Vader Protocol contest
Findings & Analysis Report

2022-02-10

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 Vader Protocol contest smart contract system written in Solidity. The code contest took place between November 9—November 15 2021.

Wardens

29 Wardens contributed reports to the Vader Protocol contest:

1. TomFrenchBlockchain
2. cmichel
3. jayjonah8
4. rfa
5. shri4net
6. xYrYuYx
7. WatchPug (jtp and ming)
8. jonah1005
9. Reigada
10. defsec
11. leastwood
12. gzeon
13. pauliax
14. hack3r-0m
15. hyh
16. elprofesor
17. Ruhum
18. Meta0xNull
19. ye0lde
20. pants
21. pmerkleplant
22. ksk2345
23. fatima_naz
24. nathaniel
25. mics
26. 0xean
27. 0x0x0x
28. evo5555
29. m3kk_kn1ght

This contest was judged by Alberto Cuesta Cañada.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 98 unique vulnerabilities and 190 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 34 received a risk rating in the category of HIGH severity, 23 received a risk rating in the category of MEDIUM severity, and 41 received a risk rating in the category of LOW severity.

C4 analysis also identified 36 non-critical recommendations and 56 gas optimizations.

Scope

The code under review can be found within the C4 Vader Protocol contest repository, and is composed of 87 smart contracts written in the Solidity programming language and includes 7573 lines of Solidity code and 95 lines of JavaScript.

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](https://code4rena.com).

High Risk Findings (34)

[H-01] Minting and burning synths exposes users to unlimited slippage

Submitted by TomFrenchBlockchain, also found by cmichel

Impact

The amount of synths minted / assets received when minting or burning synths can be manipulated to an unlimited extent by manipulating the reserves of the pool

Proof of Concept

See `VaderPool.mintSynth`: https://github.com/code-423n4/2021-11-vader/blob/607d2b9e253d59c782e921bfc2951184d3f65825/contracts/dex-v2/pool/VaderPoolV2.sol#L126-L167

Here a user sends `nativeDeposit` to the pool and the equivalent amount of `foreignAsset` is minted as a synth to be sent to the user. However the user can't specify the minimum amount of synth that they would accept. A frontrunner can then manipulate the reserves of the pool in order to make `foreignAsset` appear more valuable than it really is so the user receives synths which are worth much less than what `nativeDeposit` is worth. This is equivalent to a swap without a slippage limit.

Burning synths essentially runs the same process in behalf so manipulating the pool in the opposite direction will result in the user getting fewer of `nativeAsset` than they expect.

Recommended Mitigation Steps

Add a argument for the minimum amount of synths to mint or nativeAsset to receive.

SamSteinGG (Vader) acknowledged and disagreed with severity:

> We believe the severity should be set to medium as there are no loss of funds and its exploit requires special circumstances to be profitable.

[H-02] Redemption value of synths can be manipulated to drain `VaderPool` of all native assets

Submitted by TomFrenchBlockchain

Impact

Draining of funds from `VaderPool`

Proof of Concept

See the `VaderPool.mintSynth` function: https://github.com/code-423n4/2021-11-vader/blob/607d2b9e253d59c782e921bfc2951184d3f65825/contracts/dex-v2/pool/VaderPoolV2.sol#L126-L167

As the pool's reserves can be manipulated through flashloans similar to on UniswapV2, 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

Prevent minting of synths or at the very least tie the exchange rate to a manipulation resistant oracle.

[H-03] VADER contains a Fee-On-Transfer

Submitted by jayjonah8, also found by rfa, shri4net, and xYrYuYx

Impact

The whitepaper says that the Vader token contains a Fee-On-Transfer so in XVader.sol, an attacker may be able to keep calling `enter()` and `leave()` while being credited more tokens than the contract actually receives eventually draining it.

Proof of Concept

1. Attacker deposits 500 Vader
2. Attacker receives credit for 500 while the xVader contract gets the 500 - fee.
3. Attacker calls `leave()` leaving the contract with a difference of the fee.
4. https://www.financegates.net/2021/07/28/another-polygon-yield-farm-crashes-to-zero-after-exploit/
5. https://github.com/code-423n4/2021-11-vader/blob/main/contracts/x-vader/XVader.sol
6. https://www.vaderprotocol.io/whitepaper

Tools Used

Manually code review

Recommended Mitigation Steps

There should be pre and post checks on balances to get the real amount

[0xstormtrooper (Vader) acknowledged]():

> Vader fee on transfer will be removed

[H-04] TwapOracle doesn't calculate VADER:USDV exchange rate correctly

Submitted by TomFrenchBlockchain

Impact

Detailed description of the impact of this finding.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/twap/TwapOracle.sol#L156

On L156 of `TwapOracle` we perform the calculation:

```
result = ((sumUSD * IERC20Metadata(token).decimals()) / sumNativ
```

This seems extremely odd as for an 18 decimal token we're then calculating

```
result = ((sumUSD * 18) / sumNative);
```

This is just plain weird. I expect what was meant is to replace this line with the below so we're properly scaling for `token` 's number of decimals.

```
uint256 scalingFactor = 10 ** IERC20Metadata(token).decimals()
result = (sumUSD * scalingFactor) / sumNative;
```

Marked as high severity as this exchange rate appears to be used in [some form of minting mechanism](#) and correctness of the oracle is listed as one of the key focuses of the audit.

Recommended Mitigation Steps

As above.

[SamSteinGG (Vader) confirmed](https://github.com/code-423n4/2021-11-vader-findings)

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[H-05] LPs of VaderPoolV2 can manipulate pool reserves to extract funds from the reserve.

Submitted by TomFrenchBlockchain, also found by WatchPug

Impact

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-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex-v2/pool/VaderPoolV2.sol#L237-L269

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

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex-v2/router/VaderRouterV2.sol#L208-L227

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-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex/math/VaderMath.sol#L73-L93

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.

Recommended Mitigation Steps

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

[H-06] Paying IL protection for all VaderPool pairs allows the reserve to be drained.

Submitted by TomFrenchBlockchain

Impact

Vader Reserve can be drained of funds.

Proof of Concept

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

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex/pool/VaderPool.sol#L77-L89

These losses are then refunded to the LP in VADER tokens from the reserve. NOTE: This IL protection is paid for ALL token pairs. THIS IS IMPORTANT!

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex/router/VaderRouter.sol#L187-L206

The loss is calculated by the comparing the amounts of each asset initially added to the pool against the amounts of each asset which are removed from the pool. There's an unspoken assumption here that the LP entered the pool at the true price at that point.

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex/math/VaderMath.sol#L73-L93

Crucially we see that if an attacker can cheaply create a pool with a token which starts off with a very low price in terms of VADER and is guaranteed to have a very high price in terms of VADER then they will benefit from a large amount of IL protection funds from the reserve.

An attacker could then perform this attack with the following.

1. Flashloan a huge amount of Vader (or flashloan + buy VADER).
2. Deploy a token TKN, which the attacker can mint as much as they like.
3. Add liquidity to a new pool with a large amount of VADER and a small amount of TKN
4. Use their ability to mint TKN to buy up all the VADER in their pool
5. Repay flashloan with VADER extracted from pool + some pre-existing funds as attacker needs to cover VADER lost to swap fees/slippage.

The attacker has now engineered a liquidity position which looks massively underwater due to IL but in reality was very cheap to produce. Nobody else can do anything to this pool except just give the attacker money by buying TKN so this attack can't be prevented. The attacker now just needs to wait for at most a year for the IL protection to tick up and then they can cash in the LP position for a nice payday of up to the amount of VADER they initially added to the pool.

Recommended Mitigation Steps

Add a whitelist to the pairs which qualify for IL protection.

SamSteinGG (Vader) disputed:

> Predicting the price fluctuations of an asset is impossible. An attacker cannot create a pool arbitrarily as that is governed by a special whitelist function that is in turn voted on by the DAO.

alcueca (judge) commented:

> As we saw in other issues, the creation of pools is permissionless

SamSteinGG (Vader) commented:

> @alcueca Again, there seems to be confusion as to the versions utilized. The submitter references the Vader V2 implementation in which pool creations are indeed permissioned (via the add supported token function) as the Vader pool factory is only relevant to the V1 implementation.

[H-07] VaderReserve does not support paying IL protection out to more than one address, resulting in locked funds

Submitted by TomFrenchBlockchain

Impact

All liquidity deployed to one of `VaderPool` or `VaderPoolV2` will be locked permanently.

Proof of Concept

Both `VaderRouter` and `VaderRouterV2` make calls to `VaderReserve` in order to pay out IL protection.

- https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex/router/VaderRouter.sol#L206
- https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex-v2/router/VaderRouterV2.sol#L227

However `VaderReserve` only allows a single router to claim IL protection on behalf of users.

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/reserve/VaderReserve.sol#L80-L83

It’s unlikely that the intent is to deploy multiple reserves so there’s no way for both `VaderRouter` and `VaderRouterV2` to pay out IL protection simultaneously.

This is a high severity issue as any LPs which are using the router which is not listed on `VaderReserve` will be unable to remove liquidity as the call to the reserve will revert. Vader governance is unable to update the allowed router on `VaderReserve` so all liquidity on either `VaderPool` or `VaderPoolV2` will be locked permanently.

Recommended Mitigation Steps

Options:

1. Allow the reserve to whitelist multiple addresses to claim funds
2. Allow the call to the reserve to fail without reverting the entire transaction (probably want to make this optional for LPs)

SamSteinGG (Vader) disputed:

> As the code indicates, only one of the two versioned instances of the AMM will be deployed and active at any given time rendering this exhibit incorrect.

[alcueca (judge) commented](#):

> Sorry @SamSteinGG, where does the code indicate that?

[SamSteinGG (Vader) commented](#):

> Correction, this was clarified during the audit in the discord channel.

[H-08] USDV and VADER rate can be wrong

Submitted by xYrYuYx

Impact

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L166

`tUSDInUSDV` can be smaller than `tUSDInVader`, and then `getRate` will return 0. This will lead wrong rate calculation.

Tools Used

Manually

Recommended Mitigation Steps

Multiple enough decimals before division

[SamSteinGG (Vader) confirmed](#)

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[H-09] VaderPoolV2 incorrectly calculates the amount of IL protection to send to LPs

Submitted by TomFrenchBlockchain

Impact

The `VaderReserve` pays out IL from `VaderPoolV2` LPs expressed in USDV with VADER (assuming a 1:1 exchange rate)

Proof of Concept

From the TwapOracle, it can be seen that `VaderPoolV2` is intended to be deployed with USDV as its `nativeAsset`:

- https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/twap/TwapOracle.sol#L281-L296
- https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex-v2/pool/BasePoolV2.sol#L58-L59

All the pairs in `VaderPoolV2` are then USDV:TKN where TKN is some other token, exactly which is irrelevant in this case.

`VaderPoolV2` offers IL protection where any IL is refunded from the `VaderReserve`

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex-v2/pool/VaderPoolV2.sol#L258-L268

The `VaderReserve` holds a balance of VADER tokens which will be used to pay out this protection.

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/reserve/VaderReserve.sol#L76-L90

The IL experienced by the LP is calculated in `VaderMath.calculateLoss`

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/dex/math/VaderMath.sol#L73-L93

This is the core of the issue. From the variable names it's clear that this is written with the assumption that it is work on units of VADER whereas it is provided amounts in terms of USDV. Checking `VaderRouterV2` we can see that we pass the output of this calculation directly to the reserve in order to claim VADER.

If an LP experienced 100 USDV worth of IL, instead of claiming the equivalent amount of VADER they would receive exactly 100 VADER as there's no handling of the exchange rate between USDV and VADER.

