Spartan Protocol Contest Findings & Analysis Report

2021-09-16

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 Spartan smart contract system written in Solidity. The code contest took place between July 14—July 21.

Wardens

18 Wardens contributed reports to the Spartan code contest:

1. cmichel
2. jonah1005
3. 0xRajeev

4. shw
5. hickuphh3
6. gpersoon
7. 0xsanson
8. tensors
9. a_delamo
10. GalloDaSballo
11. natus
12. maplesyrup (heiho1 and thisguy__)
13. Jmukesh
14. heiho1
15. hrkrshnn
16. zer0dot
17. k
18. 7811

This contest was judged by ghoul.sol.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 63 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 13 received a risk rating in the category of HIGH severity, 15 received a risk rating in the category of MEDIUM severity, and 35 received a risk rating in the category of LOW severity.

C4 analysis also identified 44 non-critical recommendations and 25 gas optimizations.

Scope

The code under review can be found within the C4 Spartan Protocol code contest repository is comprised of 29 smart contracts written in the Solidity programming language and included 2,506 lines of Solidity code.

Severity Criteria

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

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

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

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

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

High Risk Findings (13)

[H-01] `SynthVault` withdraw forfeits rewards

Submitted by cmichel

The `SynthVault.withdraw` function does not claim the user's rewards. It decreases the user's weight and therefore they are forfeiting their accumulated rewards. The `synthReward` variable in `_processWithdraw` is also never used - it was probably intended that this variable captures the claimed rewards.

Usually, withdrawal functions claim rewards first but this one does not. A user that withdraws loses all their accumulated rewards.

Recommend claiming the rewards with the user's deposited balance first in `withdraw`.

verifyfirst (Spartan) confirmed but disagreed with severity:

> We understand there is a risk of losing unclaimed rewards if a user directly interacts with the synth-vault and not the DAPP. This is a design choice to protect the withdrawal function. We affirm the synthReward variable to be culled.

[H-02] `Pool.sol` & `Synth.sol` : Failing Max Value Allowance

Submitted by hickuphh3, also found by shw, jonah1005, 0xRajeev and cmichel

In the `_approve` function, if the allowance passed in is `type(uint256).max`, nothing happens (ie. allowance will still remain at previous value). Contract integrations (DEXes for example) tend to hardcode this value to set maximum allowance initially, but this will result in zero allowance given instead.

This also makes the comment `// No need to re-approve if already max` misleading, because the max allowance attainable is `type(uint256).max - 1`, and re-approval does happen in this case.

This affects the `approveAndCall` implementation since it uses `type(uint256).max` as the allowance amount, but the resulting allowance set is zero.

Recommend keeping it simple and removing the condition.

```
    function _approve(address owner, address spender, uint256 amount
        require(owner != address(0), "!owner");
        require(spender != address(0), "!spender");
        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
```

SamusElderg (Spartan) confirmed:

> We acknowledge the issue in the max approval for approveAndCall, which we don't use. Furthermore, the issue is only a problem if a user directly approves a maximum possible amount which would mean they are assuming trust in the contract.

> We will also change _approve in the pool and synth contracts. Risk, as outlined above, is low.

ghoul-sol (judge) commented:

> This is high risk as explained in #152

[H-03] Result of `transfer` / `transferFrom` not checked

Submitted by gpersoon, also found by jonah1005, zer0dot, 0xRajeev, JMukesh, cmichel, k, shw, 7811, heiho1 and maplesyrup.

A call to `transferFrom` or `transfer` is frequently done without checking the results. For certain ERC20 tokens, if insufficient tokens are present, no revert occurs but a result of "false" is returned. It's important to check this. If you don't, you could mint tokens without have received sufficient tokens to do so and could loose funds. Its also a best practice to check this.

Recommend always checking the result of `transferFrom` and `transfer`.

verifyfirst (Spartan) confirmed:

> The intention was to not allow non-standard tokens with non-boolean returns however in the interest of future proofing the protocol we agree with this issue

ghoul-sol (judge) commented:

> There are a lot of reported issues in relation of non-standard ERC20 and `transfer` return values. Some wardens report it all in one issue, some divided it into multiple issues. To keep playing field equal, I'll keep one issue per warden and make others invalid.

[H-04] Members lose SPARTA tokens in `removeLiquiditySingle()`

Submitted by 0xRajeev, also found by cmichel and XXX

When a member calls `removeLiquiditySingle()` requesting only SPARTA in return, i.e. `toBASE` = true, the LP tokens are transferred to the Pool to withdraw the constituent SPARTA and TOKENs back to the Router. The withdrawn TOKENs are then transferred back to the Pool to convert to SPARTA and directly transferred to the member from the Pool. However, the member's SPARTA are left behind in the Router instead of being returned along with converted SPARTA from the Pool.

In other words, the `_member`'s BASE SPARTA tokens that were removed from the Pool along with the TOKENs are never sent back to the `_member` because the `_token`'s transferred to the Pool are converted to SPARTA and only those are sent back to member directly from the Pool via `swapTo()`.

This effectively results in member losing the SPARTA component of their Pool LP tokens which get left behind in the Router and are possibly claimed by future transactions that remove SPARTA from Router.

LPs sent to Pool, SPARTA and TOKENs withdrawn from Pool to Router, TOKENs from Router sent to Pool, and TOKENs in Pool converted to BASE SPARTA and sent to member directly from the Pool.

Recommend:

1. BASE SPARTA should also be transferred to the Pool before `swapTo()` so they get sent to the member along with the converted TOKENs via `swapTo()`
2. Use `swap(BASE)` instead of `swapTo()` so that TOKENs are swapped for BASE SPARTA in Pool and sent back to ROUTER. Then send all the SPARTA from ROUTER to member.

verifyfirst (Spartan) confirmed:

> This bug was missed in a last minute edit before pushing to code423n4, wouldn't have made it past testNet testing. However, it is a good find.

[H-05] Synth `realise` is vulnerable to flash loan attacks

*Submitted by jonah1005, also found by a*delamo_

Synth `realise` function calculates `baseValueLP` and `baseValueSynth` base on AMM spot price which is vulnerable to flash loan attack. `Synth` 's lp is subject to `realise` whenever the AMM ratio is different than Synth's debt ratio.

The attack does not necessarily require a flash loan. A big whale of the lp token holders could keep calling `realise` by shifting token ratio of AMM pool back and forth.

The vulnerability is located at `Synth.sol` L187-L199. Where the formula here is dangerous.

Here's a script for conducting flashloan attack:

```
flashloan_amount = init_amount
user = w3.eth.accounts[0]
marked_token.functions.transfer(user, flashloan_amount).transact
marked_token.functions.transfer(token_pool.address, flashloan_am
token_pool.functions.addForMember(user).transact({'from': user})
received_lp = token_pool.functions.balanceOf(user).call()
synth_balance_before_realise = token_synth.functions.mapSynth_LP
token_synth.functions.realise(token_pool.address).transact()
token_pool.functions.transfer(token_pool.address, received_lp).t
token_pool.functions.removeForMember(user).transact({'from': use
token_synth.functions.realise(token_pool.address).transact()
synth_balance_after_realise = token_synth.functions.mapSynth_LPP
print('synth_lp_balance_after_realise', synth_balance_after_real
print('synth_lp_balance_before_realise', synth_balance_before_re
```

Output:

```
synth_balance_after_realise 131785996482931390816 2
synth_balance_before_realise 2063953488372093023256
```

Calculating Lp token's value base on AMM protocol is known to be dangerous. There are a few steps that might solve the issue:

1. calculate token's price from a reliable source. Implement a TWAP oracle or uses chainlink oracle.
2. calculate lp token value based on anti-flashloan formula. Alpha finance's formula is a good reference: https://blog.alphafinance.io/fair-lp-token-pricing

verifyfirst (Spartan) confirmed and disagreed with severity:

> A proposal has been suggested to limit the use of realise() for a DAO proposal. This will allow only liquidity providers to choose the outcome of a function that directly affects them.

[H-06] `SynthVault` rewards can be gamed

Submitted by cmichel

The `SynthVault._deposit` function adds `weight` for the user that depends on the spot value of the deposit synth amount in `BASE`.

This spot price can be manipulated and the cost of manipulation is relative to the pool's liquidity. However, the reward (see `calcReward`) is measured in BASE tokens unrelated to the pool. Therefore, if the pool's liquidity is low and the reward reserve is high, the attack can be profitable:

1. Manipulate the pool spot price of the `iSYNTH(_synth).LayerONE()` pool by dripping a lot of `BASE` into it repeatedly (sending lots of smaller trades is less costly due to the path-independence of the continuous liquidity model). This increases the `BASE` per `token` price.
2. Call `SynthVault.depositForMember` and deposit a *small* amount of synth token. The `iUTILS(_DAO().UTILS()).calcSpotValueInBase(iSYNTH(_synth).LayerONE(), _amount)` will return an inflated weight due to the price.
3. Optionally drip more `BASE` into the pool and repeat the deposits
4. Drip back `token` to the pool to rebalance it

The user's `weight` is now inflated compared to the deposited / locked-up amount and they can claim a large share of the rewards. The cost of the attack depends on the pool's liquidity and the profit depends on the reserve. It could therefore be profitable under certain circumstances.

Recommend tracking a TWAP price of the synth instead, store the deposited synths instead, and compute the weight & total weight on the fly based on the TWAP * deposit amount instead of at the time of deposit.

verifyfirst (Spartan) acknowledged:

> There is already a discussion in place to change spot rate to swap rate calculation for weights.

