Vader Protocol
Findings & Analysis Report

2021-06-14

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

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

During the code contest outlined in this document, C4 conducted an analysis of the Vader Protocol smart contract system written in Solidity. The code contest took place between April 21 and April 28, 2021.

Wardens

10 Wardens contributed reports to the Vader code contest:

- cmichel
- rajeev
- shw
- pauliax (Thunder)
- jvaqa
- s1m0
- a_delamo
- toastedsteaksandwich
- mukesh jaiswal
- gpersoon

This contest was judged by LSDan.

Final report assembled by moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 121 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 24 received a risk rating in the category of HIGH severity, 16 received a risk rating in the category of MEDIUM severity, and 23 received a risk rating in the category of LOW severity.

C4 analysis also identified 60 non-critical recommendations including gas optimizations.

Scope

The code under review can be found within the C4 vader code contest repository and comprises 23 smart contracts written in the Solidity programming language.

Severity Criteria

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

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

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

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

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

High Risk Findings

[H-01] Unhandled return value of transfer in `transferOut()` of Pools.sol

ERC20 implementations are not always consistent. Some implementations of transfer and `transferFrom` could return 'false' on failure instead of reverting. It is safer to wrap such calls into `require()` statements to handle these failures.

The transfer call on L211 of `transferOut()` could be made on a user-supplied untrusted token address (from the different call sites) whose implementation can be malicious.

For reference, see similar finding from Consensys Diligence Audit of AAVE Protocol V2

Recommend requirements to check the return value and revert on 0/false or use OpenZeppelin's SafeERC20 wrapper functions.

strictly-scarce (vader) disputed:

> Not valid. Since the funds came in, and did not revert, they can leave. If the call passes, then the transferout is valid.

[H-02] Flash attack mitigation does not work as intended in USDV.sol

One of the stated protocol (review) goals is to detect susceptibility to "Any attack vectors using flash loans on Anchor price, synths or lending." As such, USDV contract aims to protect against flash attacks using `flashProof()` modifier which uses the following check in `isMature()` to determine if currently executing contract context is at least `blockDelay` duration ahead of the previous context:

```
lastBlock[tx.origin] + blockDelay <= block.number
```

However, `blockDelay` state variable is not initialized which means it has a default uint value of 0. So unless it is set to >= 1 by `setParams()` which can be called only by the DAO (which currently does not have the capability to call `setParams()` function), `blockDelay` will be 0, which allows current executing context (`block.number`) to be the same as the previous one (`lastBlock[tx.origin]`). This effectively allows multiple calls on this contract to be executed in the same transaction of a block which enables flash attacks as opposed to what is expected as commented on L41: "// Stops an EOA from doing a flash attack in the same block"

Even if the DAO can call `setParams()` to change `blockDelay` to >= 1, there is a big window of opportunity for flash attacks until the DAO votes, finalizes and approves such a proposal. Moreover, such proposals can be cancelled by a DAO minority or replaced by a malicious DAO minority to launch flash attacks.

Recommend initalizing `blockDelay` to >= 1 at declaration or in constructor.

strictly-scarce (vader) confirmed:

> The actual issue is simply:
>
> `blockDelay` state variable is not initialized
>
> It is intended to be initialised to 1, so this is a bug. Severity: 2

[H-03] Missing DAO functionality to call `changeDAO()` function in Vader.sol

`changeDAO()` is authorized to be called only from the DAO (per modifier) but DAO contract has no corresponding functionality to call `changeDAO()` function. As a result, DAO address cannot be changed (L192-L196).

Recommend adding functionality to DAO to be able to call `changeDAO()` of Vader.sol.

strictly-scarce (vader) commented:

> #46

dmvt (judge) commented:

> Unlike in issues #140, #157, #158, & #159; without this functionality, missing functionality in the DAO becomes a very serious issue. As a result, this one is very high risk were it to be overlooked.

[H-04] Proposals can be cancelled

Anyone can cancel any proposals by calling `DAO.cancelProposal(id, id)` with `oldProposalID == newProposalID`. This always passes the minority check as the proposal was approved.

An attacker can launch a denial of service attack on the DAO governance and prevent any proposals from being executed.

Recommend checking that `oldProposalID == newProposalID`

strictly-scarce (vader) confirmed:

> This is valid, can fix with a `require()`

strictly-scarce (vader) commented:

[H-05] Flash loans can affect governance voting in DAO.sol

Flash loans can significantly increase a single voter's weight and be used to impact the voting outcome. A voter can borrow a significant quantity of tokens to increase their voting weight in a transaction within which, they also deterministically influence the voting outcome to their choice.

This has already happened in the case of MakerDAO governance where a flash loan was used to affect voting outcome and noted by the Maker team as: "a practical example for the community that flash loans can and may impact system governance"

Given that flash loans are a noted concern, the impact of it to DAO governance which can control all critical protocol parameters should be mitigated as in other places.

Recommend accounting for flash loans in `countMemberVotes()` by using weight from previous blocks or consider capping the weight of individual voters. (L158-L163)

strictly-scarce (vader) disputed:

> Not valid. All pools use slip-based fees so flash loan attack by buying up USDV or synths is not going to work.

dmvt (judge) commented:

> The funds to execute this attack do not need to come from a pool. It could be done as simply as malicious members pooling their funds in a flash loan contract, and each borrowing the funds in turn to vote.

[H-06] Incorrect burn address in Vader.sol

The `internal _transfer()` function is called from external facing `transfer()`, `transferFrom()`, and `transferTo()` functions all of which have different sender addresses. It is `msg.sender` for `transfer()`, sender parameter for `transferFrom()` and `tx.origin` for `transferTo()`.

These different senders are reflected in the sender parameter of `_transfer()` function. While this sender parameter is correctly used for transfer of tokens within `_transfer`, the call to `_burn()` on L129 incorrectly uses `msg.sender` as the burn address which is correct only in the case of the `transfer()` caller's context. This is incorrect for `transferFrom()` and `transferTo()` caller contexts.

This will incorrectly burn the fees from a different (intermediate contract) account for all users of the protocol interacting with the `transferTo()` and `transferFrom()` functions and lead to incorrect accounting of token balances or exceptional conditions. Protocol will break and lead to fund loss.

Recommend changing L129 to: `_burn(sender, _fee);`

strictly-scarce (vader) confirmed:

> Valid, disagree with severity though. Funds-not-at-risk. Recommend: 2

[H-07] Wrong `calcAsymmetricShare` calculation

The inline-comment defines the number of asymmetric shares as `(u * U * (2 * A^2 - 2 * U * u + U^2))/U^3` but the `Utils.calcAsymmetricShare` function computes `(uA * 2U^2 - 2uU + u^2) / U^3` which is not equivalent as can be seen from the `A^2` term in the first term which does not occur in the second one.

The associativity on `P * part1` is wrong, and `part2` is not multiplied by `P`.

The math from the spec is not correctly implemented and could lead to the protocol being economically exploited, as the asymmetric share (which is used to determine the collateral value in base tokens) could be wrong. For example, it might be possible to borrow more than the collateral put up.