As VADER and USDV are very unlikely to trade at parity LPs could get sustantially more or less than the amount of IL they experienced.

Recommended Mitigation Steps

Add handling for the conversion rate between VADER and USDV using a tamper resistant oracle (TwapOracle could potentially fulfil this role).

[SamSteinGG (Vader) confirmed]()

[H-10] calculate Loss is vulnerable to flashloan attack

Submitted by jonah1005

Impact

The VaderPool would compensate users' IL. The formula it uses to calculate lp value is vulnerable to manipulation.

The formula to calculate the lp value is similar to warp finance which is known to be unsafe. [warpfinance-incident-root-cause-analysis]() (Please to refer to the POC section)

The Attacker can purchase an old lp position, manipulate price, take IL compensation and drain the reserve. I consider this is a high-risk issue.

Proof of Concept

[VaderMath.sol#L69-L93]()

The lp value is calculated as `[(A0 * P1) + V0]` and `// [(A1 * P1) + V1]`. Assume that there's an ETH pool and there's 100 ETH and 100 Vader in the pool.

1. Attacker deposit 1 ETH and 1 Vader and own 1% of the liquidity.
2. Wait 1 year
3. Start flash loan and buy a lot ETH with 99900 Vader.
4. There's 0.1 ETH 100,000 Vader in the pool.
5. Burn 1 % lp at the price 1 ETH = 1,000,000 Vader.
6. A0 * P1 + V0 = 1 (eth) * 1,000,000 (price) + 100 (vader)
7. A1 * P1 + V1 = 0.001 (eth) * 1,000,000 (price) + 10,000 (vader)
8. IL compensation would be around `9891000`.

Tools Used

None

Recommended Mitigation Steps

Please use the fair lp pricing formula from alpha finance instead. fair-lp-token-pricing

SamSteinGG (Vader) disputed:

> The described attack scenario can not be executed as the pool would actually consume the flash loan. The CLP model follows a non-linear curve that actually diminishes in value as the trade size increases, meaning that at most 25% of the total assets in the pool can be drained at a given iteration. This, on top with the fees of each transaction render this attack vector impossible. Please request a tangible attack test from the warden if this is meant to be accepted as valid.

alcueca (judge) commented:

> The CLP model isn't mentioned in the readme or the whitepaper. The issue is valid according to the materials supplied.

SamSteinGG (Vader) commented:

> @alcueca As the grading guidelines of C4 state, a documentation issue cannot constitute more than a low risk finding. We advise the severity to be lowered.

[H-11] (dex-v1) BasePool.mint() function can be frontrun

Submitted by Reigada

Impact

In the contract BasePool the mint function can be frontrun. This will assign the NFT to the attacker which later on he can burn it retrieving the corresponding `\_nativeAsset` and `\_foreignAsset` initially deposited by the frontrun victim.
https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/pool/BasePool.sol#L149-L194

Proof of Concept

User1 transfers 1000 `\_nativeAsset` tokens and 1000 `\_foreignAsset` tokens into the BasePool contract. User1 calls the `BasePool.mint()` function to retrieve his NFT. Attacker is constantly polling for an increase of the balance of `\_nativeAsset` and `\_foreignAsset` of the contract OR attacker is constantly scanning the mempool for `mint()` function calls. Attacker detects an increase of balance of `\_nativeAsset` and `\_foreignAsset` OR attacker detects a `mint()` function call in the mempool. Attacker frontruns the mint call and retrieves the NFT. Gets a NFT that is worth 1000 `\_nativeAssets` and 1000 `\_foreignAssets`. User1 gets a NFT that is worth 0 `\_nativeAssets` and 0 `\_foreignAssets`. Attacker burns the NFT retrieving the corresponding `\_nativeAsset` and `\_foreignAsset` initially deposited by the victim.

Tools Used

Manual testing

Recommended Mitigation Steps

Include in the `mint()` function the transfer of `\_nativeAssets` and `\_foreignAssets` to the smart contract.

SamSteinGG (Vader) disputed:

> The pool is meant to be utilized via the router or smart contracts and is not meant to be utilized directly. The exact same “flaw” exists in Uniswap V2 whereby if you transfer assets directly someone else can claim them on your behalf.

alcueca (judge) commented:

> Ah, so this how you prevent direct access to the pools. The issue is valid due to lack of documentation on the usage of the router.

SamSteinGG (Vader) commented:

> Firstly, documentation related issues cannot constitute a high risk vulnerability. Secondly, this type of documentation does not exist in Uniswap V2 either. We advise this finding to be set to no risk.

[H-12] Attacker can get extremely cheap synth by front-running create Pool

Submitted by jonah1005, also found by defsec

Impact

`createPool` is a permissionless transaction.

1. Anyone can create a token pool.
2. Token price is set by the first lp provider.
3. User can get a synthetic asset.

Assume a new popular `coin` that the DAO decides to add to the protocol. The attacker can create the pool and set it to be extremely cheap. (By depositing 1 wei `coin` and 10^18 wei Vader.) The attacker can mint a lot of synth by providing another 10^18 wei Vader.

There’s no way to revoke the pool. The `coin` pool would be invalid since the attacker can drain all the lp in the pool.

I consider this is a high-risk issue.

Proof of Concept

- [VaderPoolFactory.sol#L43-L89]
- [VaderPoolV2.sol#L115-L167]

Tools Used

None

Recommended Mitigation Steps

Restrict users from minting synth from a new and illiquid pool. Some thoughts about the fix:

1. Decide minimum liquidity for a synthetic asset (e.g 1M Vader in the pool)
2. Once there's enough liquidity pool, anyone can deploy a synthetic asset after a cool down. (e.g. 3 days

The pool can remain permissionless and safe.

SamSteinGG (Vader) disputed:

> This is an invalid finding as creating pools is not a permissionless operation, the token must be in the supported list of assets.

alcueca (judge) commented:

> I can't see a check for a token to be in a supported list of assets.

SamSteinGG (Vader) commented:

> @alcueca There seems to be some confusion. The submitted of the bounty links the Vader Pool Factory of DEX V1 and the Pool of DEX V2 which are not interacting between them. As such, the finding is invalid.

[H-13] Anyone Can Arbitrarily Mint Synthetic Assets In `VaderPoolV2.mintSynth()`

Submitted by leastwood

Impact

The `mintSynth()` function is callable by any user and creates a synthetic asset against `foreignAsset` if it does not already exist. The protocol expects a user to first approve the contract as a spender before calling `mintSynth()`. However, any arbitrary user could monitor the blockchain for contract approvals that match `VaderPoolV2.sol` and effectively frontrun their call to `mintSynth()` by setting the `to` argument to their own address. As a result, the `nativeDeposit` amount is transferred from the victim, and a synthetic asset is minted and finally transferred to the malicious user who is represented by the `to` address.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L126-L167

Tools Used

Manual code review. Discussions with dev.

Recommended Mitigation Steps

Consider removing the `from` argument in `mintSynth()` and update the `safeTransferFrom()` call to instead transfer from `msg.sender`.

SamSteinGG (Vader) commented:

> The pool contracts, similarly to Uniswap V2, are never meant to be interacted with directly.

[H-14] Anyone Can Arbitrarily Mint Fungible Tokens In `VaderPoolV2.mintFungible()`

Submitted by leastwood

Impact

The `mintFungible()` function is callable by any user that wishes to mint liquidity pool fungible tokens. The protocol expects a user to first approve the contract as a spender before calling `mintFungible()`. However, any arbitrary user could monitor the blockchain for contract approvals that match `VaderPoolV2.sol` and effectively frontrun their call to `mintFungible()` by setting the `to` argument to their own address. As a result, the `nativeDeposit` and `foreignDeposit` amounts are transferred from the victim, and LP tokens are minted and finally transferred to the malicious user who is represented by the `to` address.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L284-L335

Tools Used

Manual code review. Discussions with dev.

Recommended Mitigation Steps

Consider removing the `from` argument in `mintFungible()` and update the `safeTransferFrom()` calls to instead transfer from `msg.sender`.

SamSteinGG (Vader) disputed:

> The pool contracts, similarly to Uniswap V2, are never meant to be interacted with directly.

alcueca (judge) commented:

> You need to enforce that somehow.

SamSteinGG (Vader) confirmed:

> Upon second consideration, the functions relating to the minting of synths and wrapped tokens should have had the onlyRouter modifier and thus are indeed vulnerable. Issue accepted.

[H-15] `VaderRouter._swap` performs wrong swap

Submitted by cmichel

The 3-path hop in `VaderRouter._swap` is supposed to first swap **foreign** assets to native assets, and then the received native assets to different foreign assets again.

The `pool.swap(nativeAmountIn, foreignAmountIn)` accepts the foreign amount as the **second** argument. The code however mixes these positional arguments up and tries to perform a `pool0` foreign -> native swap by using the **foreign** amount as the **native amount:**

```
function _swap(
    uint256 amountIn,
    address[] calldata path,
    address to
) private returns (uint256 amountOut) {
    if (path.length == 3) {
      // ...
      // @audit calls this with nativeAmountIn = amountIn. but s
      return pool1.swap(0, pool0.swap(amountIn, 0, address(pool1
    }
}

// @audit should be this instead
return pool1.swap(pool0.swap(0, amountIn, address(pool1)), 0, tc
```

Impact

All 3-path swaps through the `VaderRouter` fail in the pool check when `require(nativeAmountIn = amountIn <= nativeBalance - nativeReserve = 0)` is checked, as foreign amount is sent but *native* amount is specified.

Recommended Mitigation Steps

Use `return pool1.swap(pool0.swap(0, amountIn, address(pool1)), 0, to);` instead.

SamSteinGG (sponsor) confirmed

[H-16] `VaderRouter.calculateOutGivenIn` calculates wrong swap

Submitted by cmichel

The 3-path hop in `VaderRouter.calculateOutGivenIn` is supposed to first swap **foreign** assets to native assets **in pool0**, and then the received native assets to different foreign assets again **in pool1**.

The first argument of `VaderMath.calculateSwap(amountIn, reserveIn, reserveOut)` must refer to the same token as the second argument `reserveIn`. The code however mixes these positions up and first performs a swap in `pool1` instead of `pool0`:

```
    function calculateOutGivenIn(uint256 amountIn, address[] calldat
        external
        view
        returns (uint256 amountOut)
    {
      if(...) {
      } else {
        return
            VaderMath.calculateSwap(
                VaderMath.calculateSwap(
                    // @audit the inner trade should not be in pool1
                    amountIn,
                    nativeReserve1,
                    foreignReserve1
                ),
                foreignReserve0,
                nativeReserve0
            );
      }

     /** @audit instead should first be trading in pool0!
        VaderMath.calculateSwap(
            VaderMath.calculateSwap(
                amountIn,
                foreignReserve0,
                nativeReserve0
            ),
            nativeReserve1,
            foreignReserve1
        );
      */
```

Impact

All 3-path swaps computations through `VaderRouter.calculateOutGivenIn` will return the wrong result. Smart contracts or off-chain scripts/frontends that rely on this value to trade will have their transaction reverted, or in the worst case lose funds.

Recommended Mitigation Steps

Return the following code instead which first trades in `pool0` and then in `pool1`:

```
return
  VaderMath.calculateSwap(
      VaderMath.calculateSwap(
          amountIn,
          foreignReserve0,
          nativeReserve0
      ),
      nativeReserve1,
      foreignReserve1
  );
```

[SamSteinGG (Vader) confirmed]

[H-17] TWAPOracle might register with wrong token order

Submitted by cmichel

The `TWAPOracle.registerPair` function takes in a `factory` and (`token0, token1`). The function accepts a `_factory` argument which means any Uniswap-like factory can be used.