[H-07] Missing slippage checks

Submitted by cmichel, also found by tensors

There are no minimum amounts out, or checks that frontrunning/slippage is sufficiently mitigated. This means that anyone with enough capital can force arbitrarily large slippage by sandwiching transactions, close to 100%. See issue page for referenced code.

Recommend adding a minimum amount out parameter. The function reverts if the minimum amount isn't obtained.

verifyfirst (Spartan) acknowledge:

> We acknowledge the issue for the protocol's AMM, but if this becomes a large issue in the future, the router is easily upgradeable to include a minimum rate parameter.

SamusEldburg (Spartan) confirmed and disagreed with severity:

> Have changed this to confirmed; even though we already were aware of it; we have discussed and are happy to add in a UI-handed arg for minAmount now rather than reactively in the future. Disagree with severity though; this wasn't a problem with V1 at all.

ghoul-sol (Judge) commented:

> I'll keep high risk as sandwich attacks are very common and risk of getting a bad swap is real.

[H-08] Dividend reward can be gamed

Submitted by cmichel

The `Router.addDividend` function tells the reserve to send dividends to the pool depending on the fees.

- The attacker provides LP to a curated pool. Ideally, they become a large LP holder to capture most of the profit, they should choose the smallest liquidity pool as the dividends are pool-independent.
- The `normalAverageFee` variable that determines the pool dividends can be set to zero by the attacker by trading a single wei in the pool `arrayFeeSize` (20) times (use `buyTo`). The fees of the single wei trades will be zero and thus the `normalAverageFee` will also be zero as, see `addTradeFee`.
- The attacker then does a trade that generates some non-zero fees, setting the `normalAverageFee` to this trade's fee. The `feeDividend` is then computed as `_fees * dailyAllocation / (_fees + normalAverageFee) = _fees * dailyAllocation / (2 * _fees) = dailyAllocation / 2`. Half of the `dailyAllocation` is sent to the pool.
- The attacker repeats the above steps until the reserve is almost empty. Each time the `dailyAllocation` gets smaller but it's still possible to withdraw almost all of it.
- They redeem their LP tokens and gain a share of the profits

The reserve can be emptied by the attacker.

Counting only the last 20 trades as a baseline for the dividends does not work. It should probably average over a timespan but even that can be gamed if it is too short. I think a better idea is to compute the dividends based on **volume** traded over a timespan instead of looking at individual trades.

verifyfirst (Spartan) acknowledged:

> Only very deep pools will be curated for dividends. Variables can be changed reactively to alter the dividends. Whilst we were aware of this and feel the attack is limited its sparked some discussion for some new ideas to solve this.

ghoul-sol (judge) commented:

> Keeping high risk as the report is valid

[H-09] arbitrary synth mint/burn from pool

Submitted by jonah1005

`Pool` can mint arbitrary `Synth` provided as long as it's a valid synth. When there are multiple curated pools and synth (which the protocol is designed for), hackers can mint expensive synthetics from a cheaper AMM pool. The hacker can burn the minted synth at the expensive pool and get profit. The arbitrage profit can be amplified with flash loan services and break all the pegs.

Pool's mintSynth logic, Synth's mintSynth logic, and Synth's authorization logic.

The price of the synthetics to be mint is calculated in `Pool` based on the AMM price of the current Pool

Here's a web3.py script of minting arbitrary `Synth` in a pool. For simplicity, two pools are set with the assumption that link is 10x expensive than dai.

```
sparta_amount = 100 * 10**18
initail_link_synth = link_synth.functions.balanceOf(user).call()
base.functions.transfer(link_pool.address, sparta_amount).transa
link_pool.functions.mintSynth(link_synth.address, user).transact
after_link_synth = link_synth.functions.balanceOf(user).call()

print('get link synth amount from link pool:', after_link_synth

sparta_amount = 100 * 10**18
initail_link_synth = link_synth.functions.balanceOf(user).call()
base.functions.transfer(dai_pool.address, sparta_amount).transac
dai_pool.functions.mintSynth(link_synth.address, user).transact(
after_link_synth = link_synth.functions.balanceOf(user).call()
```

```
print('get link synth amount from dai pool:', after_link_synth -
```

The log of the above script

```
get link synth amount from link pool: 97078046905036524413
get link synth amount from dai pool: 97078046905036524136
```

Recommend Checking the provided synth's underlying token in `mintSynth`

```
require(iSYNTH(synthOut).LayerONE() == TOKEN, "invalid synth");
```

verifyfirst (Spartan) confirmed:

> We agree and appreciate this finding being valid high risk issue.

[H-10] Hijack token pool by burning liquidity token

Submitted by jonah1005

`Pool` allows users to burn lp tokens without withdrawing the tokens. This allows the hacker to mutate the pools' rate to a point that no one can get any lp token anymore (even if depositing token).

The liquidity tokens are calculated at `Utils:calcLiquidityUnits`

```
// units = ((P (t B + T b))/(2 T B)) * slipAdjustment
// P * (part1 + part2) / (part3) * slipAdjustment
uint slipAdjustment = getSlipAdustment(b, B, t, T);
uint part1 = t*(B);
uint part2 = T*(b);
uint part3 = T*(B)*(2);
uint _units = (P * (part1 + (part2))) / (part3);
return _units * slipAdjustment / one;  // Divide by 10**18
```

where `P` stands for `totalSupply` of current Pool. If `P` is too small (e.g, 1) then all the units would be rounding to 0.

Since any person can create a `Pool` at `PoolFactory`, hackers can create a Pool and burn his lp and set `totalSupply` to 1. He will be the only person who owns the Pool's lp from now on. Pool's burn logic and Utils' lp token formula.

Here's a script of a user depositing 1M token to a pool where `totalSupply` equals 1

```
dai_pool.functions.burn(init_amount-1).transact()
print('total supply', dai_pool.functions.totalSupply().call())
dai.functions.transfer(dai_pool.address, 1000000 * 10**18).trans
dai_pool.functions.addForMember(user).transact()
print('lp received from depositing 1M dai: ', dai_pool.functions
```

Output:

```
total supply 1
lp received from depositing 1M dai:  0
```

Recommend removing `burn` or restrict it to privileged users only.

verifyfirst (Spartan) confirmed:

> We agree to this issue and will restrict access to burn in the pool contract. We have already proposed adding a 1 week withdraw coolOff for all users per pool from the genesis of creation. Users can only add liquidity within this period.

[H-11] Misuse of AMM model on minting `Synth` (resubmit to add more detail)

Submitted by jonah1005

`Pool` calculates the amount to be minted based on `token_amount` and `sparta_amount` of the Pool. However, since `token_amount` in the pool would not

decrease when users mint `Synth`, it's always cheaper to mint `synth` than swap the tokens.

The synthetics would be really hard to be on peg. Or, there would be a flash-loan attacker to win all the arbitrage space.

In **Pool's mint** `synth`, The `synth` amount is calculated at L:232

```
uint output = iUTILS(_DAO().UTILS()).calcSwapOutput(_actualInput
```

which is the same as swapping base to token at L:287

```
uint256 _X = baseAmount;
uint256 _Y = tokenAmount;
_y =  iUTILS(_DAO().UTILS()).calcSwapOutput(_x, _X, _Y); // Calc
```

However, while swapping tokens decrease pool's token, mint just mint it out of the air.

Here's a POC: Swap sparta to token for ten times

```
for i in range(10):
    amount = 10 * 10**18
    transfer_amount = int(amount/10)
    base.functions.transfer(token_pool.address, transfer_amount)
    token_pool.functions.swapTo(token.address, user).transact()
```

Mint `Synth` for ten times

```
for i in range(10):
    amount = 10 * 10**18
    transfer_amount = int(amount/10)
    base.functions.transfer(token_pool.address, transfer_amount)
    token_pool.functions.mintSynth(token_synth.address, user).tr
```

The Pool was initialized with 10000:10000 in both cases. While the first case(swap token) gets `4744.4059` and the second case gets `6223.758`.

The debt should be considered in the AMM pool so I recommend to maintain a debt variable in the Pool and use `tokenAmount - debt` when the Pool calculates the token price. Here's some idea of it:

```
uint256 public debt;
function _tokenAmount() returns (uint256) {
    return tokenAmount - debt;
}

// Swap SPARTA for Synths
function mintSynth(address synthOut, address member) external re
    require(iSYNTHFACTORY(_DAO().SYNTHFACTORY()).isSynth(synthOu
    uint256 _actualInputBase = _getAddedBaseAmount(); // Get rec

    // Use tokenAmount - debt to calculate the value
    uint output = iUTILS(_DAO().UTILS()).calcSwapOutput(_actualI

    // increment the debt
    debt += output

    uint _liquidityUnits = iUTILS(_DAO().UTILS()).calcLiquidityU
    _incrementPoolBalances(_actualInputBase, 0); // Update recor
    uint _fee = iUTILS(_DAO().UTILS()).calcSwapFee(_actualInputB
    fee = iUTILS(_DAO().UTILS()).calcSpotValueInBase(TOKEN, _fee
    _mint(synthOut, _liquidityUnits); // Mint the LP tokens dire
    iSYNTH(synthOut).mintSynth(member, output); // Mint the Synt
    _addPoolMetrics(fee); // Add slip fee to the revenue metrics
    emit MintSynth(member, BASE, _actualInputBase, TOKEN, output
    return (output, fee);
}
```

verifyfirst (Spartan) confirmed:

> We agree with the issue submitted, discussions are already in progress around ensuring the mint rate considers the floating debt. Potential high risk, however, hard to create a scenario to prove this.

