided USDV aren't accounted separately, total reserves number if used everywhere instead:

Synth minters provide native asset, say USDV, to the system: https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L187

Synth minters get synths and no LP shares, while to account for their deposit, the total USDV balance is increased: https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L187

When LP enters, it gets liquidity shares proportionally to current reserves (NFT mint, notice the reserveNative, which is BasePoolV2's pair.reserveNative, total amount of native asset in the Pool): https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/BasePoolV2.sol#L497

When LP exits, it gets a proportion of current reserves back (NFT burn): https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/BasePoolV2.sol#L223

The same happens when fungible LP mints (same reserveNative): https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L336 And burns: https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L401

Recommended Mitigation Steps

Account for LP provided liquidity separately from total amount variables, i.e. use only LP provided native reserves variables in LP shares mint and burn calculations. That should suffice as total amount of native assets is still to be used elsewhere, whenever

the whole pool is concerned, for example, in rescue function, swap calculations and so forth.

SamSteinGG (Vader) acknowledged

[H-13] Council veto protection does not work

Submitted by TomFrenchBlockchain

Council can veto proposals to remove them to remain in power.

Proof of Concept

The Vader governance contract has the concept of a “council” which can unilaterally accept or reject a proposal. To prevent a malicious council preventing itself from being replaced by the token holders, the veto function checks the calldata for any proposal action directed at `GovernorAlpha` to see if it matches the `changeCouncil` function selector.

Note this is done by reading from the `proposal.calldatas` array.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/governance/GovernorAlpha.sol#L568-L603

If we look at the structure of a proposal however we can see that the function selector is held (in the form of the signature) in the `signatures` array rather than being included in the calldata. The `calldata` array then holds just the function arguments for the call rather than specifying which function to call.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/governance/GovernorAlpha.sol#L71-L72

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/governance/GovernorAlpha.sol#L356-L362

Indeed if we look at the `TimeLock` contract we see that the signature is hashed to calculate the function selector and is prepended onto the calldata.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/governance/Timelock.sol#L292-L299

Looking at the function signature of the `changeCouncil` we can see that the value that the `veto` function will check against `this.changeCouncil.signature` will be the first 4 bytes of an abi encoded address and so will always be zero no matter what function is being called.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/governance/GovernorAlpha.sol#L623

High risk as this issue gives the council absolute control over the DAO such that they cannot be removed.

Recommended Mitigation Steps

Hash the function signatures to calculate function selectors and then check those rather than the calldata.

This is something that should be picked up by a test suite however, I'd recommend writing tests to ensure that protections you add to the code have any affect and more broadly check that the code behaves as expected.

SamSteinGG (Vader) acknowledged

[H-14] Denial of service

Submitted by danb

https://github.com/code-423n4/2021-12-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L334 on the first deposit, the total liquidity is set to `nativeDeposit`. this might be a very low number compared to `foreignDeposit`. It can cause a denial of service of the pair.

Impact

A pair can enter a denial of service state.

Proof of Concept

consider the following scenario: the owner of the pool calls `setFungibleTokenSupport` for a new token, for example weth. a malicious actor calls `mintFungible`, with `nativeDeposit == 1` and `foreignDeposit == 10 ether`. `totalLiquidityUnits` will be 1. the pool can be arbitraged, even by the attacker, but `totalLiquidityUnits` will still be 1. this means that 1 liquidity token is equal to all of the pool reserves, which is a lot of money. It will cause a very high rounding error for anyone trying to mint liquidity. then, anyone who will try to mint liquidity will either:

1. fail, because they can't mint 0 liquidity if their amount is too small.
2. get less liquidity tokens than they should, because there is a very high rounding error, and its against new liquidity providers.

The rounding error losses will increase the pool reserves, which will increase value of liquidity tokens, so the hacker can even profit from this.

after this is realised, no one will want to provide liquidity, and since the pair cannot be removed or replaced, it will cause denial of service for that token forever.

SamSteinGG (Vader) acknowledged

Medium Risk Findings (6)

[M-01] `VaderPoolV2.mintFungible` exposes users to unlimited slippage

Submitted by TomFrenchBlockchain, also found by pauliax and robee

Frontrunners can extract up to 100% of the value provided by LPs to VaderPoolV2 as fungible liquidity.

Proof of Concept

Users can provide liquidity to `VaderPoolV2` through the `mintFungible` function.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex-v2/pool/VaderPoolV2.sol#L311-L317

This allows users to provide tokens in any ratio and the pool will calculate what fraction of the value in the pool this makes up and mint the corresponding amount of liquidity units as an ERC20.

