Fei Protocol Audit

OPENZEPPELIN SECURITY | FEBRUARY 5, 2021

Security Audits

The Fei Protocol team asked us to review and audit their Protocol smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `29aeefddd97f31c7f2a598fb3dca3ef24dc0beb4` of the fei-protocol/fei-protocol-core repository. In scope were the contracts directly inside the `/contracts` folder, however the contracts within `/contract/mocks` were out of scope. *Note: the repository in question is private at the time of writing, so many hyperlinks will only work for the Fei team.*

System overview

Fei Protocol is the first protocol creating a stablecoin using a new stability mechanism called *direct incentives*. This stablecoin, named `FEI`, is not fully collateralized, but instead uses a system of incentives to encourage trading activity that brings the token back to its USD peg. This is achieved by attaching incentives to trades through an ETH/FEI Uniswap Pair, that mint or burn `FEI` tokens in certain situations.

The protocol uses a bonding curve to price `FEI`, implements a *Protocol Controlled Value* to enable the incentive design, and has a governance token called `TRIBE` to allow user-controlled upgrades to the system

Below we summarize some of the key aspects of the Fei Protocol.

and the user is minted tokens to represent the amount that the protocol owes them. Protocol Controlled Value (PCV) instead is money that the protocol owns outright – that is not owed to a user. This value is what allows the protocol to provide incentives and influence market conditions. The protocol's bonding curve is what allows the protocol to calculate and accrue PCV.

Genesis Period

The Genesis period is 3 days in which a contract known as the `GenesisGroup` collects the ETH needed to bootstrap the protocol. During this time, any user can deposit ETH into the contract, and receive `FGEN` tokens in return. These tokens entitle the user to an amount of `FEI` and `TRIBE` tokens as a reward after launch. Users contributing to the Genesis are guaranteed that they will receive `FEI` at a maximum price of $0.90, which should then be pegged to $1 after launch and after a thawing period.

The Genesis period ends after 3 days, or earlier if the protocol collects enough ETH. The contract will then add liquidity to the `FEI/TRIBE` and `ETH/FEI` Uniswap pools, initialize the bonding curve oracle, and deposit rewards back into the `GenesisGroup` contracts for contributors to collect.

Fei Reward pool

After the Genesis period ends, the Fei Reward Pool is initialized. Users can deposit `FEI` and `TRIBE` in the `FEI/TRIBE` Uniswap pool to get `FEI/TRIBE` liquidity pool tokens and stake them into the Fei Reward Pool to earn `TRIBE`. Initially, the pool will hold 20% of `TRIBE`'s total supply and will release them over the course of two years.

Incentive contracts

Incentive contracts control the direction and magnitude of incentives for each `FEI` transfer. The incentive contract can either be a minter, a burner or both. When a user sends `FEI` to an incentivized address, the final balance after the transfer will either be affected by a mint or burn incentive. The primary incentivized address in Fei Protocol is the `FEI/ETH` Uniswap pool. When the `FEI/ETH` price is below the peg, the incentive contract will mint `FEI` to traders on a buy operation, taking into account a time-weighted magnitude of the distance from the peg. When a user sells `FEI` at or below the peg, the trade is disincentivized by a dynamic burn mechanism.

Privileged roles

At deployment, the Fei team will set the `admin` address for the protocol. This protocol admin will be granted high-level permissions, becoming a `Governor`, `Revoker`, and `Guardian` in the system. The admin will also be the `beneficiary` in the `IDO` contract.

The admin will be able to:

- Add and revoke roles,
- Perform emergency reweights to restore the `FEI` peg,
- Modify the `scale`, `buffer`, and `allocations` of the bonding curve,
- Change the address of the `FEI` token,
- Transfer/allocate unallocated `TRIBE` tokens,
- Flip the kill switch for the bonding curve oracle,
- Change the oracle durations used to calculate time-weighted average prices,
- Change the protocol controlled value (PCV) deposit address,
- Change the reweight incentive amount,
- Change the minimum distance from the peg required for reweight incentivization,
- Change incentive contract addresses,
- Add and remove addresses that are exempt from incentivizations,
- Change the growth rate and time weight for the incentivization calculations,
- Withdraw `FEI` from the Uniswap pool back to the protocol treasury,
- Setup new token pairs for use by the protocol,
- Change oracle addresses,
- Change the genesis group address,
- Assign a new beneficiary for the IDO contract

There are functions in place to transfer these roles to other addresses and to allow for the abdication of the `Guardian` role in the future.

In addition to these roles, there are also `Minter`, `Burner`, and `PCV Controller` roles. The `Governor` can assign these roles to any address, and has all of their abilities.