Recommend clarifying if the comment or the code is correct and fix them if not.

strictly-scarce (vader) confirmed:

> Valid

strictly-scarce (vader) commented:

> Whilst the math is incorrect, in the current implementation it is not yet implemented, so disagree with Severity (funds not lost), recommend: 2

[H-08] Wrong liquidity units calculation

The spec defines the number of LP units to be minted as `units = (P (a B + A b))/(2 A B) * slipAdjustment = P * (part1 + part2) / part3 * slipAdjustments` but the `Utils.calcLiquidityUnits` function computes `((P * part1) + part2) / part3 * slipAdjustments`.

The associativity on `P * part1` is wrong, and `part2` is not multiplied by `P`.

The math from the spec is not correctly implemented and could lead to the protocol being economically exploited, as redeeming the minted LP tokens does not result in the initial tokens anymore.

Recommend fixing the equation.

strictly-scarce (vader) confirmed:

> Valid, but funds not at risk.

[H-09] Incorrect initialization gives IL protection of only 1 second instead of 100 days in Router.sol

Incorrect initialization of `timeForFullProtection` to 1 sec instead of 8640000 secs (100 days) as indicated in code comments, appears to be a test setting mistakenly carried over for deployment. Therefore, unless `timeForFullProtection` is reset to 100 days by `setParams()` (calling this function is a missing functionality in the DAO currently), the Impermanent Loss (IL) protection "rule" of 100 days will not apply in `Utils.getProtection()`.

This breaks a key value proposition of the Vader protocol which is IL protection as indicated in the specification:

> "Impermanent Loss Protection: The deposit value for each member is recorded when they deposit. When they go to withdraw, the redemption value is computed. If it is less than the deposit value, the member is paid the deficit from the reserve. The protection issued increases from 0 to 100% linearly for 100 days."

Recommend changing to `"timeForFullProtection = 8640000; //100 days"` on L84

strictly-scarce (vader) disputed:

> It's deliberately set to 1 second to conduct adequate testing.

[H-10] Anyone can list anchors / curate tokens

The `Router.listAnchor` function can be called by anyone and tokens can be added. The only check is that `require(iPOOLS(POOLS).isAnchor(token));` but this can easily be set by calling `Pools.addLiquidity(VADER, token, _)` once even without actually sending any tokens to the contract. This makes it an essentially useless check.

This only works initially as long as the `anchorLimit` has not been reached yet. However, the `replaceAnchor` can be used in the same way and flash loans can be used to get around the liquidity restrictions and push another anchor token out of the price range as these checks use the current reserves.

Anchored pools are automatically curated pools and determine if a pool receives rewards. An attacker can remove rewards of a curated pool this way and add rewards to their own pool with a custom token they control.

After a pool has been anchored through flash loans, liquidity can be withdrawn which could make the anchor price easy to manipulate in the next block and launch other attacks.

Recommend revisiting the `_isAnchor[token] = true;` statement in `addLiquidity`, it seems strange without any further checks. Consider making

`listAnchor` / `replaceAnchor` DAO-only functions and make them flash-loan secure. One should probably use time-weighted prices for these pools for the bounds check.

strictly-scarce (vader) disputed:

> The protocol is intended to be launched with 5 anchors so it can only be attacked by using `replaceAnchor()`, in which case slip-based fees apply for attacks and thwart the attack path.

[H-11] Swap token can be traded as fake base token

The `Pools.swap` function does not check if `base` is one of the base tokens. One can transfer `token`s to the pool and set `base=token` and call `swap(token, token, member, toBase=false)`

The `_actualInput = getAddedAmount(base, token);` will return the **token** amount added but use the ratio compared to the **base** reserve `calcSwapOutput(_actualInput=tokenInput, mapToken_baseAmount[token], mapToken_tokenAmount[token]); = tokenIn / baseAmount * tokenAmount` which yields a wrong swap result.

It breaks the accounting for the pool as `token`s are transferred in, but the `base` balance is increased (and `token` balance decreased). LPs cannot correctly withdraw again, and others cannot correctly swap again.

Another example scenario is that the token pool amount can be stolen. Send `tokenIn=baseAmount` of tokens to the pool and call `swap(base=token, token, member, toBase=false)`. Depending on the price of `token` relative to `base` this could be cheaper than trading with the base tokens.

Recommend checking that `base` is either `USDV` or `VADER`.

strictly-scarce (vader) confirmed:

> Valid, funds can be lost

[H-12] `getAddedAmount` can return wrong results

The `getAddedAmount` function only works correctly when called with `(VADER/USDV, pool)` or `(pool, pool)`. However, when called with (`token`, `pool`) where `token` is neither `VADER/USDV/pool`, it returns the wrong results:

1. It gets the `token` balance
2. And subtracts it from the stored `mapToken_tokenAmount[_pool]` amount which can be that of a completely different token

Anyone can break individual pairs by calling `sync(token1, token2)` where the `token1` balance is less than `mapToken_tokenAmount[token2]`. This will add the difference to `mapToken_tokenAmount[token2]` and break the accounting and result in a wrong swap logic.

Furthermore, this can also be used to swap tokens without having to pay anthing with `swap(token1, token2, member, toBase=false)`.

Recommend adding a require statement in the `else` branch that checks that `_token == _pool`.

strictly-scarce (vader) confirmed:

> Valid, funds can be lost

strictly-scarce (vader) commented:

> Would bundle this issue with: https://github.com/code-423n4/2021-04-vader-findings/issues/205

[H-13] 4 Synths can be minted with fake base token

The `Pools.mintSynth` function does not check if `base` is one of the base tokens. One can transfer `token`s to the pool and set `base=token` and call `mintSynth(token, token, member)`.

The `_actualInput = getAddedAmount(base, token);` will return the **token** amount added but use the ratio compared to the **base** reserve `calcSwapOutput(_actualInput=tokenInput, mapToken_baseAmount[token], mapToken_tokenAmount[token]); = tokenIn / baseAmount * tokenAmount` which yields a wrong swap result.

It breaks the accounting for the pool as `token` s are transferred in, but the `base` balance is increased.

The amount that is minted could also be inflated (cheaper than sending the actual base tokens), especially if `token` is a high-precision token or worth less than base.

Recommend checking that `base` is either `USDV` or `VADER` in `mintSynth`.

strictly-scarce (vader) confirmed:

> Valid, funds can be lost.

strictly-scarce (vader) commented:

> would bundle this issue with: https://github.com/code-423n4/2021-04-vader-findings/issues/205

[H-14] Missing access restriction on `lockUnits/unlockUnits`

The `Pool.lockUnits` allows anyone to steal pool tokens from a `member` and assign them to `msg.sender`. Anyone can steal pool tokens from any other user.

Recommend adding access control and require that `msg.sender` is the router or another authorized party.

strictly-scarce (vader) confirmed:

> Valid, although this is part of the partially-complete lending code.