When using the actual Uniswap factory's `IUniswapV2Factory(factory).getPair(token0, token1)` call, it could be that the `token0` and `token1` are reversed as it [ignores the order].

Meaning, the `price0/1CumulativeLast` could also be reversed as it matches the internal order. The code however pushes the `_pairs` assuming that the internal `price0CumulativeLast, price1CumulativeLast` order matches the order of the function arguments `token0, token1`.

```
    _pairs.push(
        PairData({
            pair: pairAddr,
            token0: token0,
            token1: token1,
            price0CumulativeLast: price0CumulativeLast,
            price1CumulativeLast: price1CumulativeLast,
            blockTimestampLast: blockTimestampLast,
            price0Average: FixedPoint.uq112x112({_x: 0}),
            price1Average: FixedPoint.uq112x112({_x: 0})
        })
    );
```

Impact

The prices could be inverted which leads to the oracle providing wrong prices.

Recommended Mitigation Steps

It should be checked if Uniswap's internal order matches the order of the `token0/1` function arguments. If not, the cumulative prices must be swapped.

```
    // pseudocode
    IUniswapV2Pair pair = IUniswapV2Pair(
        IUniswapV2Factory(factory).getPair(token0, token1)
    );
    pairAddr = address(pair);
    price0CumulativeLast = pair.price0CumulativeLast();
    price1CumulativeLast = pair.price1CumulativeLast();
    (price0CumulativeLast, price1CumulativeLast) = token0 == pair.tc
```

> The same issue exists in `update`

SamSteinGG (Vader) confirmed

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[H-18] Attacker can claim more IL by manipulating pool price then `removeLiquidity`

Submitted by gzeon

Impact

Vader reimburse user IL immediately when user withdraw from the pool (VaderRouterV2.sol:L227), an attacker can therefore manipulate the pool balance causing a high IL, remove liquidity and restore the pool balance such that he will receive a larger IL reimbursement.

Proof of Concept

Let’s assume our attacker own 100% of FOO-VADER

1. Attacker add 100 FOO and 100 VADER to the Pool
2. wait some block, or 1 year for max IL protection
3. In 1 transaction, attacker
    - Swap 9900 FOO to 99 Vader
    - Pool now have 10000 FOO and 1 VADER
    - By VaderMath.sol:L84 the loss is 100*1/10000+100-2 = 98.01 VADER
    - Remove liquidity and receive 10000 FOO and 99.01 VADER
    - Restore the pool balance
4. Such that the attacker will gain 98.01 VADER without risk

The profit is constrained by gas cost, pool fee, % of pool controlled by the attacker and % of IL protection.

Recommended Mitigation Steps

Use twap price to determine P1 in VaderMath.sol:L84 when calculating IL to reduce risk of manipulation

SamSteinGG (Vader) commented:

> Duplicate of #2

alcueca (judge) commented:

> Doesn't seem like a duplicate to me, @SamSteinGG?

SamSteinGG (Vader) commented:

> @alcueca The stated trade cannot occur as trades are inherently limited by the CLP design of the protocol to one third of the available pair liquidity. As such, the illustrated pair would actually result in almost zero units retrieved back.

[H-19] Governance veto can be bypassed

Submitted by gzeon

Impact

Since `veto` ensure none of the actions in proposal being vetoed point to the contract (GovernorAlpha.sol:L562), a malicious proposal can be designed to have an action that point to governance and therefore effectively cannot be vetoed.

Proof of Concept

For any attacker who want to launch a governance attack using a malicious proposal, they simply need to add an action that point to governance that does nothing (or anything).

Recommended Mitigation Steps

Some other design can be proposal are vetoable whenever the differential is less than x%, even if it involves governance change, s.t. council can veto most malicious proposal while it is still possible to change council given high enough vote differential.

SamSteinGG (Vader) commented:

> Duplicate of #61

alcueca (judge) commented:

> Not a duplicate

[H-20] Early user can break `addLiquidity`

Submitted by WatchPug

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex/pool/BasePool.sol#L161-L163

```
    uint256 totalLiquidityUnits = totalSupply;
    if (totalLiquidityUnits == 0)
        liquidity = nativeDeposit; // TODO: Contact ThorChain on prc
```

In the current implementation, the first `liquidity` takes the `nativeDeposit` amount and uses it directly.

However, since this number (`totalLiquidityUnits`) will later be used for computing the `liquidity` issued for future `addLiquidity` using `calculateLiquidityUnits` .

A malicious user can add liquidity with only `1 wei` USDV and making it nearly impossible for future users to add liquidity to the pool.

Recommendation

Uni v2 solved this problem by sending the first 1000 tokens to the zero address.

The same should work here, i.e., on first mint (totalLiquidityUnits == 0), lock some of the first minter's tokens by minting ~1% of the initial amount to the zero address instead of to the first minter.

SamSteinGG (Vader) commented:

> Duplicate of #24

alcueca (judge) commented:

> Not a duplicate

[H-21] Lack of access control allow attacker to `mintFungible()` and `mintSynth()` with other user's wallet balance

Submitted by WatchPug

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/VaderPoolV2.sol#L284-L335

```
    function mintFungible(
            IERC20 foreignAsset,
            uint256 nativeDeposit,
            uint256 foreignDeposit,
            address from,
            address to
        ) external override nonReentrant returns (uint256 liquidity)
            IERC20Extended lp = wrapper.tokens(foreignAsset);

            require(
                lp != IERC20Extended(_ZERO_ADDRESS),
                "VaderPoolV2::mintFungible: Unsupported Token"
            );

            (uint112 reserveNative, uint112 reserveForeign, ) = getF
                foreignAsset
            ); // gas savings

            nativeAsset.safeTransferFrom(from, address(this), native
            foreignAsset.safeTransferFrom(from, address(this), forei

            PairInfo storage pair = pairInfo[foreignAsset];
            uint256 totalLiquidityUnits = pair.totalSupply;
            if (totalLiquidityUnits == 0) liquidity = nativeDeposit;
            else
                liquidity = VaderMath.calculateLiquidityUnits(
                    nativeDeposit,
                    reserveNative,
                    foreignDeposit,
                    reserveForeign,
                    totalLiquidityUnits
                );

            require(
                liquidity > 0,
                "VaderPoolV2::mintFungible: Insufficient Liquidity I
            );

            pair.totalSupply = totalLiquidityUnits + liquidity;

            _update(
                foreignAsset,
                reserveNative + nativeDeposit,
                reserveForeign + foreignDeposit,
                reserveNative,
                reserveForeign
            );

            lp.mint(to, liquidity);

            emit Mint(from, to, nativeDeposit, foreignDeposit);
        }
```

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/VaderPoolV2.sol#L126-L167

Funds are transferred from the `from` parameter, and the output tokens are transferred to the `to` parameter, both passed by the caller without proper access control.

Impact

This issue allows anyone to call `mintFungible()` and `mintSynth()` and steal almost all their wallet balances for all the users who have approved the contract before.

[SamSteinGG (Vader) commented](#):

> Duplicate #67

[alcueca (judge) commented]():

> Not a duplicate.

[SamSteinGG (Vader) commented]():

> @alcueca Can you elaborate as to why it is not a duplicate?

[H-22] `mintSynth()` and `burnSynth()` can be front run

Submitted by WatchPug

- https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/VaderPoolV2.sol#L126-L155
- https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/VaderPoolV2.sol#L179-L197

Given that `mintSynth()` and `burnSynth()` will issue and redeem assets based on the price of the pool (reserves), and they will create price impact based on the volume being minted and burnt.

However, the current implementation provides no parameter for slippage control, making them vulnerable to front-run attacks. Especially for transactions with rather large volumes.

Recommendation

Consider adding a `minAmountOut` parameter.

[H-23] `Synth` tokens can get over-minted

Submitted by WatchPug

Per the document:

> It also is capable of using liquidity units as collateral for synthetic assets, of which it will always have guaranteed redemption liquidity for.

However, in the current implementation, `Synth` tokens are minted based on the calculation result. While `nativeDeposit` be added to the reserve, `reserveForeign` will remain unchanged, not deducted nor locked.

Making it possible for `Synth` tokens to get over-minted.

Proof of Concept

- The Vader pool for BTC-USDV is newly created, with nearly 0 liquidity.
- Alice add liquidity with `100,000 USDV` and `1 BTC`;
- Bob `mintSynth()` with `100,000 USDV`, got `0.25 BTC vSynth`;
- Alice remove all the liquidity received at step 1, got all the `200k USDV` and `1 BTC`.

The `0.25 BTC vSynth` held by Bob is now backed by nothing and unable to be redeemed.

This also makes it possible for a sophisticated attacker to steal funds from the Vader pool.

The attacker may do the following in one transaction:

1. Add liquidity with `10 USDV` and `10,000 BTC` (flash loan);
2. Call `mintSynth()` with `10 USDV`, repeat for 10 times, got `1461 BTC vSynth`;
3. Remove liquidity and repay flash loan, keep the `1461 BTC vSynth`;
4. Wait for other users to add liquidity and when the BTC reserve is sufficient, call `burnSynth()` to steal `USDV` from the pool.

SamSteinGG (Vader) confirmed

> Given that the codebase attempts to implement the Thorchain rust code in a one-to-one fashion, findings that relate to the mathematical accuracy of the codebase will only be accepted in one of the following cases:

- The code deviates from the Thorchain implementation
- A test case is created that illustrates the problem

> While intuition is a valid ground for novel implementations, we have re-implemented a battle-tested implementation in another language and as such it is considered secure by design unless proven otherwise.

[H-24] Wrong design/implementation of `addLiquidity()` allows attacker to steal funds from the liquidity pool

Submitted by WatchPug

The current design/implementation of Vader pool allows users to `addLiquidity` using arbitrary amounts instead of a fixed ratio of amounts in comparison to Uni v2.

We believe this design is flawed and it essentially allows anyone to manipulate the price of the pool easily and create an arbitrage opportunity at the cost of all other liquidity providers.

An attacker can exploit this by adding liquidity in extreme amounts and drain the funds from the pool.

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/VaderPoolV2.sol#L284-L335

```
    function mintFungible(
        IERC20 foreignAsset,
        uint256 nativeDeposit,
        uint256 foreignDeposit,
        address from,
        address to
    ) external override nonReentrant returns (uint256 liquidity) {
        IERC20Extended lp = wrapper.tokens(foreignAsset);

        require(
            lp != IERC20Extended(_ZERO_ADDRESS),
            "VaderPoolV2::mintFungible: Unsupported Token"
        );

        (uint112 reserveNative, uint112 reserveForeign, ) = getReser
            foreignAsset
        ); // gas savings
        nativeAsset.safeTransferFrom(from, address(this), nativeDepo
        foreignAsset.safeTransferFrom(from, address(this), foreignDe

        PairInfo storage pair = pairInfo[foreignAsset];
        uint256 totalLiquidityUnits = pair.totalSupply;
        if (totalLiquidityUnits == 0) liquidity = nativeDeposit;
        else
            liquidity = VaderMath.calculateLiquidityUnits(
                nativeDeposit,
                reserveNative,
                foreignDeposit,
                reserveForeign,
                totalLiquidityUnits
            );

        require(
            liquidity > 0,
            "VaderPoolV2::mintFungible: Insufficient Liquidity Provi
        );

        pair.totalSupply = totalLiquidityUnits + liquidity;

        _update(
            foreignAsset,
            reserveNative + nativeDeposit,
            reserveForeign + foreignDeposit,
            reserveNative,
            reserveForeign
        );

        lp.mint(to, liquidity);

        emit Mint(from, to, nativeDeposit, foreignDeposit);
    }
```

Proof of Concept

Given:

- A Vader pool with `100,000 USDV` and `1 BTC`;
- The `totalPoolUnits` is `100`.