[H-12] wrong `calcLiquidityHoldings` that leads to dead fund in the Pool

Submitted by jonah1005

The lptoken minted by the `Pool` contract is actually the mix of two types of tokens. One is the original lptokens user get by calling `addForMember`. This lpToken is similar to lp of Uniswap, Crv, Sushi, ... etc. The other one is the debt-lp token the Synth contract will get when the user calls `mintSynth`. The `Synth` contract can only withdraw `Sparta` for burning debt-lp. Mixing two types of lp would raise several issues.

LP user would not get their fair share when they burn the lP.

1. Alice adds liquidity with Sparta 1000 and token B 1000 and create a new Pool.
2. Bob mint Synth with 1000 Sparta and get debt.
3. Alice withdraw all lp Token
4. Bob burn all Synth.

The pool would end up left behind a lot of token B in the Pool while there's no lp holder.

I would say this is a high-risk vulnerability since it pauses unspoken risks and losses for all users (all the time)

The logic of burn original lp and burn debt-lp.

I do not know whether this is the team's design choice or its composite with a series of bugs. If this is the original design, I do not come up with a fix. It's a bit similar to the impermanent loss. However, the loss would be left behind in the Pool. This is more complicated and maybe worse than the impermanent loss. If this is the design choice, I think it's worth emphasize and explain to the users.

verifyfirst (Spartan) confirmed and disagreed with severity:

> We are in discussion of a viable solution to limit the effects of a bank run. One example is limiting the minted synths based on the depth of its underlying pool.

SamusElderg (Spartan) commented:

> LP units are only used for accounting; even if they were drained to zero or vice versa on the synth contract; they would result in the same redemption value when burning. Hence the risk is low; however there is already discussions on implementing controls to synths including a maximum synthSupply vs tokenDepth ratio to prevent top-heavy synths ontop of the pools which isn't really specific to the warden's scenario; however does help limit those 'unknowns' that the warden addressed.

[H-13] Flash loan manipulation on `getPoolShareWeight` of `Utils`

Submitted by shw

The `getPoolShareWeight` function returns a user's pool share weight by calculating how many SPARTAN the user's LP tokens account for. However, this approach is vulnerable to flash loan manipulation since an attacker can swap a large number of TOKEN to SPARTAN to increase the number of SPARTAN in the pool, thus effectively increasing his pool share weight.

According to the implementation of `getPoolShareWeight`, a user's pool share weight is calculated by `uints * baseAmount / totalSupply`, where `uints` is the number of user's LP tokens, `totalSupply` is the total supply of LP tokens, and `baseAmount` is the number of SPARTAN in the pool. Thus, a user's pool share weight is proportional to the number of SPARTAN in the pool. Consider the following attack scenario:

1. Supposing the attacked pool is SPARTAN-WBNB. The attacker first prepares some LP tokens (WBNB-SPP) by adding liquidity to the pool.
2. The attacker then swaps a large number of WBNB to SPARTAN, which increases the pool's `baseAmount`. He could split his trade into small amounts to reduce slip-based fees.
3. The attacker now wants to increase his weight in the `DaoVault`. He adds his LP tokens to the pool by calling the `deposit` function of `Dao`.

4. `Dao` then calls `depositLP` of `DaoVault`, causing the attacker's weight to be recalculated. Due to the large proportion of SPARTAN in the pool, the attacker's weight is artificially increased.
5. With a higher member weight, the attacker can, for example, vote the current proposal with more votes than he should have or obtain more rewards when calling `harvest` of the `Dao` contract.
6. The attacker then swaps back SPARTAN to WBNB and only loses the slip-based fees.

Referenced code: Utils.sol#L46-L50, Utils.sol#L70-L77, DaoVault.sol#L44-L56, Dao.sol#L201, and Dao.sol#L570.

A possible mitigation is to record the current timestamp when a user's weight in the `DaoVault` or `BondVault` is recalculated and force the new weight to take effect only after a certain period, e.g., a block time. This would prevent the attacker from launching the attack since there is typically no guarantee that he could arbitrage the WBNB back in the next block.

SamusElderg (Spartan) confirmed and disagreed with severity:

> Recommended mitigation has been included in contributors ongoing discussions to make this more resistant to manipulation

ghoul-sol (judge) commented:

> Keeping high risk because of impact

Medium Risk Findings (14)

[M-01] `Dao.sol` : Insufficient validation for proposal creation

Submitted by hickuphh3, 0xRajeev, also found by gpersoon and shw

In general, creating invalid proposals is easy due to the lack of validation in the `new*Proposal()` functions.

- The `typeStr` is not validated at all. For example, one can call `newActionProposal()` with `typeStr = ROUTER` or `typeStr = BAD_STRING`, both of which will pass. The first will cause `finaliseProposal()` to fail because the proposed address is null, preventing `completeProposal()` from executing. The second does nothing because it does not equate to any of the check `typeStr`, and so `completeProposal()` isn't executed at all.
- Not checking the proposed values are null. The checks only happen in `finaliseProposal()` when the relevant sub-functions are called, like the `move*()` functions.

All of these scenarios lead to a mandatory 15 day wait since proposal creation in order to be cancelled, which prevents the creation of new proposals (in order words, denial of service of the DAO).

Recommended Steps:

1. Since the number of proposal types is finite, it is best to restrict and validate the `typeStr` submitted. Specifically,
    - `newActionProposal()` should only allow `FLIP_EMISSIONS` and `GET_SPARTA` proposal types
    - `newAddressProposal()` should only allow `DAO`, `ROUTER`, `UTILS`, `RESERVE`, `LIST_BOND`, `DELIST_BOND`, `ADD_CURATED_POOL` and `REMOVE_CURATED_POOL` proposal types
    - `newParamProposal()` should only allow `COOL_OFF` and `ERAS_TO_EARN` proposal types
2. Perhaps have a "catch-all-else" proposal that will only call `_completeProposal()` in `finaliseProposal()`

```
function finaliseProposal() external {
        ...
        } else if (isEqual(_type, 'ADD_CURATED_POOL')){
                _addCuratedPool(currentProposal);
  } else if (isEqual(_type, 'REMOVE_CURATED_POOL')){
    _removeCuratedPool(currentProposal);
  } else {
```

```
                completeProposal(_proposalID);
        }
```

3. Do null validation checks in `newAddressProposal()` and `newParamProposal()`

```
function newAddressProposal(address proposedAddress, string memo
    require(proposedAddress != address(0), "!address");
                // TODO: validate typeStr
                ...
}

function newParamProposal(uint32 param, string memory typeStr) e
    require(param != 0, "!param");
                // TODO: validate typeStr
                ...
}
```

verifyfirst (Spartan) confirmed and disagreed with severity:

> A valid issue in reducing annoyance for DAO proposals. Suggested mitigation fixes the issue. Medium Severity.

ghoul-sol (judge) commented:

> The only attack vector here is DoS attack which could be fought by the community by pooling funds for flashbot TX that front runs the attacker. While inconvenient, it doesn't stop the protocol for long. Agree with sponsor about severity.

[M-02] Missleading `onlyDAO` modifiers

Submitted by cmichel, 0xRajeev, 0xsanson, gpersoon, also found by hickuphh3 and shw

Several contracts implement an `onlyDAO` modifier which, as the name suggests, should only authorize the function to be executed by the DAO. However, some

implementations are wrong and either allow the DAO or the deployer to execute, or even only the deployer:

Incorrect implementations:

- `BondVault.onlyDAO` : allows deployer + DAO
- `DAO.onlyDAO` : allows deployer
- `DAOVault.onlyDAO` : allows deployer + DAO
- `poolFactory.onlyDAO` : allows deployer + DAO
- `Router.onlyDAO` : allows deployer + DAO
- `Synth.onlyDAO` : allows deployer
- `synthFactory.onlyDAO` : allows deployer
- `synthVault.onlyDAO` : allows deployer + DAO

In all of these functions, the deployer may execute the function as well which is a centralization risk. The deployer can only sometimes be purged, as in `synthFactory` , in which case nobody can execute these functions anymore.

Recommend renaming it to `onlyDeployer` or `onlyDeployerOrDAO` depending on who has access.

verifyfirst (Spartan) acknowledged:

> This is by design a choice. However, there are current discussions around renaming the high level access modifiers to be more descriptive in their purpose.

ghoul-sol (judge) commented:

> This is a non-critical issue because there's no in-code bugs, it's rather error-prone naming.

ghoul-sol (judge) commented:

> On second look, I'll keep it a medium risk as deployer cannot be purged in all contracts which introduces systemic risk.

[M-03] Improper access control of `claimAllForMember` allows anyone to reduce the weight of a member

Submitted by shw, also found by 0xRajeev

The `claimAllForMember` function of `Dao` is permissionless, allowing anyone to claim the unlocked bonded LP tokens for any member. However, claiming a member's LP tokens could decrease the member's weight in the `BondVault`, thus affecting the member's votes and rewards in the `Dao` contract.

For example, an attacker can intentionally front-run a victim's `voteProposal` call to decrease the victim's vote weight to prevent the proposal from being finalized:

1. Supposing the victim's member weight in the `BondVault` is 201, the total weight is 300. The victim has some LP tokens claimable from the vault, and if claimed, the victim's weight will be decreased to 101. To simplify the situation, assuming that the victim's weight in the `DaoVault` and the total weight of the `DaoVault` are both 0.
2. The victim wants to vote on the current proposal, which requires the majority consensus. If the victim calls `voteProposal`, the proposal will be finalized since the victim has the majority weight (201/300 > 66.6%).
3. An attacker does not want the proposal to be finalized, so he calls `claimAllForMember` with the victim as the parameter to intentionally decrease the victim's weight.
4. As a result, the victim's weight is decreased to 101, and the total weight is decreased to 200. The victim cannot finalize the proposal since he has no majority anymore (101/200 < 66.6%).