[H-15] Wrong slippage protection on Token -> Token trades

The `Router.swapWithSynthsWithLimit` allows trading token to token and specifying slippage protection. A token to token trade consists of two trades:

1. token to base
2. base to token

The slippage protection of the second trade (base to token) is computed wrong:

```
require(iUTILS(UTILS()).calcSwapSlip(
    inputAmount, // should use outToken here from prev trade
    iPOOLS(POOLS).getBaseAmount(outputToken)
  ) <= slipLimit
);
```

It compares the **token** input amount (of the first trade) to the **base** reserve of the second pair.

Slippage protection fails and either the trade is cancelled when it shouldn't be or it is accepted even though the user suffered more losses than expected.

Recommend it should use the base output from the first trade to check for slippage protection. Note that this still just computes the slippage protection of each trade individually. An even better way would be to come up with a formula to compute the slippage on the two trades at once.

strictly-scarce (vader) confirmed:

> Valid, although disagree with severity, the wrongly compute slip amount would just fail the trade or allow the second trade to go thru with no protection.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements

> 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 1

[H-16] Tokens can be stolen through `transferTo`

I know that it's stated that:

> VADER, USDV, SYNTHS all employ the `transferTo()` function, which interrogates for `tx.origin` and skips approvals. The author does not subscribe to the belief that this is dangerous

In my opinion, it can be very dangerous. Imagine the following scenario:

1. I create a custom attacker ERC20 token that has a hook in the `_transfer` function that checks tx.origin for USDV/VADER/SYNTHS and calls `transferTo` to steal these funds.
2. I set up a honeypot by providing liquidity to the `BASE <> ATTACKER` pool.
3. I target high-profile accounts holdinging VADER/USDV/SYNTHS and airdrop them free tokens.
4. Block explorers / Vader swap websites could show that this token has value and can be traded for actual `BASE` tokens.
5. User wants to sell the airdropped `ATTACKER` token to receive valuable tokens through the Vader swap and has all their tokens (that are even completely unrelated to the tokens being swapped) stolen.

In general, a holder of any of the core assets of the protocol risks all their funds being stolen if they ever interact with an unvetted external contract/token. This could even be completely unrelated to the VADER protocol.

Recommend removing `transferTo` and use `permit + transferFrom` instead to move tokens from `tx.origin`.

strictly-scarce (vader) acknowledged:

> This attack path has already been assessed as the most likely, no new information is being presented here.

> Do not interact with attack contracts, interacting with an ERC20 is an attack contract.

0xBrian commented:

> @strictly-scarce (vader) What would be the downside of adopting the suggested mitigation? Since we cannot communicate effectively with all users to tell them not to interact with certain kinds of contracts (and even if we could, they may not be able to discern which are OK and which aren't), we don't want to set up a thicket for fraudsters to operate. If the downside of the mitigation is not too bad, I think it could be worth it in order to deny fraudsters an opportunity to steal.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 0

[H-17] Transfer fee is burned on wrong accounts

The `Vader._transfer` function burns the transfer fee on `msg.sender` but this address might not be involved in the transfer at all due to `transferFrom`.

Smart contracts that simply relay transfers like aggregators have their Vader balance burned or the transaction fails because these accounts don't have any balance to burn, breaking the functionality.

Recommend that It should first increase the balance of `recipient` by the full amount and then burn the fee on the `recipient`.

strictly-scarce (vader) confirmed:

> For composabilty with the rest of the ecosystem, this should be addressed, although disagree with the severity, no funds are lost, just the aggregrator cannot transfer unless they first transfer to themselves, which most often do.

Mervyn853 commented:

> Our decision matrix for severity:
>
> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.
>
> Recommended: 2

[H-18] Vault rewards can be gamed

The `_deposit` function increases the member's *weight* by `_weight = iUTILS(UTILS()).calcValueInBase(iSYNTH(_synth).TOKEN(), _amount);` which is the swap output amount when trading the deposited underlying synth amount.

Notice that anyone can create synths of custom tokens by calling `Pools.deploySynth(customToken)`.

Therefore an attacker can deposit valueless custom tokens and inflate their member weight as follows:

1. Create a custom token and issue lots of tokens to the attacker
2. Create synth of this token

3. Add liquidity for the `TOKEN <> BASE` pair by providing a single wei of `TOKEN` and `10^18` BASE tokens. This makes the `TOKEN` price very expensive.
4. Mint some synths by paying BASE to the pool
5. Deposit the fake synth, `_weight` will be very high because the token pool price is so high.

Call `harvest(realSynth)` with a synth with actual value. This will increase the synth balance and it can be withdrawn later.

Anyone can inflate their member weight through depositing a custom synth and earn almost all vault rewards by calling `harvest(realSynth)` with a valuable "real" synth. The rewards are distributed pro rata to the member weight which is independent of the actual synth deposited.

The `calcReward` function completely disregards the `synth` parameter which seems odd. Recommend thinking about making the rewards based on the actual synths deposited instead of a "global" weight tracker. Alternatively, whitelist certain synths that count toward the weight, or don't let anyone create synths.

strictly-scarce (vader) confirmed:

> This is a valid attack path.

> The counter is two fold:

1. In the vault, `require(isCurated(token))` this will only allow synths of curated tokens to be deposited for rewards. The curation logic does a check for liquidity depth, so only deep pools can become synths. Thus an attacker would need to deposit a lot of BASE.
2. In the vaults, use `_weight = iUTILS(UTILS()).calcSwapValueInBase(iSYNTH(_synth).TOKEN(), _amount);` , which computes the weight with respect to slip, so a small manipulated pool cannot be eligible. The pool would need to be deep.

> The Vault converts all synths back to common accounting asset - USDV, so member weight can be tracked.

> **strictly-scarce (vader) commented**: Disagree with severity, since the daily rewards can be claimed by anyone in a fee-bidding war but no actual extra inflation occurs.

> Severity: 2

[H-19] Vault rewards last claim time not always initialized

The `harvest` calls `calcCurrentReward` which computes `_secondsSinceClaim = block.timestamp - mapMemberSynth_lastTime[member][synth];`. As one can claim different synths than the synths that they deposited, `mapMemberSynth_lastTime[member][synth]` might still be uninitialized and the `_secondsSinceClaim` becomes the current block timestamp.

The larger the `_secondsSinceClaim` the larger the rewards. This bug allows claiming a huge chunk of the rewards.

Recommend letting users only harvest synths that they deposited.

strictly-scarce (vader) confirmed:

> This is valid.

> The member should only claim against synths they have deposited, where the time would be initialised.

strictly-scarce (vader) commented:

> Would place this as severity: 2, since the anyone can participate in claiming rewards, but no extra inflation occurs.

[H-20] Vault Weight accounting is wrong for withdrawals

When depositing two different synths, their weight is added to the same `mapMember_weight[_member]` storage variable. When withdrawing the full amount of one synth with `_processWithdraw(synth, member, basisPoints=10000` the full weight is decreased.

The second deposited synth is now essentially weightless.

Users that deposited more than one synth can not claim their fair share of rewards after a withdrawal.

Recommed that the weight should be indexed by the synth as well.

strictly-scarce (vader) confirmed:

> This is valid.

> The weight should be reduced only as applied to a specific synth

> There is no loss of funds, just less rewards for that member, disputing severity level.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 2

dmvt (judge) commented:

> My viewpoint on this and the last several reward based high risk issues is that loss of rewards is loss of funds. High risk is appropriate.