The attacker can do the following in one transaction:

1. Add liquidity with `100,000 USDV` and 0 BTC, get `50 liquidityUnits`, representing 1/3 shares of the pool;
2. Swap `0.1 BTC` to USDV, repeat for 5 times; spent `0.5 BTC` and got `62163.36 USDV`;
3. Remove liquidity, get back `45945.54 USDV` and `0.5 BTC`; profit for: 62163.36 + 45945.54 - 100000 = 8108.9 USDV.

SamSteinGG (Vader) disputed:

> This is the intended design of the Thorchain CLP model. Can the warden provide a tangible attack vector in the form of a test?

alcueca (judge) commented:

> Sponsor is acknowledging the issue.

SamSteinGG (Vader) commented:

> @alcueca We do not acknowledge the issue. This is the intended design of the CLP model and the amount supplied for a trade is meant to be safeguarded off-chain. It is an inherent trait of the model.

[H-25] Wrong design of `swap()` results in unexpected and unfavorable outputs

Submitted by WatchPug

The current formula to calculate the `amountOut` for a swap is:

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex/math/VaderMath.sol#L99-L111

```
    function calculateSwap(
        uint256 amountIn,
        uint256 reserveIn,
        uint256 reserveOut
    ) public pure returns (uint256 amountOut) {
        // x * Y * X
        uint256 numerator = amountIn * reserveIn * reserveOut;

        // (x + X) ^ 2
        uint256 denominator = pow(amountIn + reserveIn);

        amountOut = numerator / denominator;
    }
```

We believe the design (the formula) is wrong and it will result in unexpected and unfavorable outputs.

Specifically, if the `amountIn` is larger than the `reserveIn`, the `amountOut` starts to decrease.

Proof of Concept

Given:

- A USDV-BTC Vader pool with the reserves of `200,000 USDV` and `2 BTC`.
- If Alice swap `2 BTC` for USDV, will get `50000 USDV` as output;
- If Bob swap `2.1 BTC` for USDV, will only get `49970.25 USDV` as output;
- If Carol swap `2.2 BTC` for USDV, will only get `49886.62 USDV` as output.

For the same pool reserves, paying more for less output token is unexpected and unfavorable.

SamSteinGG (Vader) disputed:

> This is the intended design of the Thorchain CLP model. Can the warden provide a tangible attack vector in the form of a test?

alcueca (judge) commented:

> It is true that the effect will be surprising to the user, and the issue is acknowledged by the sponsor.

SamSteinGG (Vader) commented:

> @alcueca We do not acknowledge the issue. This is the intended design of the CLP model and the amount supplied for a trade is meant to be safeguarded off-chain. It is an inherent trait of the model.

[H-26] All user assets which are approved to VaderPoolV2 may be stolen

Submitted by TomFrenchBlockchain, also found by cmichel

Impact

Total loss of funds which have been approved on `VaderPoolV2`

Proof of Concept

`VaderPoolV2` allows minting of fungible LP tokens with the `mintFungible` function

https://github.com/code-423n4/2021-11-vader/blob/607d2b9e253d59c782e921bfc2951184d3f65825/contracts/dex-v2/pool/VaderPoolV2.sol#L284-L290

Crucially this function allows a user supplied value for `from` which specifies where the `nativeAsset` and `foreignAsset` should be pulled from. An attacker can then provide any address which has a token approval onto `VaderPoolV2` and mint themselves LP tokens - stealing the underlying tokens.

Recommended Mitigation Steps

Remove `from` argument and use msg.sender instead.

[SamSteinGG (Vader) disputed]():

> pool is not meant to be interacted with

[alcueca (judge) commented]():

> And how are you going to ensure that the pool is not interacted with, @SamSteinGG?

SamSteinGG (Vader) confirmed:

> @alcueca Upon second consideration, the functions relating to the minting of synths and wrapped tokens should have had the onlyRouter modifier and thus are indeed vulnerable. Issue accepted.

[H-27] Unrestricted vestFor

Submitted by pauliax, also found by hack3r-0m

Impact

Anyone can call function `vestFor` and block any user with a tiny amount of Vader. This function has no auth checks so a malicious actor can front-run legit `vestFor` calls with insignificant amounts. This function locks the user for 365 days and does not allow updating the value, thus forbids legit conversions.

Recommended Mitigation Steps

Consider introducing a whitelist of callers that can vest on behalf of others (e.g. Converter).

SamSteinGG (Vader) confirmed

[H-28] Incorrect Price Consultation Results

Submitted by leastwood

Impact

The `TwapOracle.consult()` function iterates over all token pairs which belong to either `VADER` or USDV`and then calculates the price of the respective asset by using both UniswapV2 and Chainlink price data. This helps to further protect against price manipulation attacks as the price is averaged out over the various registered token pairs.

Let's say we wanted to query the price of `USDV`, we would sum up any token pair where `USDV == pairData.token0`.

The sum consists of the following:

- Price of `USDV` denominated in terms of `token1` (`USDV/token1`).
- Price of token1 denominated in terms of `USD` (`token1/USD`).

Consider the following example:

- `SUSHI` is the only registered token pair that exists alongside `USDV`.
- Hence, calculating `sumNative` gives us an exchange rate that is denominated as `USDV/SUSHI`.
- Similarly, `sumUSD` gives us the following denominated pair, `SUSHI/USD`.
- I'd expect the result to equal `sumUSD * token.decimals() * sumNative` which should give us a USDV/USD denominated result.

However, the protocol calculates it as `(sumUSD * token.decimals()) / sumNative` which gives us a `SUSHI^2 / (USD*USDV)` denominated result. This seems incorrect.

I'd classify this issue as high risk as the oracle returns false results upon being consulted. This can lead to issues in other areas of the protocol that use this data in performing sensitive actions.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L115-L157

Similar working implementation listed below:

- https://github.com/gg2001/dpx-oracle/blob/master/contracts/UniswapV2Oracle.sol#L184-L211
- https://github.com/gg2001/dpx-oracle/blob/master/contracts/UniswapV2Oracle.sol#L291-L304

Tools Used

Manual code review.

Recommended Mitigation Steps

To calculate the correct consultation of a given token, the result should return `sumUSD * token.decimals() * sumNative` instead to ensure the target token to consult is denominated in `USD` and contains the correct number of decimals.

[SamSteinGG (Vader) confirmed](https://github.com/code-423n4/2021-11-vader-findings):

> The description seems slightly incorrect as it uses a power where multiplication is performed but the general idea is correct.

[H-29] VaderPoolV2.mintFungible exposes users to unlimited slippage

Submitted by TomFrenchBlockchain

Impact

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

Proof of Concept

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

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/VaderPoolV2.sol#L271-L335

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 liqudity.

Recommended Mitigation Steps

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

[SamSteinGG (Vader) confirmed]

> Given that the codebase attempts to implement the Thorchain rust code in a one-to-one fashion, findings that relate to the mathematical accuracy of the codebase will only be accepted in one of the following cases:

- The code deviates from the Thorchain implementation
- A test case is created that illustrates the problem

> While intuition is a valid ground for novel implementations, we have re-implemented a battle-tested implementation in another language and as such it is considered secure by design unless proven otherwise.

> An additional note on this point is that any behaviour that the Thorchain model applies is expected to be the intended design in our protocol as well.

> An important example is the slippage a user incurs on joining a particular LP pool for which there is no check as there can't be any. Enforcing an LP unit based check here is meaningless given that LP units represent a share that greatly fluctuates (1 unit of LP out of 100 units is different than 1 out of 1000, however, a slippage check for 100 units of DAI for example is valid).

[H-30] Newly Registered Assets Skew Consultation Results

Submitted by leastwood

Impact

The `TwapOracle.consult()` function iterates over all token pairs which belong to either `VADER` or USDV`and then calculates the price of the respective asset by using both UniswapV2 and Chainlink price data. This helps to further protect against price manipulation attacks as the price is averaged out over the various registered token pairs.

If a new asset is added by first registering the token pair and aggregator, the consultation result for that token pair will remain skewed until the next update interval. This is due to the fact that the native asset amount will return `0` due to the default `price1Average` value being used. However, the Chainlink oracle will return a valid result. As a result, the query will be skewed in favour of `sumUSD` resulting in incorrect consultations.

I'd classify this issue as high risk as the oracle returns false results upon being consulted. This can lead to issues in other areas of the protocol that use this data in performing sensitive actions.

Proof of Concept

- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L115-L157
- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L314
- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L322-L369

Tools Used

Manual code review.

Recommended Mitigation Steps

Consider performing proper checks to ensure that if `pairData.price1Average._x == 0`, then the Chainlink aggregator is not queried and not added to `sumUSD`. Additionally, it may be useful to fix the current check to assert that the `pairData.price1Average.mul(1).decode144()` result is not `0`, found [here](). `require(sumNative != 0)` is used to assert this, however, this should be `require(pairData.price1Average.mul(1).decode144() != 0)` instead.

[SamSteinGG (Vader) confirmed]()

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[H-31] Unused slippage params

Submitted by pauliax, also found by TomFrenchBlockchain

Impact

Unused slippage params. function `addLiquidity` in VaderRouter (both V1 and V2) do not use slippage parameters:

```
    uint256, // amountAMin = unused
    uint256, // amountBMin = unused
```

making it susceptible to sandwich attacks / MEV. For a more detailed explanation, see: https://github.com/code-423n4/2021-09-bvecvx-findings/issues/57

Recommended Mitigation Steps

Consider paying some attention to the slippage to reduce possible manipulation attacks from mempool snipers.

SamSteinGG (Vader) disputed:

> Slippage checks are impossible in the Thorchain CLP model.

alcueca (judge) commented:

> Taking as main over #1 as it is a more general issue, but refer to #1 for a more detailed description and justification for the severity rating.

[H-32] Covering impermanent loss allows profiting off asymmetric liquidity provision at expense of reserve holdings

Submitted by hyh

Impact

Pool funds will be siphoned out over time as swaps and asymmetric LP provision are balancing each other economically, while with introduction of IL reimbursement a malicious user can profit immediately from out of balance pool with a swap and profit again from IL coverage. This requires locking liquidity to a pool, but still represents an additional profit without additional risk at expense of reserve funds.

Another variant of exploiting this is to add liquidity in two steps: deposit 1 with 0 slip adjustment, perfectly matching current market price, deposit 2 with more Vader than market price suggests, moving pool out of balance with Vader becoming cheaper, then exiting deposit 1 with profit because slip adjustment reduce deposit 2's share issuance and deposit 1's now has more asset claims than before. Deposit 2 then need to wait and exit after some time.

IL is calculated as `((originalAsset * releasedVader) / releasedAsset) + originalVader - ((releasedAsset * releasedVader) / releasedAsset) + releasedVader` , i.e. original deposit values without taking account of slip adjustment are used, so providing more Vader in deposit 2 leads to greater IL, which this way have 2 parts: market movements related and skewed liquidity provision related. IL covering compensates for slip adjustments this way.

Proof of Concept

The steps to reproduce are:

1. add asymmetric LP via mint (with NFT),
2. either swap gathering profit from pool skew or do symmetric deposit beforehand and exit it now
3. wait for some period for IL protection to be enabled, then withdraw, having IL covered by reserve fund

Router addLiquidity: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/router/VaderRouterV2.sol#L114

NFT mint: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/pool/BasePoolV2.sol#L168

Router removeLiquidity: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/router/VaderRouterV2.sol#L227

NFT burn: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L237

IL calculation: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/math/VaderMath.sol#L73

Recommended Mitigation Steps

Asymmetric liquidity provision doesn't provide much business value, introducing substantial attack surface, so the core recommendation here is to remove a possibility to add liquidity asymmetrically: instead of penalizing LP with slip adjustment do biggest liquidity addition with 0 slip adjustment that user provided funds allow, and return the remaining part.