Similarly, an attacker can front-run a victim's `harvest` call to intentionally decrease the victim's reward since the amount of reward is calculated based on the victim's current weight.

See issue page page for referenced code

Consider removing the `member` parameter in the `claimAllForMember` function and replace all `member` to `msg.sender` to allow only the user himself to claim unlocked

bonded LP tokens.

verifyfirst (Spartan) confirmed and disagreed with severity:

> Although a low risk issue, it is valid and the suggested mitigation is correctly proposed.

ghoul-sol (judge) commented:

> Making this medium risk as no funds are lost.

[M-04] _deposit resetting user rewards can be used to grief them and make them loose rewards via `depositForMember`

Submitted by GalloDaSballo

The function `_deposit` sets `mapMemberSynth_lastTime` to a date in the future in `synthVault.sol` L107.

`mapMemberSynth_lastTime` is also used to calculate rewards earned.
`depositForMember` allows anyone, to "make a donation" for the member and cause that member to lose all their accrued rewards. This can't be used for personal gain, but can be used to bring misery to others.

`depositForMember` (in `synthVault.sol` on L95 can be called by anyone.

This will set the member and can be continuously exploited to make members never earn any reward.

```
mapMemberSynth_lastTime[_member][_synth] = block.timestamp + mi
```

This is the second submission under the same exploit.

This can be mitigated by harvesting for the user right before changing `mapMemberSynth_lastTime[_member][_synth]`

verifyfirst (Spartan) commented:

> Suggested mitigation solves the issue.

[M-05] Pools can be created without initial liquidity

Submitted by cmichel

The protocol differentiates between public pool creations and private ones (starting without liquidity). However, this is not effective as anyone can just flashloan the required initial pool liquidity, call `PoolFactory.createPoolADD`, receive the LP tokens in `addForMember` and withdraw liquidity again.

Recommend considering burning some initial LP tokens or taking a pool creation fee instead.

SamusElderg (Spartan) confirmed:

> Whilst we were aware of this (more of a deterrent than prevention) contributors have discussed some methods of locking this liquidity in and making it at least flash loan resistant. For instance, a withdraw-lock (global by pool) for 7 days after the pool's genesis so that no user can withdraw liquidity until 7 days have passed. There are other ideas floating around too; but regardless this issue will be addressed in some way prior to launch

[M-06] Pool: `approveAndCall` sets unnecessary approval

Submitted by cmichel

The `Pool.approveAndCall` function approves the `recipient` contract with the max value instead of only the required `amount`.

For safety, the approval should not be set to the max value, especially if the amount that the contract may use is already known in this call, like this is the case for `approveAndCall`.

Recommend only approving `amount`.

[M-07] Synth: `approveAndCall` sets unnecessary approval

Submitted by cmichel

The `Synth.approveAndCall` function approves the `recipient` contract with the max value instead of only the required `amount`.

For safety, the approval should not be set to the max value, especially if the amount that the contract may use is already known in this call, like this is the case for `approveAndCall`.

Recommend only approving `amount`.

[M-08] `SynthVault` deposit lockup bypass

Submitted by cmichel

The `SynthVault.harvestSingle` function can be used to mint & deposit synths without using a lockup. An attacker sends `BASE` tokens to the pool and then calls `harvestSingle`. The inner `iPOOL(_poolOUT).mintSynth(synth, address(this));` call will mint synth tokens to the vault based on the total `BASE` balance sent to the pool, including the attacker's previous transfer. They are then credited the entire amount to their `weight`.

This essentially acts as a (mint +) deposit without a lock-up period.

Recommend syncing the pool before sending `BASE` to it through `iRESERVE(_DAO().RESERVE()).grantFunds(reward, _poolOUT);` such that any previous `BASE` transfer is wasted. This way only the actual reward's weight is increased.

verifyfirst (Spartan) disputed:

> Although this is true, the attacker is not benefiting from any gain. They are only minting extra synths into the synthVault into their weight. It is no different to - minting and then staking into the vault.

SamusElderg (Spartan) confirmed:

> @verifyfirst (Spartan) in my opinion this one should be confirmed and the recommended mitigation also makes sense; any attempt to send in BASE by a bad actor can be attributed to the existing LPers instead

[M-09] In the beginning its relatively easy to gain majority share

Submitted by gpersoon

When the DAO is just deployed it is relatively easy to gain a large (majority) share, by depositing a lot in the `DAOVault` and/of `BONDVault`. Then you could submit a proposal and vote it in. Luckily there is a `coolOffPeriod` of 3 days. But if others are not paying attention in these 3 days you might get your vote passed by voting for it with your majority share. The riskiest proposal would be to replace the DAO (moveDao), because that way you could take over everything.

Recommend pay attention to the proposals when the DAO is just deployed In Dao.sol and making sure you initially have a majority vote.

verifyfirst (Spartan) acknowledged:

> The recommended mitigation steps were already going to be in place for mainNet including emissions etc.

[M-10] `grantFunds` will revert after a DAO upgrade.

Submitted by gpersoon

When the DAO is upgraded via `moveDao`, it also updates the DAO address in BASE. However it doesn't update the DAO address in the `Reserve.sol` contract. This could be done with the function `setIncentiveAddresses(..)`

Now the next time `grantFunds` of `DAO.sol` is called, its tries to call `_RESERVE.grantFunds(...)`

The `grantFunds` of `Reserve.sol` has the modifier `onlyGrantor()`, which checks the msg `.sender` == DAO. However in the mean time, the DAO has been updated and `Reserve.sol` doesn't know about it and thus the modifier will not allow access to the function. Thus `grantFunds` will revert.

`Dao.sol` **L452**

```
    function moveDao(uint _proposalID) internal {
        address _proposedAddress = mapPID_address[_proposalID]; // G
        require(_proposedAddress != address(0), "!address"); // Prop
        DAO = _proposedAddress; // Change the DAO to point to the ne
        iBASE(BASE).changeDAO(_proposedAddress); // Change the BASE
        daoHasMoved = true; // Set status of this old DAO
        completeProposal(_proposalID); // Finalise the proposal
    }

    function grantFunds(uint _proposalID) internal {
        uint256 _proposedAmount = mapPID_param[_proposalID]; // Get
        address _proposedAddress = mapPID_address[_proposalID]; // G
        require(_proposedAmount != 0, "!param"); // Proposed grant a
        require(_proposedAddress != address(0), "!address"); // Prop
        _RESERVE.grantFunds(_proposedAmount, _proposedAddress); // G
        completeProposal(_proposalID); // Finalise the proposal
    }
```

`Reserve.sol` **L17**

```
    modifier onlyGrantor() {
        require(msg.sender == DAO || msg.sender == ROUTER || msg.ser
        _;
    }

    function grantFunds(uint amount, address to) external onlyGrantc
        ....
    }

    function setIncentiveAddresses(address _router, address _lend, a
        ROUTER = _router;
        LEND = _lend;
        SYNTHVAULT = _synthVault;
        DAO = _Dao;
```

```
}
```

Recommend calling `setIncentiveAddresses(..)` when a DAO upgrade is done.

verifyfirst (Spartan) confirmed:

> Non critical, however this will be implemented to future proof the protocol

[M-11] Block usage of `addCuratedPool`

Submitted by gpersoon, also found by hickuphh3, and cmichel

The function `curatedPoolCount()` contains a for loop over the array `arrayPools`. If `arrayPools` would be too big then the loop would run out of gas and `curatedPoolCount()` would revert. This would mean that `addCuratedPool()` cannot be executed anymore (because it calls `curatedPoolCount()`)

The array `arrayPools` can be increased in size arbitrarily by repeatedly doing the following:

- create a pool with `createPoolADD()` (which requires 10,000 SPARTA)
- empty the pool with `remove()` of Pool.sol, which gives back the SPARTA tokens These actions will use gas to perform.

```
45
46 function createPoolADD(uint256 inputBase, uint256 inputToken,
47     require(getPool(token) == address(0)); // Must be a valid
48     require((inputToken > 0 && inputBase >= (10000*10**18)),
49     Pool newPool; address _token = token;
50     if(token == address(0)){_token = WBNB;} // Handle BNB ->
51     require(_token != BASE && iBEP20(_token).decimals() == 18
52     newPool = new Pool(BASE, _token); // Deploy new pool
53     pool = address(newPool); // Get address of new pool
54     mapToken_Pool[_token] = pool; // Record the new pool addr
55     _handleTransferIn(BASE, inputBase, pool); // Transfer SPA
56     _handleTransferIn(token, inputToken, pool); // Transfer T
57     arrayPools.push(pool); // Add pool address to the pool ar
58     ..
```

```
59
60 function curatedPoolCount() internal view returns (uint){
61     uint cPoolCount;
62     for(uint i = 0; i< arrayPools.length; i++){
63         if(isCuratedPool[arrayPools[i]] == true){
64             cPoolCount += 1;
65         }
66     }
67     return cPoolCount;
68 }
```

```
function addCuratedPool(address token) external onlyDAO {
    ...
    require(curatedPoolCount() < curatedPoolSize, "maxCurated");
```

```
187
188 function remove() external returns (uint outputBase, uint ou
189     return removeForMember(msg.sender);
190 }
191
192 // Contract removes liquidity for the user
193 function removeForMember(address member) public returns (uin
194     uint256 _actualInputUnits = balanceOf(address(this)); //
195     outputBase = iUTILS(_DAO().UTILS()).calcLiquidityHolding
196     outputToken = iUTILS(_DAO().UTILS()).calcLiquidityHoldin
197     _decrementPoolBalances(outputBase, outputToken); // Upda
198     _burn(address(this), _actualInputUnits); // Burn the LP
199     iBEP20(BASE).transfer(member, outputBase); // Transfer t
200     iBEP20(TOKEN).transfer(member, outputToken); // Transfer
201     emit RemoveLiquidity(member, outputBase, outputToken, _a
202     return (outputBase, outputToken);
203 }
```

Recommend creating a variable `curatedPoolCount` and increase it in `addCuratedPool` and decrease it in `removeCuratedPool`.

verifyfirst (Spartan) confirmed and disagreed with severity:

> We agree with the issue, this could be more efficient.