[H-21] Anyone Can Avoid All Vether Transfer Fees By Adding Their Address to the Vether `ExcludedAddresses` List.

`Vether.sol` implements a fee on every token transfer, unless either the sender or the recipient exists on a list of excluded addresses `(mapAddress_Excluded)`. However, the `addExcluded()` function in `Vether.sol` has no restrictions on who can call it. So any user can call `addExcluded` with their own address as the argument, and bypass all transfer fees.

Alice calls:

(1) `Vether.addExcluded(aliceAddress)`, which adds Alice's address to `mapAddress_Excluded`. (2) Alice can now freely transfer Vether with no fees.

Recommend adding restrictions to who can call `addExcluded`, perhaps by restricting it to a caller set by `DAO.sol`

strictly-scarce (vader) commented:

> Vether contract is outside of contest

dmvt (judge) commented:

> https://github.com/code-423n4/2021-04-vader-findings/issues/3#issuecomment-849043144

> The warden should be paid out on this issue, in my opinion, because the code was included in the repo to be reviewed. The work to review the contract was done despite the fact that the team has addressed the issue and has already deployed `vether.sol`. I do not think that any issues related to `Vether.sol` should be included in the final report generated by @code423n4.

> It was unclear to me (and obviously most of the wardens) that `Vether.sol` was considered out of scope.

moneylegobatman (C4 Editor) commented:

> Leaving report and discussion in for transparency, since finding was awarded.

[H-22] Users may unintentionally remove liquidity under a phishing attack.

The `removeLiquidity` function in `Pools.sol` uses `tx.origin` to determine the person who wants to remove liquidity. However, such a design is dangerous since the pool assumes that this function is called from the router, which may not be true if the user is under a phishing attack, and he could unintentionally remove liquidity.

Referenced code: Pool.sol#L77-L79

Recommend consider making the function `_removeLiquidity` external, which can be utilized by the router, providing information of which person removes his liquidity.

strictly-scarce (vader) acknowledged:

> If a user has been phished, consider all their funds already stolen.

> Vader's security assumption is a user is not phished.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 0

dmvt (judge) commented:

> This is reasonably easy to mitigate as an issue and failure to do so does leave an attack vector open. If exploited it will result in a loss of user funds.

[H-23] Anyone can curate pools and steal rewards

The `Router.curatePool` and `replacePool` don't have any access restriction. An attacker can get a flash loan of base tokens and replace existing curated pools with their own curated pools.

Curated pools determine if a pool receives rewards. An attacker can remove rewards of a curated pool this way and add rewards to their own pool with a custom token they control. They can then go ahead and game the reward system by repeatedly swapping in their custom pool with useless tokens, withdraw liquidity, and in the end, pay back the base flashloan.

Recommend preventing the replacing of curations through flash loans. Also, consider making pool curations DAO-exclusive actions.

strictly-scarce (vader) disputed:

> Slip-based pools cannot be attacked with flash loans.

dmvt (judge) commented:

> Further comment from @cmichelio:
>
> > I can curate my custom token using `curatePool` without using a flashloan or using replacePool by temporarily providing liquidity to the pool without trading in it and getting slip-fee'd. I'm not trading in the pool, and don't think providing/removing liquidity comes with a fee. I think this is still an issue.

[H-25] Incorrect initialization causes VADER emission rate of 1 second instead of 1 day in Vader.sol

Incorrect initialization (perhaps testing parameterization mistakenly carried over to deployment) of `secondsPerEra` to 1 sec instead of 86400 secs (1 day) causes what should be the daily emission rate to be a secondly emission rate.

This causes inflation of VADER token and likely breaks VADER<>USDV peg and other protocol invariants. Protocol will break and funds will be lost.

Recommend Initializing `secondsPerEra` to 86400 on L67.

strictly-scarce (vader) acknowledged:

> This is purely for testing purposes.

Medium Risk Findings

[M-01] User may not get IL protection if certain functions are called directly in `Pools.sol`

Functions `removeLiquidity()` and `removeLiquidityDirectly()` when called directly, do not provide the the user with IL protection unlike when calling the corresponding `removeLiquidity()` function in `Router.sol`. This should be prevented, at least for `removeLiquidity()` or highlighted in the specification and user documentation.

Recommend adding access control (e.g. via a modifier `onlyRouter`) so `removeLiquidity()` function of Pools contract can be called only from corresponding Router contract's `removeLiquidity()` function which provides IL protection. Alternatively, highlight in the specification and user documentation about which contract interfaces provide IL protection to users.

strictly-scarce (vader) acknowledged:

> User should use the Router, as designed.

[M-02] Undefined behavior for DAO and GRANT vote proposals in `DAO.sol`

Given that there are only three proposal types (GRANT, UTILS, REWARD) that are actionable, it is unclear if 'DAO' type checked in `voteProposal()` is a typographical error and should really be 'GRANT'. Otherwise, GRANT proposals will only require quorum (33%) and not majority (50%).

Recommend changing 'DAO' on L83 to 'GRANT' or if not, specify what DAO proposals are and how GRANT proposals should be handled with quorum or majority.

Also, check and enforce that `mapPID_types` are only these three actionable proposal types: GRANT, UTILS, REWARD.

strictly-scarce (vader) acknowledged:

> DAO not yet fully implemented

[M-03] Lack of input validation in `replacePool()` allows curated pool limit bypass in `Router.sol`

There is no input validation in `replacePool()` function to check if `oldToken` exists and is curated. Using a non-existing `oldToken` (even 0 address) passes the check on L236 (because `Pools.getBaseAmount()` will return 0 for the non-existing token) and `newToken` will be made curated. This can be used to bypass the `curatedPoolLimit` enforced only in `curatePool() function.`

Recommend checking if `oldToken` exists and is curated as part of input validation in `replacePool()` function.

strictly-scarce (vader) confirmed:

> Valid

[M-04] `flashProof` is not flash-proof

The `flashProof` modifier is supposed to prevent flash-loan attacks by disallowing performing several sensitive functions in the same block.

However, it performs this check on `tx.origin` and not on an individual user address basis. This only prevents flash loan attacks from happening within a single transaction.

But flash loan attacks are theoretically not limited to the same transaction but to the same block as miners have full control of the block and include several vulnerable transactions back to back. (Think transaction *bundles* similar to flashbot bundles that most mining pools currently offer.)

A miner can deploy a proxy smart contract relaying all contract calls and call it from a different EOA each time bypassing the `tx.origin` restriction.

The `flashProof` modifier does not serve its purpose.

Recommend trying to apply the modifier to individual addresses that interact with the protocol instead of `tx.origin`.

Furthermore, attacks possible with flash loans are usually also possible for whales, making it debatable if adding flash-loan prevention logic is a good practice.

strictly-scarce (vader) confirmed:

> Flash loans with the help of miners *was not intended to be countered*, although a check for `msg.sender` AND `tx.origin` will be applied.