This will also guard against cases when user added liquidity with big slip adjustment penalty without malicious intent, not realizing that this penalty will take place, an effect that poses reputational risk to any project using the approach.

Allowing only symmetric liquidity addition removes the described attack surface.

[SamSteinGG (Vader) marked as duplicate]()

[alcueca (judge) commented]():

> Duplicate of which other issue, @SamSteinGG?

[H-33] Mixing different types of LP shares can lead to losses for Synth holders

Submitted by hyh

Impact

Users that mint Synths do not get pool shares, so exiting of normal LP can lead to their losses as no funds can be left for retrieval.

Proof of Concept

3 types of mint/burn: NFT, Fungible and Synths. Synths are most vilnerable as they do not have share: LP own the pool, so Synth's funds are lost in scenarios similar to:

1. LP deposit both sides to a pool
2. Synth deposit and mint a Synth
3. LP withdraws all as she owns all the pool liquidity, even when provided only part of it
4. Synth can't withdraw as no assets left

burn NFT LP: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/pool/BasePoolV2.sol#L270

burn fungible LP: https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex-v2/pool/VaderPoolV2.sol#L374

Recommended Mitigation Steps

Take into account liquidity that was provided by Synth minting.

SamSteinGG (Vader) confirmed

[H-34] Incorrect Accrual Of `sumNative` and `sumUSD` In Producing Consultation Results

Submitted by leastwood

Impact

The `TwapOracle.consult()` function iterates over all token pairs which belong to either `VADER` or USDV`and then calculates the price of the respective asset by using both UniswapV2 and Chainlink price data. This helps to further protect against price manipulation attacks as the price is averaged out over the various registered token pairs.

Let's say we wanted to query the price of `USDV`, we would sum up any token pair where `USDV == pairData.token0`.

The sum consists of the following:

- Price of `USDV` denominated in terms of `token1` (`USDV/token1`).
- Price of token1 denominated in terms of `USD` (`token1/USD`).

Consider the following example:

- `SUSHI` and `UNISWAP` are the only registered token pairs that exist alongside `USDV`.
- Hence, calculating `sumNative` gives us an exchange rate that is denominated as the sum of `USDV/SUSHI` and `USDV/UNISWAP`.
- Similarly, `sumUSD` gives us the following denominated pairs, `SUSHI/USD` and `UNISWAP/USD`.
- Summing `sumUSD` and `sumNative` produces an entirely incorrect result as compared to multiplying the two results first and then summing.
- The issue is equivalent to the same issue as performing `(p1 + p2)*(q1 + q2)` as compared to `(p1*q1 + p2*q2)`. Obviously, these two results are not equivalent, however, the `consult()` function treats them as such.
- If we multiply the native price and Chainlink oracle results, then we can correctly calculate the price as such; `(SUSHI/USD * USDV/SUSHI + UNISWAP/USD * USDV/UNISWAP) / 2`, which should correctly give us the correct denomination and average price.

However, the protocol calculates it as `((SUSHI/USD + UNISWAP/USD) * token.decimals()) / (USDV/SUSHI + USDV/UNISWAP)` which gives us an incorrectly denominated result.

I'd classify this issue as high risk as the oracle returns false results upon being consulted. This can lead to issues in other areas of the protocol that use this data in performing sensitive actions.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L115-L157

Similar working implementation listed below:

- https://github.com/gg2001/dpx-oracle/blob/master/contracts/UniswapV2Oracle.sol#L184-L211
- https://github.com/gg2001/dpx-oracle/blob/master/contracts/UniswapV2Oracle.sol#L291-L304

Tools Used

Manual code review.

Recommended Mitigation Steps

To calculate the correct consultation of a given token, the returned result should consist of a sum of `priceUSD * token.decimals() * priceNative` divided by the number of calculations. This should correctly take the average token pair price.

The following snippet of code details the relevant fix:

```
    function consult(address token) public view returns (uint256
        uint256 pairCount = _pairs.length;

        for (uint256 i = 0; i < pairCount; i++) {
            PairData memory pairData = _pairs[i];

            if (token == pairData.token0) {
                //
                // TODO - Review:
                //   Verify price1Average is amount of USDV agai
                //

                priceNative = pairData.price1Average.mul(1).decc
                if (pairData.price1Average._x != 0) {
                    require(priceNative != 0);
                } else {
                    continue; // should skip newly registered as
                }

                (
                    uint80 roundID,
                    int256 price,
                    ,
                    ,
                    uint80 answeredInRound
                ) = AggregatorV3Interface(_aggregators[pairData.
```

```
                        .latestRoundData();

                require(
                    answeredInRound >= roundID,
                    "TwapOracle::consult: stale chainlink price'
                );
                require(
                    price != 0,
                    "TwapOracle::consult: chainlink malfunction'
                );
                priceUSD = uint256(price) * (10**10);
                result += ((priceUSD * IERC20Metadata(token).dec
            }
        }
        require(sumNative != 0, "TwapOracle::consult: Sum of nat
        return result;
    }
```

[SamSteinGG (Vader) confirmed](#)

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

Medium Risk Findings (23)

[M-01] Unbounded loop in TwapOracle.update can result in oracle being locked

Submitted by TomFrenchBlockchain, also found by pauliax

Impact

Loss of ability of `TwapOracle` to update should too many pools be added.

Proof of Concept

`TwapOracle` allows an unlimited number of pairs to be added and has no way of removing pairs after the fact. At the same time `TwapOracle.update` iterates through all pairs in order to update value for each pair.

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/twap/TwapOracle.sol#L322-L369

`TwapOracle.registerPair` is a permissioned function so that only the owner can add new pairs however should the owner account be compromised or not mindful of the number of pairs being added it is possible to put the oracle into a state in which it is unable to update. The oracle cannot recover from this state.

Recommended Mitigation Steps

Possible options:

- Add a method to stop tracking a particular pair
- Allow updating a subset of all pairs at a time.

[SamSteinGG (Vader) disagreed with severity]():

> The severity of the finding should be reduced as it relies on ill behavior from its owner which is a multisignature address.

[alcueca (judge) commented]():

> The severity rating is valid since the signers might not be aware of the limitation, or the limitation might be reached by natural means.

[SamSteinGG (Vader) commented]():

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-02] Should a Chainlink aggregator become stuck in a stale state then TwapOracle will become irrecoverably broken

Submitted by TomFrenchBlockchain

Impact

Inability to call `consult` on the TwapOracle and so calculate the exchange rate between USDV and VADER.

Proof of Concept

Should any of the Chainlink aggregators used by the TwapOracle becomes stuck in such a state that the check on L143-146 of `TwapOracle.sol` consistently fails (through a botched upgrade, etc.) then the `consult` function will always revert.

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/twap/TwapOracle.sol#L143-L146

There is no method to update the address of the aggregator to use so the `TwapOracle` will be irrecoverable.

Recommended Mitigation Steps

Allow governance to update the aggregator for a pair (ideally with a timelock.)

SamSteinGG (Vader) diagreed with severity:

> The scenario of a Chainlink oracle ceasing function is very unlikely and would cause widespread issues in the DeFi space as a whole.

alcueca (judge) commented:

> I'm going to maintain the severity 2 rating despite the low probability of a Chainlink aggregator being permanently disabled. The risk exists, and in general third-party dependencies should be treated with respect in code and documentation.

SamSteinGG (Vader) commented:

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-03] Permissioned nature of `TwapOracle` allows owner to manipulate oracle

Submitted by TomFrenchBlockchain

Impact

Potentially frozen or purposefully inaccurate USDV:VADER price feed.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/3a43059e33d549f03b021d6b417b7eeba66cf62e/contracts/twap/TwapOracle.sol#L322

Only the owner of `TwapOracle` can call `update` on the oracle. Should the owner desire they could cease calling `update` on the oracle for a period. Over this period the relative prices of VADER and USDC will vary.

After some period `timeElapsed` the owner can call `update` again. A TWAP is a lagging indicator and due to the owner ceasing to update the oracle so `timeElapsed` will be very large, therefore we're averaging over a long period into the past resulting in a value which may not be representative of the current USDV:VADER exchange rate.

The owner can therefore selectively update the oracle so to result in prices which allow them to extract value from the system.

Recommended Mitigation Steps

Remove the permissioning from `TwapOracle.update`

SamSteinGG (Vader) marked as duplicate

alcueca commented:

> Duplicate of which other issue, @SamSteinGG?

SamSteinGG (Vader) commented:

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-04] Inconsistent balance when supplying transfer-on-fee or deflationary tokens

Submitted by Reigada

Impact

In the contract StakingRewards, the stake function assume that the amount of stakingToken is transferred to the smart contract after calling the safeTransferFrom function (and thus it updates the `\_balances` mapping). However, this may not be true if the stakingToken is a transfer-on-fee token or a deflationary/rebasing token, causing the received amount to be less than the accounted amount in the `\_balances` mapping.

Same can be applied for the withdraw function.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/staking-rewards/StakingRewards.sol#L100-L102

Tools Used

Manual code review

Recommended Mitigation Steps

Get the actual received amount by calculating the difference of token balance before and after the transfer. For example: `uint256 balanceBefore = stakingToken.balanceOf(address(this)); stakingToken.safeTransferFrom(msg.sender, address(this), amount); uint256 receivedAmount = stakingToken.balanceOf(address(this)) - balanceBefore; _totalSupply = _totalSupply.add(receivedAmount); _balances\[msg.sender] = _balances\[msg.sender].add(receivedAmount);`

[0xstormtrooper (Vader) acknowledged]:

> VADER / USDV fee on transfer will be removed

[M-05] LinearVesting does not calculate vested amount linearly

Submitted by xYrYuYx

Impact

- [https://github.com/code-423n4/2021-11-vader/blob/main/contracts/tokens/vesting/LinearVesting.sol#L261]
- [https://github.com/code-423n4/2021-11-vader/blob/main/contracts/tokens/vesting/LinearVesting.sol#L294]

These calculations are incorrect for linear vesting.

Proof of Concept

i.e. if start amount is 10000, and duration is 100 seconds. After 50 seconds, user can claim 5000 which is 50%

After another 10 seconds, user need to claim 1000 which is 10%, but current calculation return 500.

Tools Used

Manual

Recommended Mitigation Steps

Change formula to `User total amount * (block.timestamp - start) / (vesting duration) - user claimed amount.`

[SamSteinGG (Vader) confirmed]

[M-06] add liquidity is vulnerable to sandwich attack

Submitted by jonah1005

add liquidity is vulnerable to MEV

Impact

`addLiquidity` in the VaderRouter and VaderRouterV2 contract does not check the minimum liquidity amount. This makes users' funds vulnerable to sandwich attacks.

The team says a minimum amount is not required as the VaderPool supports imbalanced mint. However, imbalanced mint is a helper function of buying tokens and providing to lp. A sandwich attack would take over more than 50% of a transaction in an illiquid pool.

Given the current network environment, most transactions in the mempool would be sandwiched. However, users may avoid this attack if they only send tx through flashbot RPC. I consider this is a medium-risk issue.

Proof of Concept

VaderRouterV2.sol#L77-L96

That says a user wants to provide 1M ETH in the pool. Attackers can sandwich this trade as follows:

1. Buy Vader with 10M ETH and makes ETH extremely cheap
2. *Put user's tx here* User's tx would first buy a lot Vader and deposit to the pool.
3. Since ETH becomes even cheaper in the pool. The MEV attacker buyback ETH and get profit.

Tools Used

None

Recommended Mitigation Steps

Always check how much liquidity a user received in a transaction. A tx would not be sandwiched if it's not profitable.