However there's no way for users to specify the minimum number of liquidity units they will accept. As the number of liquidity units minted is calculated from the current reserves, this allows frontrunners to manipulate the pool's reserves in such a way that the LP receives fewer liquidity units than they should. e.g. LP provides a lot of `nativeAsset` but very little `foreignAsset`, the frontrunner can then sell a lot of `nativeAsset` to the pool to devalue it.

Once this is done the attacker returns the pool's reserves back to normal and pockets a fraction of the value which the LP meant to provide as liquidity.

Recommended Mitigation Steps

Add a user-specified minimum amount of LP tokens to mint.

SamSteinGG (Vader) acknowledged

Jack the Pug (judge) decreased severity to medium and commented:

> I'm downgrading this [from `high`] to `med` and merging all the issues related to slippage control into this one.

[M-02] Adding pair of the same `foreignAsset` would replace oracle of earlier entry

Submitted by gzeon

Oracles are mapped to the `foreignAsset` but not to the specific pair. Pairs with the same `foreignAsset` (e.g. UniswapV2 and Sushi) will be forced to use the same oracle. Generally this should be the expected behavior but there are also possibility that while adding a new pair changed the oracle of an older pair unexpectedly.

Proof of Concept

https://github.com/code-423n4/2021-12-vader/blob/9fb7f206eaff1863aeeb8f997e0f21ea74e78b49/contracts/lbt/LiquidityBasedTWAP.sol#L271

```
            oracles[foreignAsset] = oracle;
```

Recommended Mitigation Steps

Bind the oracle to pair instead

SamSteinGG (Vader) confirmed

[M-03] No way to remove `GasThrottle` from `VaderPool` after deployment

Submitted by TomFrenchBlockchain

Potential DOS on swaps on `VaderPool`.

Proof of Concept

BasePool makes use of a `validateGas` modifier on swaps which checks that the user's gas price is below the value returned by `_FAST_GAS_ORACLE`.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex/pool/BasePool.sol#L292

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/dex/utils/

GasThrottle.sol#L8-L22

Should `_FAST_GAS_ORACLE` be compromised to always return zero then all swaps will fail. There is no way to recover from this scenario.

Recommended Mitigation Steps

Either remove GasThrottle.sol entirely or allow governance to turn it off as is done in VaderPoolV2.sol

[M-04] `VaderReserve.reimburseImpermanentLoss` improperly converts USDV to VADER

Submitted by TomFrenchBlockchain

IL isn't properly converted from being in terms of USDV to VADER, resulting in reserve paying out incorrect amount.

Proof of Concept

`VaderReserve.reimburseImpermanentLoss` receives an `amount` in terms of USDV and converts this to an amount of VADER to send to `recipient`.

However as shown in the link if there is a previous price stored for USDV, the amount of VADER tokens to be sent to the `recipient` is `amount / usdvPrice`.

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/reserve/VaderReserve.sol#L84-L110

`usdvPrice` is the total USD value of foreign assets divided by the total amount of USDV in a number of pairs. It's then some measure of the inverse of the price of USDV in USD, nothing to do with converting into VADER.

The reserve will then improperly calculate the amount of VADER to pay out once there is a single reading of the USDV price.

Recommended Mitigation Steps

It looks like both branches of this if statement are supposed to be run, i.e. convert from USDV to USD and then to VADER but I can't be sure. Care should be taken so that the calculation being performed is the expected one.

SamSteinGG (Vader) acknowledged

[M-05] Users can lock themselves out of being able to convert VETH, becoming stuck with the deprecated asset

Submitted by TomFrenchBlockchain

I've put this as a medium issue as we're leaking value as users are stuck with assets which are likely to be worth much less as they are deprecated. It could also be low as it's not exploitable by outside parties and the loss isn't taken by the protocol but the user.

Impact

Potential for users to lose the right to convert VETH to VADER, being stuck with a deprecated token.

Proof of Concept

Should a user have a zero allowance of VETH on the converter, no VETH will be taken and no VADER will be paid out as L147 will set the amount to zero.

https://github.com/code-423n4/2021-12-vader/blob/28d3405447f0c3353964ca755a42562840d151c5/contracts/tokens/converter/Converter.sol#L145-L150

There is a `minVader` check on L153 to enforce a minimum output of VADER but should this be improperly set the transaction would succeed with the user receiving much less VADER than they expect.

Crucially, the merkle proof that was used in this transaction will be marked as invalid so the user will not be able to try again once they have set the proper allowance.