[M-05] Interest debt is capped after a year

The `Utils.getInterestOwed` function computes the `_interestPayment` as:

```
    uint256 _interestPayment =
      calcShare(
          timeElapsed,
          _year,
          getInterestPayment(collateralAsset, debtAsset)
      ); // Share of the payment over 1 year
```

However, `calcShare` caps `timeElpased` to `_year` and therefore the owed interest does not grow after a year has elapsed.

The impact is probably small because the only call so far computes the elapsed time as `block.timestamp - mapCollateralAsset_NextEra[collateralAsset][debtAsset];` which most likely will never go beyond a year.

It's still recommended to fix the logic bug in case more functions will be added that use the broken function.

Recommend using a different function than `calcShare` that does not cap.

strictly-scarce (vader) confirmed:

> A member who doesn't interact with the contract for more than a year misses out on some rewards, so severity:1

[M-06] Canceled proposals can still be executed

Proposals that passed the threshold ("finalized") can be cancelled by a minority again using the `cancelProposal` functions. It only sets `mapPID_votes` to zero but `mapPID_timeStart` and `mapPID_finalising` stay the same and pass the checks in `finaliseProposal` which queues them for execution.

Proposals cannot be cancelled.

Recommend setting a cancel flag and check for it in `finaliseProposal` and in execution.

strictly-scarce (vader) confirmed:

> Valid

[M-07] Completed proposals can be voted on and executed again

A proposal that is completed has its state reset, including the votes. Users can just vote on it again and it can be executed again.

Completed proposals should most likely not be allowed to be voted on / executed again. This could also lead to issues in backend scripts that don't expect any voting/execution events to be fired again after the `FinalisedProposal` event has fired.

Recommend adding an `executed` flag to the proposals and disallow voting/finalising on already executed proposals.

strictly-scarce (vader) disputed:

> It might be intended to have repeated proposals.

[M-09] Divide before multiply

Here you have more information:
https://gist.github.com/alexon1234/e5038a9f66136ae210be692f8803d874

strictly-scarce (vader) questioned:

> Can't quite understand the assertion that a division is made before a multiply in the code outlined

```
uint _units = (((P * part1) + part2) / part3);
    return (_units * slipAdjustment) / one;  // Divide by 10**18
```

> `_units` will be `0 -> 2**256`. `slipAdjustment` will be `0 -> 10**18` `one` is `10**18`

```
// returns 0
 return (0 * 10**18) / 10**18;
 return (2**256 * 0) / 10**18;
return (<10**9 * <10**9) / 10**18;
  // returns  non-zero
 return (>=10**9 * >=10**9) / 10**18;
```

[M-10] Incorrect operator used in `deploySynth()` of `Pools.sol`

The `deploySynth()` function in `Pools.sol` is expected to perform a check on the token parameter to determine that it is neither VADER or USDV before calling Factory's `deploySynth()` function.

However, the `require()` incorrectly uses the '||' operator instead of '&&' which allows both VADER and USDV to be supplied as the token parameters. This will allow an attacker to deploy either VADER or USDV as a Synth which will break

assumptions throughout the entire protocol. Protocol will break and funds may be lost.

Recommend changing '||' operator to '&&' in the require statement: `require(token != VADER && token != USDV);`

strictly-scarce (vader) addressed:

> Duplicate https://github.com/code-423n4/2021-04-vader-findings/issues/21

0xBrian commented:

> https://github.com/vetherasset/vaderprotocol-contracts/pull/159/commits/2f69f8317ce98846fbe227a3bf6ca1b644d01ff2#diff-5de3130299a0ddc914d7a906802a4cc093ed18d7a89c52a4aafefc8a11ac3f54R193

[M-11] Allowing duplicated anchors could cause bias on anchor price.

In `Router.sol`, the setup of the five anchors can be interrupted by anyone adding a new anchor due to the lack of access control of the `listAnchor` function. Also, duplicate anchors are allowed. If the same anchor is added three times, then this anchor biases the result of `getAnchorPrice`.

Referenced code: Router.sol#L245-L252

PoC: Link to PoC See the file `200_listAnchor.js` for a PoC of this attack. To run it, use `npx hardhat test 200_listAnchor.js`.

Recommend only allowing `listAnchor` to be called from the deployer by adding a `require` statement. Also, check if an anchor is added before by `require(_isCurated == false)`.

strictly-scarce (vader) acknowledged:

> Deployer will list the anchors, seems highly unlikely they will get griefed in practice. Severity: 1

[M-12] Transfer fee avoidance

The `Vether4.addExcluded()` function on mainnet (0x4Ba6dDd7b89ed838FEd25d208D4f644106E34279) allows a user to exclude an address from transfer fees for a cost of 128 VETH. By exploiting the conditions in which fees are taken, it is possible to set up a contract for a once-off cost in which all users can use to avoid transfer fees.

All transfer fees can be avoided by routing transfers through an excluded contract. An estimated $140k of transfer fees was accumulated at the time of writing. These fees can be avoided in future, causing an indirect loss of funds for the contract.

Recommend that the `_transfer()` function should be updated to only exclude transfer fees if the sender has been excluded, not both the sender and the recipient. This would prevent any user from being able to set up a central transfer forwarder as demonstrated. Moreover, the `Transfer(_from, address(this), _fee);` event should only be emitted if the sender has been excluded from transfer fees.

[strictly-scarce (vader) disputed]

[M-13] Init function can be called by everyone

Most of the solidity contracts have an init function that everyone can call. This could lead to a race condition when the contract is deployed. At that moment a hacker could call the `init` function and make the deployed contracts useless. Then it would have to be redeployed, costing a lot of gas.

```
DAO.sol:    function init(address _vader, address _usdv, address
Factory.sol:    function init(address _pool) public {
Pools.sol:    function init(address _vader, address _usdv, addre
Router.sol:    function init(address _vader, address _usdv, addr
USDV.sol:    function init(address _vader, address _vault, addre
Utils.sol:    function init(address _vader, address _usdv, addre
```

```
Vader.sol:    function init(address _vether, address _USDV, addr
Vault.sol:    function init(address _vader, address _usdv, addre
```

Recommend adding a check to the `init` function, for example that only the deployer can call the function.

strictly-scarce (vader) confirmed:

> Yes, but only once. Could add a deployer check tho

dmvt (judge) commented:

> After considerable evaluation and seeing the wide range of threat factors that were put forward by wardens related to this issue, I've decided that the potential threat here does extend beyond gas.
>
> A worst case scenario could cause significant damage.
>
> It is extremely unlikely that an attacker could successfully time this type of attack.
>
> An attacker would have to successfully intercept more than one init due to the highly coupled nature of the contract. If they did so incorrectly, the entire system would not function. Presuming they succeeded, the team would also have to overlook the failure of or forget to make multiple critical transaction calls in their deployment scripts. To realize significant financial gains, the attacker would have to leave their exploit code in place for an extended period of time.
>
> The likelihood is extremely low, but the impact would be critical. For this reason, I'm normalizing all of these reports to a Medium Risk.

[M-14] Pool functions can be called before initialization in *`init`*() of Pools.sol

All the external/public functions of `Pools.sol` can be called by other contracts even before `Pools.sol` contract is initialized. This can lead to exceptions, state corruption or incorrect accounting in other contracts, which may require redeployment of said contract.