We could learn a lot about MEV from Robert Miller'tweets.

[SamSteinGG (Vader) disputed]():

> The design of the Thorchain CLP model is meant to prevent flash-loan based attacks as it allows a maximum trade size of 25% on a given iteration with a high-enough fee to render the attack unprofitable. Please request a tangible test case from the warden to consider this exhibit valid.

[alcueca (judge) commented]():

> There is no documentation stating that the deployment of Vader is restricted to Thorchain. The issue is valid.

[SamSteinGG (Vader) commented]():

> @alcueca The model itself is what has this trait, it does not relate to the blockchain implementation itself. It is intended functionality as with its parent implementation.

[M-07] Missing hasStarted modifier, can lead to user vesting before the owner begin the vesting

Submitted by rfa

Impact

In the `claimConverted()` function, the user can vest their vader token for a certain amount of time, but `hasStarted` modifier is missing, this can lead to `claimConverted()` function is callable by anyone, and the user can claim eventhough the vesting havent been started by the owner.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/contracts/tokens/vesting/LinearVesting.sol#L158

Recommended Mitigation Steps

add hasStarted modifier

[SamSteinGG (Vader) commented]():

> Duplicate of #89

alcueca (judge) commented:

> Not a duplicate, different line.

SamSteinGG (Vader) commented:

> @alcueca This should be invalid.

[M-08] User may not receive the full amount of IL compensation

Submitted by jonah1005

Impact

The user would not get full IL compensation if there’s not enough funds in the reserve. VaderReserve.sol#L76-L91

VaderReserve.sol#L85

```
            uint256 actualAmount = _min(reserve(), amount);
```

While this is reasonable, users should be able to specify the minimum received amount in the transaction. Otherwise, it’s vulnerable to some kind of MEV attack.

I consider this is a medium-risk issue.

Proof of Concept

The user can not specify a minimum IL compensation in the router. VaderRouter.sol#L169-L207

The user may not receive the full amount of compensation. VaderReserve.sol#L76-L91

Tools Used

None

Recommended Mitigation Steps

Users should be able to protect themselves when burning lp.

Some possible fixes:

1. Return the actual amount this line `reserve.reimburseImpermanentLoss(msg.sender, coveredloss);`
2. Checks whether there’s slippage. Revert if the user doesn’t receive the full amount.

We can add a new parameter in the function.

```
        require(actualCompensation > minimumCompensation);
```

Or we can check the `amountAMin` after receiving the compensation. (assume tokenA is native token)

```
        require(amountA +actualCompensation > amountAMin);
```

SamSteinGG (Vader) disputed:

> This is intended functionality of the protocol to account for rounding errors and is a principle similar to SushiSwap’s MasterChef contract.

alcueca (judge) commented:

> The sponsor acknowledges the issue.

SamSteinGG (Vader) commented:

> @alcueca They are never meant however it should still be possible to do so. With Uniswap V2, it is possible to interact with the contracts directly however doing so (without using a smart contract) will result in the same vulnerabilities as described

> in the issue. This is not intended usage and as such does not constitute an issue unless it is implied that the Uniswap V2 implementation is incorrect. Intended functionality of the protocol cannot constitute a risk issue. This has been classified as a medium risk issue but it follows the standardized conventions of implementations like Sushiswap which are live.

[M-09] The first lp provider can destroy the pool

Submitted by jonah1005

Impact

First lp provider received liquidity amount same as the nativeDeposit amount and decides the rate. If the first lp sets the pool's rate to an extreme value no one can deposit to the pool afterward. (please refer to the proof of concept section)

A malicious attacker can DOS the system by back-running the `setTokenSupport` and setting the pools' price to the extreme. I consider this is a medium-risk issue.

Proof of Concept

```
        deposit_amount = 1000 * 10**18
        get_token(dai, user, deposit_amount*3)
        get_token(vader, user, deposit_amount*3)
        dai.functions.approve(pool.address, deposit_amount*3).transa
        link.functions.approve(pool.address, deposit_amount*3).trans


        # deposit_amount = 1000 * 10**18

        # # first deposit 1 wei Dai and 1 vader to the pool
        router.functions.addLiquidity(dai.address, vader.address, 1,
        print('received liquidity', pool.functions.positions(0).call
        # output log:
        # 1000000000000000000

        # normally deposit to the pool
        router.functions.addLiquidity(dai.address, vader.address, de
        print('received liquidity', pool.functions.positions(1).call

        # output log:
```

```
        # 5000000000000000005000000000000000000000

        # no one can deposit to the pool now
        # there would be revert

        router.functions.addLiquidity(dai.address, vader.address, 1,
```

[VaderMath.sol#L42]

```
        ((totalPoolUnits * poolUnitFactor) / denominator) * slip
```

Since the scale of the total supply is (10**18)^2, the operation would overflow.

Tools Used

None

Recommended Mitigation Steps

Set a minimum deposit amount (both asset amount and native amount) for the first lp provider.

[SamSteinGG (Vader) confirmed]

[M-10] SHOULD CHECK RETURN DATA FROM CHAINLINK AGGREGATORS

Submitted by defsec

Impact

The consult function in the contract TwapOracle.sol fetches the asset price from a Chainlink aggregator using the latestRoundData function. However, there are no checks on timeStamp, resulting in stale prices. The oracle wrapper calls out to a chainlink oracle receiving the latestRoundData(). It then checks freshness by verifying that the answer is indeed for the last known round. The returned updatedAt timestamp is not checked.

If there is a problem with chainlink starting a new round and finding consensus on the new value for the oracle (e.g. chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the chainlink system) consumers of this contract may continue using outdated stale data (if oracles are unable to submit no new round is started)

Proof of Concept

1. Navigate to "https://github.com/code-423n4/2021-11-vader/blob/607d2b9e253d59c782e921bfc2951184d3f65825/contracts/twap/TwapOracle.sol#L141" contract.
2. consult function does not check timestamp on the latestRoundData.

Tools Used

None

Recommended Mitigation Steps

Consider to add checks on the return data with proper revert messages if the price is stale or the round is incomplete, for example:

```
(uint80 roundID, int256 price, , uint256 timeStamp, uint80 answe
require(timeStamp != 0, "...");
```

Consider checking the oracle responses updatedAt value after calling out to chainlinkOracle.latestRoundData() verifying that the result is within an allowed margin of freshness.

- https://docs.chain.link/docs/faq/#how-can-i-check-if-the-answer-to-a-round-is-being-carried-over-from-a-previous-round
- https://blog.openzeppelin.com/secure-smart-contract-guidelines-the-dangers-of-price-oracles/

SamSteinGG (Vader) confirmed

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-11] TWAP Oracle inflexible `_updatePeriod`

Submitted by elprofesor

Impact

Update periods in TWAP oracles reflect risk of an asset. Updating more frequently accurately prices an asset but increases capabilities of manipulation (which should be harder with more stable assets), whereas longer update periods prevent manipulation but does not accurately price assets (due to the time difference between updates). Volatility of an asset should be considered when calculating update periods. However, in Vader's `TwapOracle.sol` no such considerations are made, the `_updatePeriod` cannot be changed after deployment of the contract. Additionally, each asset uses the same `_updatePeriod` which does not adequately account for the differences in risk for each asset. This could lead to price manipulation or inadequate pricing of assets.

Proof of Concept

[the only chance to set `_updatePeriod`]

[`_updatePeriod` used in time elapsed calculation]

Recommended Mitigation Steps

Add the following function

```
    function setUpdatePeriod(uint256 newUpdatePeriod) external only(
        require(newUpdatePeriod > minimumUpdatePeriod, "Update peric
        _updatePeriod = newUpdatePeriod;
    }
```

[CloudEllie (organizer) commented]:

> Warden has requested that we add a second mitigation step/strategy to his submission:

2. Consider adding a configurable update period per asset, this way we are not incorrectly assuming the same risk profile per asset.

[SamSteinGG (Vader) diagreed with severity](#):

> This finding is accurate as the update period should change per oracle, however, it is not of high risk severity.

[alcueca (judge) commented](#):

> Agree with sponsor, assets are not directly at risk. Severity 2.

[SamSteinGG (Vader) commented](#):

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-12] Missing duplicate veto check

Submitted by defsec

Impact

On the GovernorAlpha contract, function veto has been added. Although the function behaviour is expected, duplicate veto process has not been checked on that function.

Proof of Concept

1. Navigate to following contract line. (https://github.com/code-423n4/2021-11-vader/blob/607d2b9e253d59c782e921bfc2951184d3f65825/contracts/governance/GovernorAlpha.sol#L562)

```
    function veto(uint256 proposalId, bool support) external onl
        ProposalState _state = state(proposalId);
        require(
            _state == ProposalState.Active || _state == Proposal
            "GovernorAlpha::veto: Proposal can only be vetoed wh
        );

        Proposal storage proposal = proposals[proposalId];
        address[] memory _targets = proposal.targets;
        for (uint256 i = 0; i < _targets.length; i++) {
            if (_targets[i] == address(this)) {
                revert(
                    "GovernorAlpha::veto: council cannot veto or
                );
            }
        }

        VetoStatus storage _vetoStatus = proposal.vetoStatus;
        _vetoStatus.hasBeenVetoed = true;
        _vetoStatus.support = support;

        if (support) {
            queue(proposalId);
        }

        emit ProposalVetoed(proposalId, support);
    }

    /**
```

2. The veto progress can be completed per proposal twice.

Tools Used

None

Recommended Mitigation Steps

Consider to check if proposals vetoed before.

```
VetoStatus storage _vetoStatus = proposal.vetoStatus;
require(!_vetoStatus.hasBeenVetoed, "Vetoed before");
```

SamSteinGG (Vader) commented:

> Duplicate of #61

alcueca (judge) commented:

> Not a duplicate

[M-13] `BasePool.mint()` Is Callable By Anyone

Submitted by leastwood

Impact

The `BasePool.mint()` function differs from its implementation in `BasePoolV2.mint()` in which it lacks an `onlyRouter` modifier. This ensures that users cannot call this function directly as `VaderRouter.addLiquidity()` performs some necessary input validation which can be bypassed by directly calling `BasePool.mint()`.

Proof of Concept

- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/router/VaderRouter.sol#L123-L150
- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/pool/BasePool.sol#L149-L194

Tools Used

Manual code review.

Recommended Mitigation Steps

Consider adding an `onlyRouter` modifier to the `BasePool.mint()` function to ensure users cannot directly call this function.

SamSteinGG (Vader) disputed:

> The pool contracts, similarly to Uniswap V2, are never meant to be interacted with directly.

alcueca (judge) commented:

> That’s what the modifier would do.

SamSteinGG (Vader) commented:

> They are never meant however it should still be possible to do so. With Uniswap V2, it is possible to interact with the contracts directly however doing so (without using a smart contract) will result in the same vulnerabilities as described in the issue. This is not intended usage and as such does not constitute an issue unless it is implied that the Uniswap V2 implementation is incorrect.

[M-14] `BasePool.swap()` Is Callable By Anyone

Submitted by leastwood

Impact

The `BasePool.swap()` function differs from its implementation in `BasePoolV2.swap()` in which it lacks an `onlyRouter` modifier. This ensures that users cannot call this function directly as `VaderRouter._swap()` performs some necessary input validation which can be bypassed by directly calling `BasePool.swap()`.

Proof of Concept

- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/router/VaderRouter.sol#L304-L351
- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/pool/BasePool.sol#L289-L379
- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/pool/BasePool.sol#L261-L268

Tools Used

Manual code review.

Recommended Mitigation Steps

Consider adding an `onlyRouter` modifier to the `BasePool.swap()` functions to ensure users cannot directly call these functions.