Someone can then lose the opportunity to convert their VETH and are left with a worthless deprecated token if they are inattentive.

It seems like this is trying to handle the case where a user doesn't have the full amount of VETH as they are entitled to convert (by setting their allowance equal to their balance?). This is a pretty suboptimal way to go about this as it's extremely implicit so users are liable to make mistakes.

I'd recommend decoupling the merkle proof from conversion of VETH to VADER:

1. Change the merkle proof to set an `amountEligibleToConvert` value in storage for each user (which would be initially set to `amount`).
2. Allow a user to then convert VETH to VADER up to their `amountEligibleToConvert` value, subtracting the amount converted from this each time.

For gas efficiency we can use a sentinel value here to mark a user which has claimed their full quota already distinctly from someone who hasn't provided a merkle proof yet (to avoid having to track this separately in another storage slot)

These two steps could be chained in a single transaction to give a similar UX as currently but would also allow users to recover in the case of partial conversions.

Recommended Mitigation Steps

As above. I'd caution against just stating "The frontend will handle this correctly so this isn't an issue", there will be users who interact with the contract manually so it's important to make the interface safe where possible.

Oxstormtrooper (Vader) acknowledged

[M-06] Oracle can be manipulted to consider only a single pair for pricing

Submitted by TomFrenchBlockchain

Loss of resilience of oracle to a faulty pricing for a single pair.

Proof of Concept

In the oracle we calculate the TVL of each pool by pulling the reserves and multiplying both assets by the result of a supposedly manipulation resistant oracle (the oracle queries its previous value for USDV and pulls the foreign asset from chainlink).

https://github.com/code-423n4/2021-12-vader/blob/fd2787013608438beae361ce1bb6d9ffba466c45/contracts/lbt/LiquidityBasedTWAP.sol#L353-L383

This value can be manipulated by skewing the reserves of the underlying pair with a flashloan attack. An attacker can then make a pool appear with an arbitrarily large `currentLiquidityEvaluation` which will result in all other pairs contributing negligibly to the final result of the oracle.

This doesn't result in loss of funds by itself afaict but should there be an issue for the chainlink price feed for the asset in any pool then an attacker can force the oracle to only use that pool for pricing USDV/VADER

Medium risk as "Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements." External requirements being a malfunctioning or deprecated chainlink pricefeed for any used asset.

Calculating TVL of the pool is equivalent to value of all LP tokens so for more information see this post: https://blog.alphafinance.io/fair-lp-token-pricing/

Recommended Mitigation Steps

Calculate fair reserves using the pool invariant and the fair prices of the two assets.

The above link contains a mitigates for Uniswap, a similar calculation would have to be performed which is specific for the Vader invariant.

SamSteinGG (Vader) disputed and commented:

> The evaluation of liquidity for a particular pair is performed based on the reserves of the previous block rendering a flash loan attack impossible. Can the warden

> clarify how he is expecting this to be exploited?

Low Risk Findings (17)

- [L-01] unsafe cast *Submitted by certora*
- [L-02] Out of gas. *Submitted by Jujic, also found by jayjonah8, pauliax, and robee*
- [L-03] Core AMM logic is written to give the impression it is working on a different asset than it is. *Submitted by TomFrenchBlockchain*
- [L-04] SHOULD CHECK RETURN DATA FROM CHAINLINK AGGREGATORS (Timestamp) *Submitted by defsec, also found by danb, hyh, Jujic, and TomFrenchBlockchain*
- [L-05] `USDV.claim` Does Not Check If Index Is Valid *Submitted by leastwood*
- [L-06] VaderMath:calculateSwapReverse require statement change to <= instead of < *Submitted by GiveMeTestEther*
- [L-07] No Method To Remove Token Pairs *Submitted by leastwood, also found by gzeon*
- [L-08] `_addVaderPair` and `_addUSDVPair` Does Not Check For Duplicate Token Pairs *Submitted by leastwood, also found by gzeon*
- [L-09] USDV minting limit is not applied if `cycleTimestamp <= block.timestamp` *Submitted by TomFrenchBlockchain, also found by gzeon and pauliax*
- [L-10] mintSynth might mint nothing *Submitted by danb*
- [L-11] validateGas does nothing *Submitted by danb*
- [L-12] _addUSDVPair can also update *Submitted by pauliax*
- [L-13] loss of precision *Submitted by danb*
- [L-14] Lack of decimal control in StakingRewards *Submitted by 0x1f8b*
- [L-15] Lack Of Router Setter Function *Submitted by defsec*
- [L-16] setGasThrottle function should be moved to BasePoolV2 *Submitted by hyh*
- [L-17] vader can be initialized twice *Submitted by danb*