Recommend using a factory pattern that will deploy and initialize atomically to prevent front-running of the initialization,

OR

Given that contracts are not using `delegatecall` proxy pattern, it is not required to use a separate `init()` function to initialize parameters when the same can be done in a constructor. If the reason for doing so is to get the deployment addresses of the various contracts, which may not all be available at the same time, then consider rearchitecting to create a "globals" contract which can hold all the globally required addresses of various contracts. see Maple protocol's for example.

OR

Prevent external/public functions from being called until after initialization is done by checking initialization state tracked by the inited variable.

strictly-scarce (vader) dipsuted:

> https://github.com/code-423n4/2021-04-vader-findings/issues/39

dmvt (judge) commented:

> Same general comments apply to this issue as with issue #18, but it is a separate type of exploit that would be slightly less detectable. This increase in risk is balanced against the exploit being much harder to effect and the likely impact being lower.

[M-15] `changeDAO` should be a two-step process in Vader.sol

`changeDAO()` updates DAO address in one-step. If an incorrect address is mistakenly used (and voted upon) then future administrative access or recovering from this mistake is prevented because `onlyDAO` modifier is used for `changeDAO()`, which requires `msg.sender` to be the incorrectly used DAO address (for which private keys may not be available to sign transactions). See finding #6 from Trail of Bits audit of Hermez Network.

Recommend using a two-step process where the old DAO address first proposes new ownership in one transaction; and then, accepts ownership from the newly proposed DAO address in a second transaction. A mistake in the first step can be recovered by granting with a new correct address again before the new DAO address accepts ownership. Ideally, there should also be a timelock enforced before the new DAO takes effect.

strictly-scarce (vader) confirmed:

> A lot has to go wrong to get to this point, so disagree with severity (funds not at risk).

> Two step-process seems wise though.

dmvt (judge) commented:

> Risk lowered because of the extremely low probability

[M-16] Copy-paste bug leading to incorrect harvest rewards in Vault.sol

The conditional in `calcReward()` function uses the same code in both if/else parts with repeated use of `reserveUSDV`, `reserveVADER` and `getUSDVAmount` leading to incorrect computed value of `_adjustedReserve` in the else part.

This will affect harvest rewards for all users of the protocol and lead to incorrect accounting. Protocol will break and lead to fund loss.

Recommend changing variables and function calls from using USDV to VADER in the else part of the conditional which has to return the adjusted reserves when synth is not an asset i.e. an anchor and therefore base is VADER. L144 should be changed to:

```
uint _adjustedReserve = iROUTER(ROUTER).getVADERAmount(reserveUSDV())
+ reserveVADER();
```

strictly-scarce (vader) confirmed:

> Funds are not at-risk, just that some users will get less rewards, some will get more.

> Recommend: 2

[Mervyn853 commented]():

> Our decision matrix for severity:
>
> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.
>
> Recommended: 2

[M-17] `Vader.redeemToMember()` vulnerable to front running

The USDV balance of the Vader contract is vulnerable to theft through the `Vader.redeemToMember()` function. A particular case is through USDV redemption front-running. Users can redeem USDV for Vader through the `USDV.redeemForMember()` function or the `Vader.redeemToMember()` function. In the case of `Vader.redeemToMember()` , a user would need to send their USDV to the contract before redemption. However, as this process does not happen in a single call, the victim's call is vulnerable to front running and could have their redeemed USDV stolen by an attacker.

User's redeem USDV could be stolen by an attacker front running their `Vader.redeemToMember()` call.

The steps are as follows:

1. User sends USDV to Vader contract to be redeemed
2. User calls `Vader.redeemToMember()`
3. The `Vader.redeemToMember()` call is detected by an attacker, who front-runs the call by calling `Vader.redeemToMember()` specifying their own address as the member parameter.
4. The full USDV balance of the Vader contract is redeemed and sent to the attacker.

Note that while this particular case is front running a redemption call, any USDV balance could be stolen in this manner. Please find the POC showing the above steps here: https://gist.github.com/toastedsteaksandwich/39bfed78b21d7e6c02fe13ea5b2023c3

Recommend that the `Vader.redeemToMember()` function should be restricted so that only the USDV contract can call it. Moreover, the amount parameter from `USDV.redeem()` or `USDV.redeemForMember()` should also be passed to `Vader.redeemToMember()` to avoid the need to sweep the entire USDV balance. In this way, the member's redemption happens in a single tx, and would only be allocated as much Vader as redeemed in USDV.

strictly-scarce (vader) disputed:

> Vader complies with a monetary security policy of "money in, money out". Contracts will only send out funds if they are first sent funds.

> This is the case for the entire system, not just `Vader.redeemToMember()`, such as swaps and adding liquidity. Vader is not designed to be interacted with directly, it should be wrapped. In this case, users should convert and redeem only thru the USDV contract, which first sends funds.

> Incidentally this is the same mechanism that uniswap employs for withdrawing liquidity, or syncing funds to balances. You can also get front-runned if you do it in two tx, it should be wrapped in 1 tx.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 0

Low Risk Findings

[L-01] Missing event for critical `init()` function in Factory.sol

The `init()` function initializes critical POOLS protocol address for this contract but is missing an event emission for off-chain monitoring tools to monitor this on-chain change.

Recommend adding an init event and emit that at the end of `init()` function.

[L-02] Uninitialized variable leads to zero-fees for first transfer in `Vader.sol`

The state variables `feeOnTransfer` is never initialized which leads to a default uint value of 0. When it is used on L126 in the first call to `_transfer()`, it will lead to a zero fee. `feeOnTransfer` is updated only in function `_checkEmission()` whose call happens later on L133, after which it has a value as calculated in that function. This causes only the first transfer to be a zero-fee transfer.

Recommend initializing `feeOnTransfer` suitably on declaration, in constructor, or `init()` function.

[L-03] Misleading comment for `deposit()` function of `Vault.sol`

Use of accurate comments helps users read, audit and maintain code. Inaccurate comments can be misleading, obstruct the flagging of vulnerabilities, or even introduce them.

Misleading comment on L76 that says `deposit()` function allows USDV and Synths, but the code only allows Synths.

Recommend using accurate and descriptive comments (even NatSpec) correctly describing the function behavior, parameters and return values.

[L-04] Fee can be at most 1% and dead code

The `Vader._checkEmission` functions caps the fee at `1000` = 10% but the max fee that can be returned from the `iUTILS(UTILS).getFeeOnTransfer` call is `100`.

```
// returns value between 0 and 100
feeOnTransfer = iUTILS(UTILS).getFeeOnTransfer(
    totalSupply,
    maxSupply
); // UpdateFeeOnTransfer
if (feeOnTransfer > 1000) {
    feeOnTransfer = 1000;
} // Max 10% if UTILS corrupted
```

It seems like there is a misunderstanding in whether the fee should be at most 1% (`Utils.sol`) or 10% (`Vader.sol`).