SamSteinGG (Vader) disputed:

> The pool contracts, similarly to Uniswap V2, are never meant to be interacted with directly.

alcueca (judge) commented:

> That's what the modifier would do.

SamSteinGG (Vader) commented:

> @alcueca They are never meant however it should still be possible to do so. With Uniswap V2, it is possible to interact with the contracts directly however doing so (without using a smart contract) will result in the same vulnerabilities as described in the issue. This is not intended usage and as such does not constitute an issue unless it is implied that the Uniswap V2 implementation is incorrect.

[M-15] Lacking Validation Of Chainlink' Oracle Queries

Submitted by leastwood

Impact

`TwapOracle.consult()` is missing additional validations to ensure that the round is complete and has returned a valid/expected price. The `consult()` improperly casts an `int256 price` to `uint256` without first checking the value. As a result, the variable may underflow and return an unexpected result, potentially causing further issues in other areas of the protocol that rely on this function.

Additionally, the `GasThrottle.validateGas()` modifier utilises Chainlink's `latestAnswer()` function which lacks additional checks for stale data. The `latestRoundData()` function facilitates additional checks and should be used over `latestAnswer()`.

Proof of Concept

- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/twap/TwapOracle.sol#L134-L150
- https://github.com/code-423n4/2021-11-vader/blob/main/contracts/dex/utils/GasThrottle.sol#L15

- https://docs.chain.link/docs/faq/#how-can-i-check-if-the-answer-to-a-round-is-being-carried-over-from-a-previous-round

Tools Used

Manual code review. Chainlink best practices.

Recommended Mitigation Steps

Consider validating the output of `latestRoundData()` to match the following code snippet:

```
        (
          uint80 roundID,
          int256 price,
          ,
          uint256 updateTime,
          uint80 answeredInRound
        ) = ETH_CHAINLINK.latestRoundData();
        require(
            answeredInRound >= roundID,
            "Chainlink Price Stale"
        );
        require(price > 0, "Chainlink Malfunction");
        require(updateTime != 0, "Incomplete round");
```

This needs to be updated in `TwapOracle.consult()` and in `GasThrottle.validateGas()`. The latter instance should have the `latestAnswer()` function replaced with `latestRoundData()` in order to avoid stale data.

[SamSteinGG (Vader) confirmed](#)

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-16] Governor’s veto protection can be exploited

Submitted by cmichel

The `GovernorAlpha`'s council cannot veto proposals that perform a call to the contract itself. This can be exploited by malicious proposal creators by appending a new call at the end of their proposal that simply calls an innocent function like `GovernorAlpha.votingDelay()`.

Impact

The veto procedure can easily be circumvented, making the council unable to veto.

Recommended Mitigation Steps

The veto check must be further restricted by specifying the actual function selector that is not allowed to be vetoed, like `changeCouncil`.

SamSteinGG (Vader) commented:

> Duplicate of #61

alcueca (judge) commented:

> Not a duplicate

[M-17] Vests can be denied

Submitted by cmichel

The `LinearVesting.vestFor` function (which is called by `Converter`) reverts if there already exists a vest for the user:

```
    require(
        vest[user].amount == 0,
        "LinearVesting::selfVest: Already a vester"
    );
```

There's an attack where a griefer frontruns the `vestFor` call and instead vests the smallest unit of VADER for the `user`. The original transaction will then revert and the vest will be denied

Recommended Mitigation Steps

There are several ways to mitigate this. The most involved one would be to allow several separate vestings per user.

[SamSteinGG (Vader) confirmed)](#)

[M-18] `TWAPOracle.getRate` does not scale the ratio

Submitted by cmichel

The `TWAPOracle.getRate` function simply performs an integer division to compute the rate.

```
    function getRate() public view returns (uint256 result) {
        uint256 tUSDInUSDV = consult(USDV);
        uint256 tUSDInVader = consult(VADER);
        // @audit shouldn't this scale by 1e18 first? otherwise easi
        result = tUSDInUSDV / tUSDInVader;
    }
```

It should first be scaled by some value, for example, `1e18`.

Impact

The rate has no decimal precision and if `tUSDInVader > tUSDInUSDV`, the rate will always be zero.

The `usdvtoVader` and `vaderToUsdv` functions will return incorrect values.

Recommended Mitigation Steps

```
    // return as a rate with 18 decimals instead
    result = tUSDInUSDV * 1e18 / tUSDInVader;
```

[SamSteinGG (Vader) confirmed](#)

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-19] Unclear `TwapOracle.consult` algorithm

Submitted by cmichel

The `TWAPOracle.consult` function is unclear to the auditor. It seems to iterate through all registered pairs that share the `token` parameter (USDV or VADER) and then sums up the foreign token pair per `token` price. And divides this sum (`sumNative`) by the summed-up USD price of these foreign token pairs (`sumUSD`).

I think the idea is to create some kind of average price but doing it like this does not seem to be effective because large prices are weighted a lot stronger than low prices.

Example

Assume there are 3 USDV pairs registered: `(ETH, DAI, USDC)` .

Oracle Price: USDV/ETH 4500, USDV/DAI 1, USDV/USDC 1 Pool price: USDV/ETH 4500, USDV/DAI 10, USDV/USDC 10

Even though the DAI and USDC pool prices are off by 10x, the final result is `4502/4520 = 0.996017699` very close to a price of `1.0` which seems strange.

Recommended Mitigation Steps

Document how the algorithm works and make sure it's correct. Resolve the `TODO` .

SamSteinGG (Vader) confirmed

> The TWAP oracle module has been completely removed and redesigned from scratch as LBTwap that is subject of the new audit.

[M-20] Tokens with fee on transfer are not supported

Submitted by WatchPug

There are ERC20 tokens that charge fee for every `transfer()` or `transferFrom()` , E.g `Vader` token.

In the current implementation, `BasePoolV2.sol#mint()` assumes that the received amount is the same as the transfer amount, and uses it to calculate liquidity units.

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/BasePoolV2.sol#L168-L229

```
    function mint(
        IERC20 foreignAsset,
        uint256 nativeDeposit,
        uint256 foreignDeposit,
        address from,
        address to
    )
        external
        override
        nonReentrant
        onlyRouter
        supportedToken(foreignAsset)
        returns (uint256 liquidity)
    {
        (uint112 reserveNative, uint112 reserveForeign, ) = getReser
            foreignAsset
        ); // gas savings

        nativeAsset.safeTransferFrom(from, address(this), nativeDepo
        foreignAsset.safeTransferFrom(from, address(this), foreignDe

        PairInfo storage pair = pairInfo[foreignAsset];
        uint256 totalLiquidityUnits = pair.totalSupply;
        if (totalLiquidityUnits == 0) liquidity = nativeDeposit;
        else
            liquidity = VaderMath.calculateLiquidityUnits(
                nativeDeposit,
                reserveNative,
                foreignDeposit,
                reserveForeign,
                totalLiquidityUnits
            );
        require(
            liquidity > 0,
            "BasePoolV2::mint: Insufficient Liquidity Provided"
        );

        uint256 id = positionId++;

        pair.totalSupply = totalLiquidityUnits + liquidity;
        _mint(to, id);

        positions[id] = Position(
            foreignAsset,
            block.timestamp,
            liquidity,
            nativeDeposit,
            foreignDeposit
        );

        _update(
            foreignAsset,
            reserveNative + nativeDeposit,
            reserveForeign + foreignDeposit,
            reserveNative,
            reserveForeign
        );

        emit Mint(from, to, nativeDeposit, foreignDeposit);
        emit PositionOpened(from, to, id, liquidity);
    }
```

Recommended

Consider calling `balanceOf()` to get the actual balances.

SamSteinGG (Vader) disagreed with severity:

> Tokens with a fee on transfer or rebasing tokens are not meant to be supported by the protocol, hence why the tokens supported are voted on by the DAO.

alcueca (judge) commented:

> Not stated in the documentation, therefore the issue is valid.

SamSteinGG (Vader) commented:

> A medium risk issue cannot constitute lack of documentation of a trait of the system. If the inclusion of tokens to the DEX was not privileged, the issue would be valid but it is not an open function and thus the scenario described would never unfold.

[M-21] VaderPoolV2.rescue results in loss of funds rather than recoverability

Submitted by TomFrenchBlockchain

Impact

Any unaccounted for tokens on `VaderPoolV2` can be siphoned off by anyone

Proof of Concept

`VaderPoolV2` has a `rescue` function which allows any unaccounted for tokens to be recovered.

https://github.com/code-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex-v2/pool/BasePoolV2.sol#L505-L517

However there is no access control on this function which means than should any tokens be sent to `VaderPoolV2` by accident they'll just be scooped up by flashbots rather than being recoverable by the original owner or Vader governance.

This also means that any rebasing tokens which are deposited into `VaderPoolV2` will have any rebases lost rather than being recoverable by Vader governance.

Recommended Mitigation Steps

Permission this function to only allow Vader governance to claim tokens.

SamSteinGG (Vader) commented:

> Duplicate #28

alcueca (judge) commented:

> Not a duplicate, this issue correctly states that the function is vulnerable to front-running.

SamSteinGG (Vader) commented:

> The function is equivalent to the Uniswap V2 rescue function which is not classified as incorrect.

[M-22] No way to remove GasThrottle after deployment

Submitted by TomFrenchBlockchain

Impact

Potential DOS on swaps

Proof of Concept

BasePool and BasePoolV2 make 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-423n4/2021-11-vader/blob/429970427b4dc65e37808d7116b9de27e395ce0c/contracts/dex/utils/GasThrottle.sol#L9-L20

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

SamSteinGG (Vader) confirmed

[M-23] Users Can Reset Bond Depositor's Vesting Period

Submitted by leastwood

Impact

The `VaderBond.deposit()` function overwrites a depositors bond info on each call with the updated `payout` information. If any of the vesting is left unclaimed before a call to `deposit()` is made, the vesting period is reset to `terms.vestingTerm`, resulting in the bond holder having to wait again in order to claim tokens that they could previously claim.

Proof of Concept

https://github.com/code-423n4/2021-11-vader/blob/main/repo/vader-bond/contracts/VaderBond.sol#L192

Tools Used

Manual code review.

Recommended Mitigation Steps

Consider preventing users from depositing to an existing bond holder or alternatively when a deposit is made, force the user to redeem any claimable tokens in the same function.

0xstormtrooper (Vader) acknowledged:

> Users will be warned that depositing resets the vesting term. This will also be documented on the contract.