Non-Critical Findings (14)

- [N-01] transfer return value of a general ERC20 is ignored *Submitted by robee, also found by 0x1f8b, defsec, and jayjonah8*
- [N-02] Open TODOs in Codebase *Submitted by pauliax, also found by defsec, Dravee, and robee*
- [N-03] wrong revert message *Submitted by certora*
- [N-04] Never used parameters *Submitted by robee, also found by jayjonah8*
- [N-05] VaderMath:calculateSlipAdjustment() wrong comments *Submitted by GiveMeTestEther*
- [N-06] Lack of address(0) check *Submitted by Dravee*
- [N-07] transferOwnership should be two step process *Submitted by defsec*
- [N-08] Solidity compiler versions mismatch *Submitted by robee*
- [N-09] Lack of check of inputs in StakingRewards *Submitted by 0x1f8b*
- [N-10] VaderPoolV2 doesn't implement queue system yet *Submitted by hyh*
- [N-11] Open Discussion That Hint Potential Security Problem Should be Avoided *Submitted by MetaOxNull*
- [N-12] Incorrect descriptions of BasePoolV2's _onlyRouter and _supportedToken *Submitted by hyh*
- [N-13] USDV LockCreated event should include the index of a created lock *Submitted by hyh*
- [N-14] wrong comment *Submitted by certora*

Gas Optimizations (29)

- [G-01] SafeMath is not needed when using Solidity version 0.8.* *Submitted by Dravee*
- [G-02] An array's length should be cached to save gas in for-loops *Submitted by Dravee, also found by defsec and robee*
- [G-03] Storage double reading. Could save SLOAD *Submitted by robee, also found by Dravee, Jujic, and robee*
- [G-04] Storage of previous prices and total liquidity weights is suboptimal for gas costs *Submitted by TomFrenchBlockchain*

- [G-05] Shorter revert messages *Submitted by pauliax, also found by Dravee, Jujic, MetaOxNull, and robee*
- [G-06] Upgrade pragma to at least 0.8.4 *Submitted by robee*
- [G-07] Changing function visibility from public to external can save gas *Submitted by defsec, also found by Dravee and robee*
- [G-08] Unused imports *Submitted by robee*
- [G-09] Use bytes32 instead of string to save gas whenever possible *Submitted by robee*
- [G-10] Unused state variables *Submitted by robee*
- [G-11] State variables that could be set immutable *Submitted by robee, also found by defsec and TomFrenchBlockchain*
- [G-12] `++i` costs less gass compared to `i++` *Submitted by Dravee, also found by defsec and robee*
- [G-13] `uint a = b++;` is a confusing syntax and can be gas-optimized *Submitted by Dravee*
- [G-14] Internal functions can be private if the contract is not herited *Submitted by Dravee, also found by robee*
- [G-15] Explicit initialization with zero not required *Submitted by Dravee, also found by MetaOxNull and robee*
- [G-16] Redudant 2nd call to lastTimeRewardApplicable in StakingRewards.sol *Submitted by AmitN*
- [G-17] `> 0` can be replaced with `!= 0` for gas optimization *Submitted by defsec*
- [G-18] Save Gas With The Unchecked Keyword *Submitted by Dravee, also found by defsec and Jujic*
- [G-19] Modifier onlyUSDV() and function _onlyUSDV() *Submitted by Jujic*
- [G-29] Missing boundary check in USDV.sol *Submitted by p4st13r4*
- [G-20] Avoid repeated calculations *Submitted by pauliax*
- [G-21] Keccak functions in constants waste gas *Submitted by p4st13r4*
- [G-22] Redundant Constant Inheritance *Submitted by defsec*
- [G-23] Less than 256 uints are not gas efficient *Submitted by defsec*

- [G-24] Functions to calculate synth name/symbol should live in factory to reduce bytecode *Submitted by TomFrenchBlockchain*
- [G-25] Make use of a bitmap for claims to save gas in Converter.sol *Submitted by TomFrenchBlockchain*
- [G-26] Unnecessary checks on VADER token address in oracle. *Submitted by TomFrenchBlockchain*
- [G-27] Inclusion of salt and chainId in merkle tree leaves increases gas costs for no reason. *Submitted by TomFrenchBlockchain*
- [G-28] Unnecessary supportedToken checks on swaps on VaderPoolV2 *Submitted by TomFrenchBlockchain*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top