Recommend clarifying what the max fee should be, and then adjusting either `Utils.getFeeOnTransfer` or the `Vader._checkEmission` cap.

[L-05] Lack of zero address validation in `init()` function

The parameters that are used in `init()` function to initialize the state variable, these state variable are used in other function to perform operation. since it lacks zero address validation, it will be problematic if there is error in these state variable. some of the function will loss their functionality which can cause the redeployment of contract.

Recommend adding require condition which check zero address validation

strictly-scarce (vader) acknowledged:

> Same as: https://github.com/code-423n4/2021-04-vader-findings/issues/13#issuecomment-826294937

> No issue found

[L-06] Events not emitted

Events not emitted for important state changes.

strictly-scarce (vader) acknowledged:

> The ethereum state machine isn't a parking lot for event data
>
> - `setParams()` - no, plenty events in DAO
> - `setAnchorParams()` - no, plenty events in DAO
>
> But these are warranted, purely for off-chain metrics:
>
> - `addDepositData()` - valid for off-chain IL tracking
> - `removeDepositData()` - valid for off-chain IL tracking

strictly-scarce (vader) commented:

> Severity: 0, no impact to protocol itself

[L-07] Swap fee not applied

Here you have more information:
https://gist.github.com/alexon1234/a2d3619fb3faa4e5676329f70bd565d3

[L-08] The Calculation For `nextEraTime` Drifts, Causing Eras To Occur Further And Further Into The Future

In `Vader.sol`, eras are intended to occur every 24 hours. This means that a correct implementation would add 24 hours to the end-time of the previous era in order to find the end-time of the next era. However, the current method for calculating the next era's end-time uses `block.timestamp`, rather than the previous era's end-time.

This line of code will cause a perpetual drift of era times, causing each era to actually be 24 hours plus the time between when the last era ended and when `Vader._transfer()` is next called.

Recommend that In `Vader.sol`, change this:

```
nextEraTime = block.timestamp + secondsPerEra;
```

to this:

```
nextEraTime = nextEraTime + secondsPerEra;
```

[L-09] `_recordBurn` does not handle 0 _eth appropriately

Contract Vether4 function `_recordBurn` does not check that `_eth > 0`, thus it is possible to pass this check multiple times:

```
if (mapEraDay_MemberUnits[_era][_day][_member] == 0)
```

If the user hasn't contributed to this day yet, it updates `mapMemberEra_Days`, `mapEraDay_MemberCount`, and `mapEraDay_Members`. However, when `msg.value` is 0, it is possible to trigger this condition again and again as `mapEraDay_MemberUnits` still remains 0.

Recommend either not allowing burns of 0 _eth, or add an extra check in the if statement.

[L-10] `getAnchorPrice` potentially returns the wrong median

The `Router.getAnchorPrice` sorts the `arrayPrices` array and always returns the third element `_sortedAnchorFeed[2]`. This only returns the median if `_sortedAnchorFeed` is of length 5, but it can be anything from `0` to `anchorLimit`.

If not enough anchors are listed initially, it might become out-of-bounds and break all contract functionality due to revert, or return a wrong median. If `anchorLimit` is set to a different value than 5, it's also wrong.

recommend checking the length of `_sortedAnchorFeed` and return `_sortedAnchorFeed[_sortedAnchorFeed.length / 2]` if it's odd, or the average of the two in the middle if it's even.

strictly-scarce (vader) confirmed:

> Valid, although it is intended design to launch with 5, which cannot be reduced after, so disagree with severity.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 2

[L-11] `listAnchor` sets `_isCurated` to true but forgets other parts of curation

The function `listAnchor` sets `_isCurated` to true but does not update the `curatedPoolCount` and does not emit the Curated event. I don't see this `curatedPoolCount` variable used anywhere so probably it's just needed for the frontend consumption.

Recommend that the best solution would be to replace `_isCurated[token] = true` ; with call to function `curatePool` . It also skips if the same anchor is listed twice.

[L-12] `curatePool` emits Curated event no matter what

The function `curatePool` emits Curated event every time. It should emit this event only when the conditions are fulfilled.

Recommend putting this event inside the most inner `if` block.

[L-13] calculations of `upgradedAmount` is not overflow protected

As contract Vether4 is using pragma solidity 0.6.4; SafeMath is not enabled by default, thus making this check inside function distribute avoidable (overflow):

```
    upgradedAmount += ownership[i];
    require(upgradedAmount <= maxEmissions, "Must not send more thar
```

Of course, this function can only be called by the deployer (who is later expected to call `purgeDeployer()`) so the issue is only theoretical.

Recommend using SafeMath here or just be informed about this theoretical issue.

[L-14] `flashProof` is not effective at the start

In contract USDV, `blockDelay` is not initialized and needs to be explicitly set by calling function `setParams()`. Otherwise, it gets a default value of 0 so `flashProof` is not effective unless the value is set.

It depends on the intentions, you can initialize it in the constructor (or in the `init()` function) or maybe this precaution is intended to be turned on later.

[L-15] Token can be burn through transfer

The token can be sent to `address(0)` through a normal `transfer()` without decreasing the `totalSupply` as it would with calling `burn()` and it could cause an unintentional burn.

Recommend consider checking the recipient address to be != `address(0)`.

[L-16] You can vote for proposal still not existent

`voteProposal()` doesn't check that `proposalID <= proposalCount`.

Recommend that in `voteProposal()`, `require(proposalID <= proposalCount, "Proposal not existent")`

It should be "<=", because `proposalCount` is updated before using it (e.g. https://github.com/code-423n4/2021-04-vader/blob/main/vader-protocol/contracts/DAO.sol#L59) in this way the proposal n. 0 is not assignable, although i'm not sure if it's wanted or not.

[L-17] Out-of-bound index access in function `getAnchorPrice`

Out-of-bound index access is possible in the function `getAnchorPrice` of `Router.sol` if the number of anchors equals 1 or 2. Also, the returned anchor price is not the overall median in those situations. Router.sol#L288

Recommend consider using `arrayPrices.length/2` as the index to get the median of prices.

[L-18] ERC20 race condition for allowances

Due to the implementation of the `approve()` function in Vader.sol, Vether.sol and mainnet Vether4 at 0x4Ba6dDd7b89ed838FEd25d208D4f644106E34279, it's possible for a user to over spend their allowance in certain situations.

The steps to the attack are as follows:

1. Alice approves an allowance of 5000 VETH to Bob.
2. Alice attempts to lower the allowance to 2500 VETH.
3. Bob notices the transaction in the mempool and front-runs it by using up the full allowance with a `transferFrom` call.
4. Alice's lowered allowance is confirmed and Bob now has an allowance of 2500 VETH, which can be spent further for a total of 7500 VETH.

Overall, Bob was supposed to be approved for at most 5000 VETH but got 7500 VETH. The POC code can be found here: https://gist.github.com/toastedsteaksandwich/db32472ae5c19c2eb188f07abddd02fa

Note that in the POC, Bob receives 7492.5 VETH instead of 7500 VETH due to transfer fees.