Low Risk Findings (41)

- [L-01] *VADER*VETHER*CONVERSION*RATE *Submitted by jayjonah8, also found by hack3r-0m*
- [L-02] ReentrancyGuard for added protection *Submitted by jayjonah8*
- [L-03] TwapOracle assumes a possibly incorrect number of decimals when scaling chainlink price feed *Submitted by TomFrenchBlockchain, also found by pauliax and xYrYuYx*
- [L-04] _totalSupply can be different from actual supply *Submitted by jayjonah8*

- [L-05] Use `_safeMint()` instead of `_mint()` *Submitted by Ruhum, also found by jayjonah8*
- [L-06] Function LinearVesting.claim() will never meet require conditions *Submitted by Reigada*
- [L-07] Did not check if vestor is address(0) *Submitted by xYrYuYx*
- [L-08] TwapOracle / registerPair function could register VADER / USDV and USDV / VADER pools. *Submitted by xYrYuYx*
- [L-09] Missing events for owner only functions that change critical parameters *Submitted by defsec*
- [L-10] Governance Veto lacks sufficient validation to protect against frontrunning *Submitted by elprofesor*
- [L-11] Missing zero-address checks in multiple constructors *Submitted by Reigada, also found by cmichel and Meta0xNull*
- [L-12] Missing balance check before and after transfer, can lead to inconsistent amount due to fee on transfer *Submitted by rfa*
- [L-13] token allocation specs in contract code does not match with whitepaper *Submitted by hack3r-0m*
- [L-14] Open TODOs *Submitted by yeOlde, also found by defsec, Meta0xNull, pants, and pauliax*
- [L-15] No boundary checking for feeAmount (GovernorAlpha.sol) *Submitted by yeOlde*
- [L-16] createEmission function specification and logic mismatch *Submitted by hack3r-0m*
- [L-17] Incompatibility With Rebasing/Deflationary/Inflationary tokens *Submitted by defsec*
- [L-18] Add zero address validation in the GovernorAlpha contract *Submitted by defsec*
- [L-19] Incorrect comments (technical issues) *Submitted by yeOlde*
- [L-20] Function AdjustMaxSupply is incorrect (or at least confusing) *Submitted by yeOlde*
- [L-21] `Converter::constructor` ignores return value from function call *Submitted by pmerkleplant*

- [L-22] `Converter.convert()` Proofs Can Be Replayed On Other Chains *Submitted by leastwood*
- [L-23] setRewardsDuration() Lack of Input Validation May Break notifyRewardAmount() *Submitted by MetaOxNull*
- [L-24] Tokens can be unsupported again *Submitted by cmichel*
- [L-25] Governor average block time is not up-to-date *Submitted by cmichel*
- [L-26] Using duplicate vesters tracks wrong total amount *Submitted by cmichel*
- [L-27] No Transfer Ownership Pattern *Submitted by defsec*
- [L-28] block.chainid may change in case of a hardfork *Submitted by defsec*
- [L-29] Wrong comment in `vaderToUsdv` *Submitted by gzeon*
- [L-30] `VaderRouterV2#addLiquidity()` is not compatible with the interface of UniswapV2Router02#addliquidity() *Submitted by WatchPug*
- [L-31] `BasePoolV2#rescue()` should be `nonReentrant` *Submitted by WatchPug*
- [L-32] Unsafe type casting *Submitted by WatchPug*
- [L-33] Possibility of reducing the maxSupply of Vader *Submitted by ksk2345*
- [L-34] `Router#initialize()` Lack of input validation for `reserve` asset *Submitted by WatchPug*
- [L-35] Toggle function *Submitted by pauliax*
- [L-36] Contracts VaderPoolFactory and VaderReserve can be initialized multiple times *Submitted by pauliax*
- [L-37] block times 13s -> 12s *Submitted by pauliax*
- [L-38] Unique vesters *Submitted by pauliax*
- [L-39] setComponents function specs and logic mismatch *Submitted by hack3r-0m*
- [L-40] Owner can maliciously set itself as dao *Submitted by hack3r-0m*
- [L-41] inconsistent use of msg.sender and _msgSender() *Submitted by hack3r-0m*

Non-Critical Findings (36)

- [N-01] BasePool does not account for Vader transfer fees when removing liquidity *Submitted by TomFrenchBlockchain*
- [N-02] VaderRouterV2 breaks compatibility with IUniswapV2Router0X *Submitted by TomFrenchBlockchain*
- [N-03] VaderRouter breaks compatibility with IUniswapV2Router0X *Submitted by TomFrenchBlockchain*
- [N-04] Use `indexed` keyword in events which can be used as filter *Submitted by xYrYuYx*
- [N-05] Typo *Submitted by xYrYuYx*
- [N-06] Missing Zero-address check *Submitted by xYrYuYx*
- [N-07] `onlyOnwer` in the synthFactory is confusing *Submitted by jonah1005*
- [N-08] Multiple Solidity pragma *Submitted by fatimanaz_*
- [N-09] Multiple Solidity pragma in repo/vader-bond/contracts/interfaces/ITreasury.sol *Submitted by fatimanaz_*
- [N-10] Multiple Solidity pragma in repo/vader-bond/contracts/test/TestToken.sol *Submitted by fatimanaz_*
- [N-11] Multiple Solidity pragma repo/vader-bond/contracts/lib/FullMath.sol *Submitted by fatimanaz_*
- [N-12] Multiple Solidity pragma In repo/vader-bond/contracts/Ownable.sol *Submitted by fatimanaz_*
- [N-13] Multiple Solidity pragma repo/vader-bond/contracts/lib/FixedPoint.sol *Submitted by fatimanaz_*
- [N-14] Multiple Solidity pragma in repo/vader-bond/contracts/Treasury.sol *Submitted by fatimanaz_*
- [N-15] Multiple Solidity pragma in repo/vader-bond/contracts/VaderBond.sol *Submitted by fatimanaz_*
- [N-16] Incorrect + misleading documentation on BasePool and BasePoolV2 *Submitted by TomFrenchBlockchain*
- [N-17] Wrong revert message in the router (UniswapV2Router -> VaderRouter) *Submitted by jonah1005*
- [N-18] No event emission for “timelock” changes (GovernorAlpha.sol) *Submitted by yeOlde*

- [N-19] No event emission for “guardian” changes (GovernorAlpha.sol) *Submitted by yeOlde*
- [N-20] Redundant Gas Modifider *Submitted by defsec, also found by jonah1005 and WatchPug*
- [N-21] might not check current block when casting vote *Submitted by pants*
- [N-22] Commented out code *Submitted by yeOlde*
- [N-23] Typos *Submitted by yeOlde*
- [N-24] Use safeTransfer instead of transfer *Submitted by defsec, also found by elprofesor and pants*
- [N-25] VaderBond insufficient validation of max payout may prevent redeeming valid payout *Submitted by elprofesor*
- [N-26] `USDV.sol` Incomplete implementation *Submitted by WatchPug*
- [N-27] `SwapQueue.sol` Incomplete implementation *Submitted by WatchPug*
- [N-28] Missing events for critical operations *Submitted by WatchPug*
- [N-29] Critical changes should use two-step procedure *Submitted by WatchPug*
- [N-30] Disregarding Check Effects in `VaderBond.redeem()` *Submitted by elprofesor*
- [N-31] `LinearVesting` missing events *Submitted by elprofesor*
- [N-32] Unsupported tokens can be given fungible LP support *Submitted by TomFrenchBlockchain*
- [N-33] Abstract contracts *Submitted by pauliax*
- [N-34] Small precision loss when dividing *Submitted by pauliax*
- [N-35] safe transfer of tokens *Submitted by pauliax*
- [N-36] Whitepaper and code declares different conversion rates *Submitted by pauliax*

Gas Optimizations (56)

- [G-01] Make use of a bitmap for claims to save gas in Converter.sol *Submitted by TomFrenchBlockchain*

- [G-02] Use of SafeERC20 for known tokens used extra gas unnecessarily *Submitted by TomFrenchBlockchain*
- [G-03] Use of single _pairs array in TwapOracle increases costs of `consult` ing *Submitted by TomFrenchBlockchain*
- [G-04] Use proxy clones to create Synths & LPTokens *Submitted by nathaniel*
- [G-05] Change if -> revert pattern to require *Submitted by mics*
- [G-06] Address of VADER in xVADER.sol can be made immutable *Submitted by TomFrenchBlockchain, also found by xYrYuYx and pauliax*
- [G-07] VADER and USDV can be made immutable in the TwapOracle *Submitted by TomFrenchBlockchain*
- [G-08] Cache length of array before loop to optimize gas *Submitted by xYrYuYx*
- [G-09] Use `unchecked` keyword to optimize gas *Submitted by xYrYuYx, also found by defsec, gzeon, and WatchPug*
- [G-10] Unused imported contract in xVader *Submitted by hack3r-0m*
- [G-11] Some public functions can be converted as external *Submitted by xYrYuYx*
- [G-12] Unnecessary validation of `proposalId>0` due to incorrect proposalId increments *Submitted by elprofesor*
- [G-13] Using SafeMath ins Solidity 0.8.9 contracts wastes gas *Submitted by 0xean*
- [G-14] Local variable assignment in XVader.leave function can be removed to save gas *Submitted by Reigada*
- [G-15] Contract StakingRewards (pragma 0.8.9) makes use of SafeMath *Submitted by Reigada, also found by 0x0x0x*
- [G-16] Using ++i consumes less gas than i++ *Submitted by Reigada*
- [G-17] No need to initialize uint256 i variable to 0 in for loops *Submitted by Reigada, also found by Meta0xNull, pants, and pauliax*
- [G-18] using memory pointer instead storage *Submitted by rfa*
- [G-19] using memory pointer instead storage *Submitted by rfa*
- [G-20] Save gas by caching array length used in for loops *Submitted by 0x0x0x, also found by WatchPug*

- [G-21] internal function _addLiquidity in the router is unnecessary *Submitted by jonah1005*
- [G-22] Unused Named Returns *Submitted by yeOlde*
- [G-23] Long Revert Strings *Submitted by yeOlde, also found by defsec, pants, pauliax, and WatchPug*
- [G-24] Assignment Of Variables To Default *Submitted by yeOlde*
- [G-25] Use existing memory version of state variables *Submitted by yeOlde, also found by cmichel*
- [G-26] Redundant Code Statement *Submitted by defsec*
- [G-27] Redundant Functions *Submitted by defsec*
- [G-28] Caching array length to save gas *Submitted by pants*
- [G-29] Unchecked{i++} is better than i++ *Submitted by pants*
- [G-30] Prefix increaments are cheaper than postfix increaments *Submitted by pants*
- [G-31] `public` functions can be `external` *Submitted by pants*
- [G-32] unessesary safe math in UniSwapV2Pair.sol line 120 *Submitted by pants*
- [G-33] calldata vs memory in solidity gas usage *Submitted by pants*
- [G-34] Zero address check not needed *Submitted by yeOlde*
- [G-35] `StakingRewards.sol#updateReward` can be split to two modifiers to save gas *Submitted by 0x0x0x*
- [G-36] Use constant `_INITIAL_EMISSION_CURVE` in `Vader.sol` *Submitted by pmerkleplant*
- [G-37] Use bytes32 Rather Than String *Submitted by Meta0xNull*
- [G-38] TwapOracle.sol update() Multiple SLOAD During Loop *Submitted by Meta0xNull*
- [G-39] Gas Optimization: Inline instead of modifier *Submitted by gzeon*
- [G-40] Gas Optimization: Simplify Math *Submitted by gzeon*
- [G-41] Unnecessary storage variables can be changed to `immutable` to save gas *Submitted by WatchPug*
- [G-42] Avoid unnecessary storage read can save gas *Submitted by WatchPug*

- [G-43] Changing function visibility from public to external can save gas *Submitted by WatchPug*
- [G-44] Combine external calls into one can save gas *Submitted by WatchPug*
- [G-45] Check that all transfers don't result in violation of max supply is unnecessary *Submitted by TomFrenchBlockchain*
- [G-46] Gas: VaderMath's endpoint functions can be made external *Submitted by hyh*
- [G-47] Store VaderPoolV2 address as immutable in LPWrapper *Submitted by TomFrenchBlockchain*
- [G-48] Add method to migrate from fungible to nonfungible liquidity *Submitted by TomFrenchBlockchain*
- [G-49] Pre-calculate values that do not change *Submitted by pauliax*
- [G-50] queueActive always false *Submitted by pauliax*
- [G-51] SafeMath with Solidity >0.8 *Submitted by pauliax*
- [G-52] Dead code *Submitted by pauliax*
- [G-53] Extra transfers when burning LP tokens *Submitted by TomFrenchBlockchain*
- [G-54] Unchecked math operations *Submitted by pauliax*
- [G-55] Unused variable lastEmission *Submitted by hack3r-0m*
- [G-56] Unused functionalities of inherited contracts *Submitted by hack3r-0m*

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