[M-12] `BondVault.sol` : Possibly unwithdrawable bondedLP funds in `claimForMember()` + `claimRate` never zeros after full withdrawals

Submitted by hickuphh3, also found by 0xRajeev

A host of problems arise from the L110-113 of the `claimForMember()` function, where `_claimable` is deducted from the bondedLP balance before the condition check, when it should be performed after (or the condition is changed to checking if the remaining bondedLP balance to zero).

```
113
114 mapBondAsset_memberDetails[asset].bondedLP[member] -= _claim
115 if(_claimable == mapBondAsset_memberDetails[asset].bondedLP[
116         mapBondAsset_memberDetails[asset].claimRate[member]
117 }
```

1. Permanently Locked Funds

If a user claims his bonded LP asset by calling `dao.claimForMember()`, or a malicious attacker helps a user to claim by calling `dao.claimAllForMember()`, either which is done such that `_claimable` is exactly half of his remaining bondedLP funds of an asset, then the other half would be permanently locked.

- Assume `mapBondAsset_memberDetails[asset].bondedLP[member] = 2 * _claimable`
- L110: `mapBondAsset_memberDetails[asset].bondedLP[member] = _claimable`
- L111: The if condition is satisfied
- L112: User's claimRate is erroneously set to 0 ⇒ `calcBondedLP()` will return 0, ie. funds are locked permanently

2. Claim Rate Never Zeroes For Final Claim

On the flip side, should a user perform a claim that enables him to perform a full withdrawal (ie. `_claimable =`

`mapBondAsset_memberDetails[asset].bondedLP[member]`, we see the following effects:

- L110: `mapBondAsset_memberDetails[asset].bondedLP[member] = 0`
- L111: The if condition is not satisfied, L112 does not execute, so the member's claimRate for the asset remains non-zero (it is expected to have been set to zero).

Thankfully, subsequent behavior remains as expected since `calcBondedLP` returns zero as `claimAmount` is set to the member's bondedLP balance (which is zero after a full withdrawal).

The `_claimable` deduction should occur after the condition check. Alternatively, change the condition check to `if (mapBondAsset_memberDetails[asset].bondedLP[member] == 0)`.

verifyfirst (Spartan) confirmed:

> Good find, suggested mitigation solves the potential to lock funds.

[M-13] Vulnerable Pool initial rate.

Submitted by jonah1005

`Pool` is created in function `createPoolADD`. The price (rate) of the token is determined in this function. Since the address is deterministic, the attacker can front-run the `createPoolADD` transaction and sends tokens to Pool's address. This would make the pool start with an extreme price and create a huge arbitrage space.

I assume pools would be created by the deployer rather than DAO at the early stage of the protocol. If the deployer calls `createPoolADD` and `addCuratedPool` at the same time then an attacker/arbitrager could actually get (huge) profit by doing this.

Assume that the deployer wants to create a BNB pool at an initial rate of 10000:300 and then make it a curated pool. An arbitrager can send 2700 BNB to the (precomputed) pool address and make iBNB 10x cheaper. The arbitrager can mint the synth at a 10x cheaper price before the price becomes normal.

poolFactory.sol L45-L62

```
pre_computed_dai_pool_address = '0x1007BDBA1BCc3C94e20d57768EF9f
dai.functions.transfer(pre_computed_dai_pool_address, initial_am
initial_amount = 10000*10**18
dai.functions.approve(pool_factory.address, initial_amount*10).t
base.functions.approve(pool_factory.address, initial_amount).tra
pool_factory.functions.createPoolADD(initial_amount, initial_amo
```

Recommend adding:

```
require(iBEP20(BASE).balanceOf(address(pool)) == 0 && iBEP20(to
```

In `createPoolADD`

verifyfirst (Spartan) acknowledged:

> Curated pools will only be added once the community feel a pool is deep enough in Liquidity. Other safeguard measures are also in place for this scenario which probably weren't documented well enough.

[M-14] BondVault `BASE` incentive can be gamed

Submitted by cmichel

`BondVault` deposits match *any* deposited `token` amount with the `BASE` amount to provide liquidity, see Docs and `DAO.handleTransferIn`. The matched `BASE` amount is the **swap amount** of the `token` trade in the pool. An attacker can manipulate the pool and have the `DAO` commit `BASE` at bad prices which they then later buys back to receive a profit on `BASE`. This is essentially a sandwich attack abusing the fact that one can trigger the `DAO` to provide `BASE` liquidity at bad prices:

1. Manipulate the pool spot price by dripping a lot of `BASE` into it repeatedly (sending lots of smaller trades is less costly due to the path-independence of the continuous liquidity model). This increases the `token` per `BASE` price.

2. Repeatedly call `DAO.bond(amount)` to drip `tokens` into the `DAO` and get matched with `BASE` tokens to provide liquidity. (Again, sending lots of smaller trades is less costly.) As the pool contains low `token` but high `BASE` reserves, the `spartaAllocation = _UTILS.calcSwapValueInBase(_token, _amount)` swap value will be high. **The contract sends even more BASE to the pool to provide this liquidity.**
3. Unmanipulate the pool by sending back the `tokens` from 1. As a lot more `BASE` tokens are in the reserve now due to the DAO sending it, the attacker will receive more `BASE` as in 1. as well, making a profit

The DAO's Bond allocation can be stolen. The cost of the attack is the trade fees in 1. + 3. as well as the tokens used in 2. to match the `BASE` , but the profit is a share on the `BASE` supplied to the pool by the DAO in 2.

Track a TWAP spot price of the `TOKEN <> BASE` pair and check if the `BASE` incentive is within a range of the TWAP. This circumvents that the `DAO` commits `BASE` at bad prices.

verifyfirst (Spartan) acknowledged and disagreed with severity:

> Implementing a TWAP needs more discussion and ideas to help with price manipulation. Attacking BOND is limited by its allocation, time and the fact that it's locked over 6months.

ghoul-sol (judge) commented:

> Per sponsor comment making this medium risk

[M-15] `DEPLOYER` can drain DAOVault funds + manipulate proposal results

Submitted by hickuphh3

2 conditions enable the `DEPLOYER` to drain the funds in the `DAOVault` .

- `DAOVault` is missing `purgeDeployer()` function

- `onlyDAO()` is callable by both the `DAO` and the `DEPLOYER`

The `DEPLOYER` can, at any time, call `depositLP()` to increase the LP funds of any account, then call `withdraw()` to withdraw the entire balance.

The only good use case for the `DEPLOYER` here is to help perform emergency withdrawals for users. However, this could use a separate modifier, like `onlyDeployer()`.

1. `DEPLOYER` calls `depositLP()` with any arbitrary amount (maybe DAOVault's pool LP balance - Alice's deposited LP balance) for Alice and pool to increase their weight and balance.
2. At this point, Alice may vote for a proposal to swing it in her favour, or remove it otherwise (to implicitly vote against it)
3. `DEPLOYER` calls `withdraw()` for the Alice, which removes 100% of her balance (and therefore, the entire DAOVault's pool balance)
4. Create a separate role and modifier for the `DEPLOYER`, so that he is only able to call `withdraw()` but not `depositLP()`
5. Include the missing `purgeDeployer()` function.

verifyfirst (Spartan) acknowledged:

> We are already aware of the privilege level the deployer holds with returning user funds to user's wallet. We disagree with the severity of this issue - however, agree it can be used to manipulate user's weight from a proposal. We will include the purge deployer into the vault to resolve this issue.

ghoul-sol (judge) commented:

> Privileged deployer is common in early stage protocols but because of lack of `purgeDeployer` function, I'll keep this medium risk.

Low Risk (35)

[L-01] Event log poisoning by griefing attackers

Event log poisoning is possible by griefing attackers who have no DAO weight but vote and emit event that takes up event log space. See L382 and L393 in `Dao.sol`.

Recommend emitting event only if non-zero weight as relevant to proposal voting/cancelling.

SamusElderg (Spartan) confirmed:

> Good point; @verifyfirst (Spartan) should we make the event conditional or is it used anywhere when the vote is zeroed out? From memory when zeroed-out it is simply done via mappings and doesn't emit an event anyway so probably safe to conditional this one (or remove it if we aren't using it in any user-facing interface)

verifyfirst (Spartan) commented:

> Yep, a conditional is a good one

[L-02] Attackers can grief voting by removing votes just before finalization

Attackers, i.e. malicious DAO members, can grief voting by removing their votes just before finalization and if that takes it below quorum, it forces others to vote and restarts another cooloff period of 3 days. This will delay the finalisation of targeted proposals and the griefing attackers lose nothing in penalty. See issue page for referenced code.