Recommend that instead of having a direct setter for allowances, `decreaseAllowance` and `increaseAllowance` functions should be exposed which decreases and increases allowances for a recipient respectively. In this way, if the `decreaseAllowance` call is front-run, the call should revert as there is insufficient allowance to be decreased. This leaves Bob with at most 5000 VETH, the original allowance.

strictly-scarce (vader) acknowledged:

> This theoretical attack has been known about for a while but never actually observed meaningfully. Addressing it costs extra gas.

[L-19] Missing input validation may set `rewardAddress` to zero-address in Vader.sol

The function `setRewardAddress` is used by DAO to change `rewardAddress` from USDV to something else. However, there is no zero-address validation on the address. This may accidentally mint rewards to zero-address.

Recommend adding zero-address check to `setRewardAddress`.

strictly-scarce (vader) confirmed:

> Sure, if this happened (and it wasn't intentional) then it would be voted to the correct one. Low likelihood, disagree with severity.

[L-20] Default value of `curatedPoolLimit` allows only one curated pool in Router.sol

The default value of `curatedPoolLimit` only allows one curated pool at any time. This can be changed with `setParams()` but DAO does not have this functionality.

This will affect the scalability of the protocol and significantly limit the liquidity pool value proposition.

Recommend changeing `curatedPoolLimit` to a higher value on L85.

strictly-scarce (vader) acknowledged:

> Intended to be 5-10 as per discussion with community. The DAO will have extra functionality to expand.

> How is this medium-risk?

[L-21] `totalBurnt` might be wrong

Vether.sol is the 4th contract version. It takes the `totalBurnt` value of the 2nd version of the contract and continues on that. It seem more logical to use the `totalBurnt` value of the 3rd version of the contract and continue on that. This way, the value of `totalBurnt` is probably not the real value.

```
vether.sol:
contract Vether4 is ERC20 {
 constructor() public {
        ...
         vether2 = 0x01217729940055011F17BeFE6270e6E59B7d0337;
         vether3 = 0x75572098dc462F976127f59F8c97dFa291f81d8b;
         ...
         totalBurnt = VETH(vether2).totalBurnt(); totalFees = 0;
```

Recommend checking if indeed vether3 should be used and update the code to use vether3.

strictly-scarce (vader) disputed:

> Vether is deployed code and can't be changed.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements

> 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 0

[L-22] Missing DAO functionality to call `setParams()` function in USDV.sol

`setParams()` is authorized to be called only from the DAO (per modifier) but DAO contract has no corresponding functionality to call `setParams()` function. As a result, `blockDelay` — a critical parameter used to prevent flash attacks, is stuck with initialized value and cannot be changed.

Recommend adding functionality to DAO to be able to call `setParams()` of USDV.sol.

strictly-scarce (vader) commented:

> https://github.com/code-423n4/2021-04-vader-findings/issues/82

dmvt (judge) commented:

> This can be easily addressed by updating the DAO address on Vader, even if it is deployed this way. Low risk and impact as a result.

[L-23] events can be emitted even after failed transaction

When a user tries to remove liquidity or initiate a swap, their transaction may fail. But, even though the transaction fails, events can still be emitted. This could be problematic if keeping track of the record off-chain.

In Pools.sol:

- https://github.com/code-423n4/2021-04-vader/blob/main/vader-protocol/contracts/Pools.sol#L92
- https://github.com/code-423n4/2021-04-vader/blob/main/vader-protocol/contracts/Pools.sol#L101

- https://github.com/code-423n4/2021-04-vader/blob/main/vader-protocol/contracts/Pools.sol#L163

In the `_removeLiquidity()`, `swap()`, and `burnSynth()` functions; event is emitted before `transferOut()` function completes. This is because the `transferOut()` function does not check return value from transfer. As such, the transaction might fail even though the event is emitted.

Recommend checking return value from `transfer()` function in order to know whether transaction was successfully executed or not.

strictly-scarce (vader) disputed:

> Events are not used for tracking state. If a tx fails, the contract's storage will not update. The contract state is the source of truth.

> Recommend: Close, no issue found.

Mervyn853 commented:

> Our decision matrix for severity:

> 0: No-risk: Code style, clarity, off-chain monitoring (events etc), exclude gas-optimisations 1: Low Risk: UX, state handling, function incorrect as to spec 2: Funds-Not-At-Risk, but can impact the functioning of the protocol, or leak value with a hypothetical attack path with stated assumptions, but external requirements 3: Funds can be stolen/lost directly, or indirectly if a valid attack path shown that does not have handwavey hypotheticals.

> Recommended: 0

dmvt (judge) commented:

> This can and will cause cascading issues for third party dapps. At worst, completely hypothetically, this could cause a user to act as if they have funds they don't or vise versa leading to reputational and economic impact. I recommend that the team treat event emission more seriously in general.

Non-Critical Findings

- [N-01] Not always reason at require
- [N-02] Functions with implicit return values
- [N-03] Different pragma solidity
- [N-04] ERC20 specification declares decimals as uint8 type
- [N-05] [INFO] Code style suggestions
- [N-06] Difference from whitepaper
- [N-08] Public functions `getSynth()` and `isSynth()` are commented out in Factory.sol
- [N-09] Named return variable in `harvest()` and other functions of Vault.sol and contracts
- [N-10] `Protection` event not used
- [N-11] Replacing an anchor does not reset `Pool.isAnchor`
- [N-12] Unrestricted `addLiquidity` could cause unintended results on front-end apps that listen to events.

Gas Optimizations

- [G-01] Gas savings by removing unnecessary conditional in `isCurated()` function of Router.sol
- [G-02] Unused ID field in structs
- [G-03] Use immutable for constant variables
- [G-04] Public function that could be declared external
- [G-05] Gas savings by moving inited bool state variable next to an address state variable declaration in Pools.sol
- [G-06] Gas savings by removing unused state variable `_isMember` and related getter function `isMember()` in Pools.sol
- [G-07] Gas savings by declaring state variables constant in USDV.sol
- [G-08] Gas savings by converting storage variable to immutable in Vader.sol
- [G-09] Gas savings by removing state variable baseline in Vader.sol

- [G-32] Gas Optimization: Avoid Unnecessary Expensive SSTORE Calls In Vether.sol By Checking If _fee Is Non-Zero
- [G-33] Gas Optimization: Vader.sol Unnecessary Conditional
- [G-34] Gas Optimization: Utils.sol Make An Unnecessary Multiplication And Division By An Identical Value
- [G-35] Gas Optimization: DAO.sol Unnecessary Multiple Return Statements
- [G-36] Extra useless steps to calculate pooledVADER and pooledUSDV
- [G-37] variable == false -> !variable
- [G-38] Extract mappings to a common struct
- [G-39] Cache duplicate calls or storage access
- [G-40] Fee on transfer conditions
- [G-42] Not needed check for uint > 0
- [G-43] You don't need to recalculate exclusion fee every time
- [G-44] token == arrayAnchors[i]
- [G-45] Gas improvement
- [G-46] Function can be simplified
- [G-47] Function can be simplified
- [G-48] Unnecessary `else if` statement in `swapWithSynthsWithLimit`
- [G-49] Unnecessary function calls in `addLiquidity`

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