Recommend:

1. Redesign to allow vote removal only within a certain window after voting and locking it thereafter.
2. Removal of votes should have an associated penalty

SamusElderg (Spartan) acknowledged:

> I don't agree with locking up voters; as we all know people like to press buttons and sometimes form an opinion later; need to allow them to change their mind (or have their mind changed by good discussion)

> However, a fee/penalty of sorts for removeVote is an interesting discussion point 👍 @verifyfirst (Spartan)

verifyfirst (Spartan) commented:

> I agree with the suggestion of a removeVote penalty to reduce the unlikely grieving

[L-03] Pool.sol: `swapTo()` should not be payable

The `swapTo()` function should not be payable since the WBNB-SPARTA pool should not receive BNB, but WBNB. The router swap functions handles the wrapping and unwrapping of BNB.

Furthermore, the `swapTo()` will not detect any deposited BNB, so any `swapTo()` calls that have msg.value > 0 will have their BNB permanently locked in the pool contract.

Recommend removing `payable` keyword in `swapTo()`.

[L-04] Incorrect event parameter used in emit

Incorrect event parameter `outputAmount` is used (instead of output) in the MintSynth event emit. `outputAmount` is a named return variable that is never set in this function and so will always be 0. This should instead be output. This will confuse the UI or offchain monitoring tools that 0 synths were minted and will lead to users panicking/complaining or trying to mint synth again.

Recommend replacing `outputAmount` with output in the emit.

verifyfirst (Spartan) confirmed:

> Code needs refactoring

[L-05] Missing check for `toPool != fromPool`

`zapLiquidity()` used to trade LP tokens of one pool to another is missing a check for toPool != fromPool which may happen accidentally. The check will prevent unnecessary transfers and avoid any fees/slippage or accounting errors.

Recommend adding `toPool != fromPool` as part of input validation.

verifyfirst (Spartan) confirmed:

> Although very low risk, recommended mitigation is valid

[L-06] Unnecessary redundant check for `basisPoints`

The threshold check for `basisPoints` while a required part of input validation is an unnecessary redundant check because `calcPart()` does a similar upper bound check and the lower bound check on 0 is only an optimization.

Recommend removing redundant check to save gas and improve readability/maintainability.

[L-07] Missing `isListedPool` checks may lead to lock/loss of user funds

This `isListedPool` check implemented by `isPool()` is missing in many functions of the contract that accept pool/token addresses from users. `getPool()` returns the default mapping value of 0 for token that do not have valid pools. This lack of input validation may lead to use of zero/invalid pool addresses in the protocol context and reverts in the best case or burn/loss of user funds in the worst case.

Recommend combine `isPool()` `isListedPool` check to `getPool()` so that it always returns a valid/listed pool in the protocol.

verifyfirst (Spartan) acknowledged and confirmed:

> Code can be cleaner

[L-08] Number of curated pools can only be 10 at any point

Without a setter for curatedPoolSize, the number of curated pools at any point can only be a max of 10 forever, and will require removing one to accommodate another one. It is unclear if this is intentional and a requirement of the protocol.

Recommend a setter for `curatedPoolSize` that allows DAO to increase it if/when required. If not, document the hardcoded limit of curated pools number.

[L-09] Incorrect event parameter logs zero address instead of WBNB

The token argument used in `CreatePool` event emit of `createPoolADD()` should really be _token so that WBNB address is logged in the event instead of zero address when token == 0. Logging a zero address could confuse off-chain user interfaces because it is treated as a burn address by convention.

Recommend using _token instead of token in event emit.

SamusElderg (Spartan) confirmed:

> Non-critical, but makes sense; will change this 👍

[L-10] Missing check for already curated pool being re-curated

`addCuratedPool()` is missing a `require(isCuratedPool[_pool] == false)` check, similar to the one in `removeCuratedPool` to ensure that the DAO is not trying to curate an already curated pool which indicates a mismatch of assumption/accounting compared to the contract state.

Recommend adding `require(isCuratedPool[_pool] == false)` before setting `isCuratedPool[_pool]` = true.

SamusElderg (Spartan) confirmed:

> Check needs to be added for this issue

[L-11] Inconsistent value of `burnSynth` between Pool and Synth

When users try to born synth, the fee and the value of Sparta is calculated at contract `Pool` while the logic of burning `Pool` s Lp and Synth is located at `Synth`

contract.

Users can send synth to the `Synth` contract directly and trigger `burnSynth` at the `Pool` contract. The Pool would not send any token out while the `Synth` contract would burn the lp and Synth. While users can not drain the liquidity by doing this, breaking the AMM rate unexpectedly is may lead to troubles. The calculation of debt and the fee would end up with a wrong answer.

Pool's `burnSynth` and Synth's `burnSynth` are tightly coupled functions. In fact, according to the current logic, `Synth:burnSynth` should only be triggered from a valid `Pool` contract.

IMHO, applying the `Money in - Money Out` model in the `Synth` contract does more harm than good to the readability and security of the protocol. Consider to let `Pool` contract pass the parameters to the `Synth` contract and add a require check in the `Synth` contract.

[L-12] `synthVault.sol` **:** `_processWithdraw` **: Replace** `synthReward` **with principle**

[L-13] Missing zero-address checks in constructors and setters

Checking addresses against zero-address during initialization or during setting is a security best-practice. However, such checks are missing in address variable initializations/changes in many places. Given that zero-address is used as an indicator for BNB, there is a greater risk of using it accidentally.

Allowing zero-addresses will lead to contract reverts and force redeployments if there are no setters for such address variables.

Recommend adding zero-address checks for all initializations/setters of all address state variables.

[L-14] Mismatch in event definition

In synthFactory.sol, there's an `event CreateSynth(address indexed token, address indexed pool)`. However the event is emitted with "synth" as second output.

Recommend thinking about what's the better variable to be emitted, and correct one of the lines.

verifyfirst (Spartan) confirmed:

> Code can be a little bit more cleaner

[L-15] Missing revert if denominator = 0

In Synth.sol, the function `burnSynth()` calculates a division between two variables. Since they can be zero, it's better to have a require with a clear error message when the division is not possible, otherwise an user wouldn't know why a transaction reverted.

Recommend adding a `require(denom != 0, "LPDebt = 0")`.

verifyfirst (Spartan) confirmed:

> low risk but will help with error identification

[L-16] Missing input validation `zapLiquidity()`

`zapLiquidity()` in Router.sol misses an input validation unitsInput > 0.

Recommend adding an input validation for `unitsInput`.

verifyfirst (Spartan) acknowledged:

> Low risk but valid

[L-17] Loss of precision

In Router.sol, there's a loss of precision that can be corrected by shifting the operations.

Consider rewriting L274-275 with `uint numerator = (_fees * reserve) / eraLength / maxTrades;` .

verifyfirst (Spartan) confirmed:

> Suggested mitigation is valid

[L-18] Missing input validation in `addLiquidityForMember()`

In Router.sol, the function `addLiquidityForMember()` doesn't check inputBase and `inputToken` . Since we know they can't both be zero (it wouldn't change anything and user pays the gas for nothing).

Recommend considering adding a require `inputBase>0 || inputToken>0` .

[L-19] Dao.sol: Unbounded Iterations in `claimAllForMember()`

The `claimAllForMember()` function iterates through the full list of `listedAssets` . Should `listedAssets` become too large, as more assets are listed, calling this function will run out of gas and fail.

A good compromise would be to take in an array of asset indexes, so that users can claim for multiple assets in multiple parts.

```
    function claimAllForMember(address member, uint256[] calldata as
        address [] memory listedAssets = listedBondAssets; // Get ar
        for(uint i = 0; i < assetIndexes.length; i++){
            uint claimA = calcClaimBondedLP(member, listedAssets[ass
            if(claimA > 0){
                _BONDVAULT.claimForMember(listedAssets[assetIndexes[
            }
        }
        return true;
    }
```

SamusElderg (Spartan) confirmed :

> Whilst the history array of bondable assets is unlikely to ever exceed maybe 2 - 4 assets; I still like this suggested compromise from a user's gas-optimization perspective. The UI will always know exactly what assets the user should be able to claim before they press the button to 'claim all'; so i would like to see us pad this idea out

[L-20] Missing parameter validation

Some parameters of functions are not checked for invalid values:

- `PoolFactory.constructor` : Validate `_base` and `_wbnb` to be contracts or at least non-zero

A wrong user input or wallets defaulting to the zero addresses for a missing input can lead to the contract needing to redeploy or wasted gas.

Recommend validating the parameters.

ghoul-sol (judge) commented:

> Keeping this a low risk. Not sure why sponsor disputed.

[L-21] Can accidentally lose tokens when removing liquidity from pool 2

The `Pool.removeLiquiditySingle` function redeems liquidity tokens for underlying to the router contract in case of the `token` being the zero address. This works if the underlying token is actually `WBNB` but if the pool token is different and the user accidentally inserted `0` as the `token` address, it tries to swap a zero-balance WBNB to `BASE` and the redeemed tokens are stuck.

If `token == 0` add a check for `pool.token == WBNB` such that it is ensured that the pool's token is actually `WBNB` .

verifyfirst (Spartan) disputed:

> In theory this is correct, however, solidity validates function parameters to be legitimate and in this instance, 0 or "0" is not a valid address.

ghoul-sol (judge) commented:

> I'll keep the issue as it's technically correct.

[L-22] `MemberWithdraws` event not fired

The `MemberWithdraws` event of the DAO contract is not used. Unused code can hint at programming or architectural errors. Recommend use it or remove it.

[L-23] `calcAsymmetricValueToken` never used

The `Utils.calcAsymmetricValueToken` function is not used. Unused code can hint at programming or architectural errors.

Recommend using it or removing it.

[L-24] `memberCount` not accurate

The function `depositForMember` of BondVault.sol adds user to the array `arrayMembers`. However it does this for each asset that a user deposits. Suppose a user deposit multiple assets, than the user is added multiple times to the array `arrayMembers`.

This will mean the `memberCount()` doesn't show accurate results. Also `allMembers()` will contain duplicate members

```
60
61 function depositForMember(address asset, address member, uint
62     if(!mapBondAsset_memberDetails[asset].isMember[member]){
63         mapBondAsset_memberDetails[asset].isMember[member] =
64         arrayMembers.push(member); // Add user to member arra
65         mapBondAsset_memberDetails[asset].members.push(member
66     }
67     ...
68
69 // Get the total count of all existing & past BondVault membe
70 function memberCount() external view returns (uint256 count){
71     return arrayMembers.length;
72 }
```

```
73 function allMembers() external view returns (address[] memory
74     return arrayMembers;
75 }
```

Use a construction like this:

```
mapping(address => bool) isMember;
if(!isMember[member]){
        isMember[member] = true;
        arrayMembers.push(member);
}
```

SamusElderg (Spartan) confirmed and disagreed with severity:

> This appears to be true 👍 Will need to have some discussion around whether it is worth the extra gas for the extra check when adding the member. My limited opinion is that it is worth the extra gas to add the extra conditional for this one and have counts lining up to the correct amount even if it isn't used elsewhere. But @verifyfirst (Spartan) ill let you decide on that!

ghoul-sol (judge) commented:

> Sponsor confirmed so I'm keeping this

[L-25] check if pool exists in `getPool`

The function `getPool` doesn't check if the pool exits (e.g. it doesn't check if the resulting pool !=0) Other functions use the results of `getPool` and do followup actions.

For example `createSynth` checks `isCuratedPool(_pool)` == true; if somehow `isCuratedPool(0)` would set to be true, then further actions could be done. As far as I can see no actual problem occurs, but this is a dangerous construction and future code changes could introduce vulnerabilities. Additionally the reverts that will occur if the result of `getPool` ==0 are perhaps difficult to troubleshoot.

```
120 function getPool(address token) public view returns(address
121     if(token == address(0)){
122         pool = mapToken_Pool[WBNB];    // Handle BNB
123     } else {
124         pool = mapToken_Pool[token];  // Handle normal token
125     }
126     return pool;
127 }
128
129 function createPoolADD(uint256 inputBase, uint256 inputToken
130     require(getPool(token) == address(0)); // Must be a vali
131
132 function createPool(address token) external onlyDAO returns(
133     require(getPool(token) == address(0)); // Must be a vali
```

```
37
38 function createSynth(address token) external returns(address
39     require(getSynth(token) == address(0), "exists"); // Synt
40     address _pool = iPOOLFACTORY(_DAO().POOLFACTORY()).getPoo
41     require(iPOOLFACTORY(_DAO().POOLFACTORY()).isCuratedPool(
```

Recommend In function `getPool` add something like:

```
require  (pool !=0, "Pool doesn't exist");
```

Note: the functions `createPoolADD` and `createPool` also have to be changed, to use a different way to verify the pool doesn't exist.

[L-26] `Approval` event not emitted if the allowance is the maximum

According to the BEP20 specification, the `Approval` event:

MUST trigger on any successful call to `approve(address _spender, uint256 _value)`.

However, the implementation of pool LP tokens and synths do not emit the `Approval` event when the allowance is the maximum number, i.e.,

type(uint256).max.

Recommend emitting the approval event whenever the approve call succeeds, even if the allowance does not change.

[L-27] Utils.sol: Combine Swap Output + Fee Calculation to avoid Rounding Errors + Integer Overflow [Updated]

For minting, burning of synths and swaps, the fee and output amounts are calculated separately via `calcSwapOutput` and `calcSwapFee`. To avoid rounding errors and duplicate calculations, it would be best to combine both of these functions and return both outputs at once.

For example, if we take `x = 60000, X = 73500, Y = 50321`, the actual swap fee should be `10164.57` and output `12451.6`. However, `calcSwapOutput` and `calcSwapFee` returns `10164` and `12451`, leaving 1 wei unaccounted for. This can be avoided by combining the calculations as suggested below. The fee and actual output will be `10164` and `12452` instead.

Functions that have to call `calcSwapOutput` within the contract (eg. `calcSwapValueInBaseWithPool`) should call this function as well, for calculation consistency.

In addition, calculations for both `calcSwapOutput` and `calcSwapFee` will phantom overflow if the input values become too large. (Eg. `x = 2^128, Y=2^128`). This can be avoided by the suggested implementation below using the FullMath library.

```
function calcSwapFeeAndOutput(uint x, uint X, uint Y) public pur
    uint xAddX = x + X;
    uint rawOutput = FullMath.mulDiv(x, Y, xAddX);
    swapFee = FullMath.mulDiv(rawOutput, x, xAddX);
    output = rawOutput - swapFee;
}

function calcSwapValueInBaseWithPool(address pool, uint amount)
    uint _baseAmount = iPOOL(pool).baseAmount();
    uint _tokenAmount = iPOOL(pool).tokenAmount();
    (_output, ) = calcSwapFeeAndOutput(amount, _tokenAmount, _ba
```

```
    }
```

The `FullMath` library is included (and made compatible with sol 0.8+) on the issue page for convenience.

[L-28] Dao.sol: Reserve emissions must be turned on for `depositLPs` and bonds

`depositLPForMember()` and `bond()` invokes `harvest()` if a user has existing LP deposits or bonded assets into the DAO. This is to prevent users from depositing more assets before calling `harvest()` to earn more DAOVault incentives. However, `harvest()` reverts if reserve emissions are turned off.

Hence, deposits / bonds performed by existing users will fail should reserve emissions be disabled.

Cache claimable rewards into a separate mapping when `depositLPForMember()` and `bond()` are called. `harvest()` will then attempt to claim these cached + pending rewards. Perhaps Synthetix's Staking Rewards contract or Sushiswap's FairLaunch contract can provide some inspiration.

[L-29] Missing zero address check on `BondVault` constructor

This is a low risk vulnerability due to the fact that it is possible to lose funds if the Base address is set to a zero address and someone sends funds to this address. As a rule, there should always be checks to make sure that initialized addresses are never a zero address.

According to Slither analysis documentation (https://github.com/crytic/slither/wiki/Detector-Documentation#exploit-scenario-49), there needs to be a zero address checkpoint when initializing a base address in a contract. In the case for `BondVault`, the constructor initializes a base address. There should be a check to make sure this address is never zero to make sure there is no way to lose funds.

Slither detector:

missing-zero-check:

`BondVault.constructor(address)._base` (contracts/BondVault.sol#37) lacks a zero-check on : `BASE = _base (contracts/BondVault.sol#38)`

See issue page for Slither output from console (JSON format):

Recommend:

1. Clone repository for Spartan Smart Contracts
2. Create a python virtual environment with a stable python version
3. Install Slither Analyzer on the python VEM
4. Run Slither against all contracts

verifyfirst (Spartan) acknowledged

[L-30] Can't add BNB with `createPoolADD`

The function `createPoolADD()` supports the input of BNB, which it detects by checking `token == address(0)` Later it calls `_handleTransferIn(token, ...);` with the original value of token, which can be 0.

However in the function `_handleTransferIn()` in poolFactory.sol there is no provision to transfer BNB (it doesn't check for _token == 0), so it will revert when you try to add BNB.

As a comparison, the function `_handleTransferIn()` of Router.sol does check for _token == address(0) and takes appropriate action.

```
45
46 function createPoolADD(uint256 inputBase, uint256 inputToken,
47 ...
48     address _token = token;
49     if(token == address(0)){_token = WBNB;} // Handle BNB -> 
50         ...
51     _handleTransferIn(token, inputToken, pool); // Transfer T
52
53 function _handleTransferIn(address _token, uint256 _amount, a
```

```
54      if(_amount > 0) {
55          uint startBal = iBEP20(_token).balanceOf(_pool);
56          iBEP20(_token).transferFrom(msg.sender, _pool, _amoun
57          actual = iBEP20(_token).balanceOf(_pool) - (startBal)
58      }
59  }
```

```
197
198 function _handleTransferIn(address _token, uint256 _amount,
199     if(_amount > 0) {
200         if(_token == address(0)){
201             require((_amount == msg.value));
202             (bool success, ) = payable(WBNB).call{value: _am
203             require(success, "!send");
204             iBEP20(WBNB).transfer(_pool, _amount); // Transf
205             actual = _amount;
206         } else {
207             uint startBal = iBEP20(_token).balanceOf(_pool);
208             iBEP20(_token).transferFrom(msg.sender, _pool, _
209             actual = iBEP20(_token).balanceOf(_pool)-(startB
210         }
211     }
212 }
```

Recommend applying the same function as `_handleTransferIn` of Router.sol to `_handleTransferIn` of poolFactory.sol. Better yet deduplicate the function by moving it to a library/included solidity file. Note: There is also a `_handleTransferIn` in Synth.sol which isn't used.

SamusElderg (Spartan) disputed:

> Whilst true; the intention is always that BNB will already be listed as standard; so the user's createPoolADD() function is irrelevant to BNB. However, this was not made clear anywhere; so is a good observation! @verifyfirst (Spartan) should we leave this? Or Block BNB pool thru that function? Or adjust the function to account for BNB even though it will already be listed?

SamusElderg (Spartan) commented:

> No need for action on this one; BNB pool will be deployed at the same time.

ghoul-sol (judge) commented:

> Per sponsor comment, low risk

[L-31] Possible divide by zero errors in `Utils`

Several functions in `Utils` do not handle edge cases where the divisor is 0, caused mainly by no liquidity in the pool. In such cases, the transactions revert without returning a proper error message.

See issue page for referenced code: Recommend checking if the divisors are 0 in the above functions to handle edge cases.

verifyfirst (Spartan) confirmed

[L-32] Purging DAO deployer immediately in a single-step is risky

The DAO deployer is used as the authorized address in the modifier onlyDAO allowing it to set various critical protocol addresses and parameters. The `purgeDeployer()` function is expected to be called by the deployer once the DAO is stable and final. However, a single-step critical action such as this is extremely risky because it may be called accidentally and is irrecoverable from such mistakes.

Scenario 1: The DAO is not yet stable and final. But the deployer, e.g. controlled by an EOA, accidentally triggers this function. The protocol parameters/addresses can no longer be changed when required. The entire protocol has to be halted and redeployed. User funds have to be returned. Protocol reputation takes a hit.

Scenario 2: The DAO is not yet stable and final but the deployer incorrectly assumes it is final and triggers this function. The protocol parameters/addresses can no longer be changed when required. The entire protocol has to be halted and redeployed. User funds have to be returned. Protocol reputation takes a hit.

While a two-step process is generally recommended for critical address changes, a single-step purge/renounce is equally risky if it is controlled by an EOA and is not timelocked.

At a minimum, make sure that (1) deployer is not an EOA but a multisig controlled by mutually independent and trustworthy entities, (2) this function is timelocked.

A better design change would be to let the DAO decide if it considers itself stable/final and let it vote for a proposal that purges the deployer.

verifyfirst (Spartan) acknowledged:

> Purge deployer is something that would be completed once the protocol is completely stable and future proof. The risk of accidentally calling purgeDeployer is very low. Suggested mitigations are considered.

ghoul-sol (judge) commented:

> Possibility of this happening is very low so making this low risk

[L-33] Calling `synthVault :_deposit` multiple times, will make you loose rewards

Calling `deposit` multiple times will change the `mapMemberSynth_lastTime` to

```
mapMemberSynth_lastTime[_member][_synth] = block.timestamp + mir
```

This is used in `calcCurrentReward` to calculate how much the user earned.

Everytime the user calls `_deposit` (via deposit), the `mapMemberSynth_lastTime` will be set to a date in the future, meaning that they will loose all the rewards they accrued. Calling `deposit` calls `_deposit` without harvesting for the user meaning that they lost those rewards.

Recommend force harvest user rewards at the beginning of every `_deposit()`

verifyfirst (Spartan) acknowledged:

> This is only an issue if a user directly interacts with the contracts - assuming they know what they are doing. DAPP has already implemented deterrents to avoid this inconvenience for users.

ghoul-sol (judge) commented:

> This requires user to call functions to deliberately harm themselves. Making this low risk.

[L-34] Attacker can trigger pool sync leading to user fund loss

An attacker can front-run any operation that depends on the pool contract's internal balance amounts being unsynced to pool's balance on token/base contracts effectively nullifying the transfer of base/tokens for those operations. This will make `_getAddedBaseAmount()` and `_getAddedTokenAmount()` return 0 (because the balances are synced) from such operations.

The affected operations are: `addForMember()`, `swapTo()` and `mintSynth()` which will all take the user funds to respective contracts but will treat it as 0 (because of the syncing) and thus not add liquidity, return swapped tokens or mint any synths to the affected users. User loses deposited funds to the contract.

Recommend adding access control to `sync()` function so that only Router can call it via `addDividend()`.

verifyfirst (Spartan) confirmed and disagreed with severity:

> Whilst we disagree with the above attack vector, it brings up a point about permissions on the pool's sync() function which was always intended to be called by anyone incase of accidentally send in. However, we have decided to permission the sync to router only just for peace of mind.

SamusElderg (Spartan) commented:

> To be clear; this is non-critical based on the warden's outlined scenario. Front-running a user's txn would mean sync() is called before the user's funds are sent in, so sync() would have no effect on a txn that hasn't happened yet. Unpermissioned sync() Might however have low risk or otherwise in other scenarios but cant simulate or think of any. Regardless we will permission sync() to close any vector that has not been thought of there.

ghoul-sol (judge) commented:

> Per sponsor comment, I align with low risk

[L-35] Vote weight can be manipulated

The vote weight is determined by the `DAOVault` and `BondVault` weight (`voteWeight = _DAOVAULT.getMemberWeight(msg.sender) + _BONDVAULT.getMemberWeight(msg.sender)`). The weight in these vaults is the deposited LP token. The `BondVault` however pays for the `BASE` part itself (see `DAO.handleTransferIn`), therefore one only needs to deposit `tokens` and the `DAO` matches the **swap value**.

Therefore, it's possible to manipulate the pool, deposit only a small amount of `tokens` (receiving a large amount of matching `BASE` by the DAO) and receive a large amount of LP tokens this way. attack can be profitable:

1. Manipulate the pool spot price by dripping a lot of `BASE` into it repeatedly (sending lots of smaller trades is less costly due to the path-independence of the continuous liquidity model). This increases the `BASE` per `token` price.
2. Repeatedly call `DAO.bond(amount)` to drip `tokens` into the `DAO` and get matched with `BASE` tokens to provide liquidity. (Again, sending lots of smaller trades is less costly.) As the LP minting is relative to the manipulated low `token` reserve, a lot of LP units are minted for a low amount of `tokens`, leading to receiving large weight.
3. Create a proposal to send the entire reserve balance to yourself by using `grantFunds`
4. Unmanipulate the pool by sending back the `tokens` from 1. This might incur a loss.

The cost of the attack is the swap fees from the manipulation of 1. and 4. plus the (small due to manipulation) amount of tokens required to send in 2. The profit can be the entire reserve amount which is unrelated to the pools (plus reclaiming lots of LP units over the span of the `BondVault` era). The attack can be profitable under certain circumstances of:

- high reserves
- low liquidity in the pool

I don't think the attack would be feasible if we couldn't get the `DAO` to commit the lion's share of the `BASE` required to acquire LP units through the `BondVault` incentives.

verifyfirst (Spartan) disputed and disagreed with severity:

> Warden must understand the bond program is extremely limited in time and amount of sparta allocated through the DAO. If the attacker was able to obtain the entire bond allocation and weight is in sparta terms, the opportunity to attack would scale along with the pool depth and therefor total weight scales up along with the bond. Grant funds will be capped at a % of the reserve.

ghoul-sol (judge) commented:

> Per sponsor comment, making this low risk

Non-Critical findings

- [N-01] Lack of emission of event when changing dao fees
- [N-02] (out of scope) mintFromDAO of Sparta.sol can go over max supply
- [N-03] Router.sol: Better changeArrayFeeSize implementation
- [N-04] Synth.sol: Redundant _handleTransferIn, onlyDAO, DEPLOYER
- [N-05] Router.sol: lastMonth variable is private
- [N-06] Dao.sol: Return votes > consensus
- [N-07] DaoVault.sol & BondVault.sol: Discrepancies in mapping visibility
- [N-08] isEqual(): Inconsistent Implementation
- [N-09] Pool.sol + Synth.sol: Inconsistent Allowance Checking Implementation
- [N-10] Dao.sol: newParamProposal takes in uint32 param
- [N-11] Utils.sol: Calculation issue with Slippage Adjustment
- [N-12] Dao.sol: Unused hasMinority()
- [N-13] Contract file name does not follow coding conventions

- [N-40] Router.revenueDetails(uint256,address) potentially vulnerable to miner manipulation
- [N-41] DaoVault.constructor(address) is missing a zero address check
- [N-42] Pool._addPoolMetrics(uint256) is subject to potential miner manipulation
- [N-43] Strict equality used in claimForMemeber()
- [N-44] `approveAndCall` approve `max` amount of token

Gas Optimization (25)

- [G-01] BondVault.sol: Optimizations
- [G-02] Type mismatch between parameters of setGenesisFactors() and state variables
- [G-03] `POOLFACTORY.curatedPoolCount()` Gas Optimization
- [G-04] state variables that can be declared as immutable
- [G-05] state variable that can be declared as constant
- [G-06] `claimAllForMember` could be optimized
- [G-07] more efficient calls to DAO functions
- [G-08] Router.sol: Optimize calculation of totalTradeFees in addTradeFee()
- [G-09] Dao.sol: Restrict Function Visibilities
- [G-10] Pool.sol: Optimizations
- [G-11] Pool.sol + Router.sol: Set revenue directly as _fee
- [G-12] Router.sol: Redundant _token initialization in addLiquiditySingleForMember()
- [G-13] Dao.sol: Define BASE as iBEP20 instead of address
- [G-14] Utils.sol: Redundant two assignment in calcLiquidityUnitsAsym()
- [G-15] Variables that can be converted into immutable
- [G-16] Use unchecked blocks in some cases to save gas.
- [G-17] Function purgeDeployer() should be declared external in BondVault.sol
- [G-18] Variable one in Utils.sol can be set to constant
- [G-19] Remove _token from addLiquiditySingleForMember

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