They can:

Security model and trust assumptions

Voters use `TRIBE` to vote. It is paid out to liquidity providers, traders, and referrers. `TRIBE` can be delegated and is transferable. The voting system has the ability to arbitrarily update many critical variables and contracts. This includes the ability to change pricing oracles, which means the voting system has all the powers of an oracle, along with many more. In this audit, we assume that the voters will not pass proposals that would harm `FEI`.

The Fei protocol is pegged to the exchange rate of `ETH/USDC` as a proxy for `ETH/USD`. Therefore, `FEI` is exposed to the risks inherent in `USDC`. In this audit, we do not assess the risks in the `USDC` protocol and their effects on the Fei protocol.

The Fei protocol is initialized to use the Uniswap pricing oracle, which is integral to the system. During this audit, we assumed that the administrator and price feeds are available, honest, and not compromised.

Findings

Here we present three client-reported issues, followed by our findings.

Bypass burn penalty (client reported)

The client reported an economic vulnerability that could be used to bypass the burn penalty applied upon selling `FEI`. Contracts which hold pooled `FEI` owned by users and with the ability to withdraw to an arbitrary address, the following attack is possible:

1. Withdraw `FEI` from the pooled contract to the `FEI/ETH` Uniswap pool.
2. The burn penalty is then borne by the contract which holds `FEI` pooled from other users.
3. Execute the swap on Uniswap to receive ETH.

This attack can be executed by a secondary Uniswap market, or other contracts that pool ERC20 tokens. This has been addressed in PR#10.

Rounding errors during reweight (client reported)

addressed in PR#22.

Attacker can manipulate ETH/FEI spot price on allocations (client reported)

The client reported a vulnerability in which an attacker can manipulate the spot price of the `ETH/FEI` Uniswap pool by performing a flash loan to borrow ETH and buy `FEI`, moving the price up right before an allocation is executed, and sell that `FEI` amount after the allocation is done to make a profit. This has been addressed in PR#81.

Update

Most of the following issues have been either fixed, partially fixed, or acknowledged by the Fei Team. Our analysis of the mitigations is limited to the specific changes made to cover the issues, and disregards all other unrelated changes in the pull requests and in the codebase.

Critical severity

[C01] Anyone can steal all the `TRIBE` tokens from the reward pool

The `FeiPool` contract allows users to deposit `FEI/TRIBE` liquidity pool (LP) tokens and earn `TRIBE` tokens as a reward. The amount of `TRIBE` that a user can claim is given by the following formula:

$$redeemableReward = \frac{releasedReward * redeemablePoolTokens}{totalRedeemablePoolTokens}$$

Where `releasedReward` is the total amount of released reward tokens from the total amount available, `redeemablePoolTokens` is the amount of reward tokens that the user can claim, and `totalRedeemablePoolTokens` is the total amount of redeemable pool tokens by all the stakers participating in the pool. Both `redeemablePoolTokens` and `totalRedeemablePoolTokens` increase over time, which means that the more time the user leaves their stake in the pool, the more rewards they will be able to claim.

the [redacted] function. An attacker can do the following:

1. Deposit an amount of `FEI/TRIBE` LP tokens when the pool opens from two different accounts, `account A` and `account B`.
2. Let some time pass so that the numerator in the formula that calculates the `amountReward` value equals or is near the pool's `rewardBalance`, which includes both the released and unreleased rewards. This will not take much time since the numerator is proportional to the released amount of rewards given by the `releasedReward` function, the number of tokens staked, and the remaining time of the pool.
3. Burn an amount of `_totalRedeemablePoolTokens - 1` pool tokens from `account B`. This way, the denominator in the formula will equal `1`, and the `amountReward` value returned by the `redeemableReward` function will equal the pool's `rewardBalance` as calculated in 2.
4. Withdraw or claim rewards from `account A`.
5. Wait until the pool closes and withdraw staked tokens from `account B`.

Since the `FeiPool` contract holds 20% of `TRIBE`s total supply, the attacker would be stealing 200,000,000 `TRIBE` tokens in the worst case. A step-by-step proof-of-concept exploit for this scenario can be found in this secret gist.

Consider disallowing external burns by removing the `ERC20Burnable` inheritance from the `Pool` contract, and only allow burning to be an internal operation triggered by claiming rewards.

Update: *Fixed in PR#32. The `FeiPool` contract no longer inherits from the `ERC20Burnable` contract. Additionally, the `burnFrom` public function was removed, and a new `_burnFrom` function was added to be used in the `_withdraw` function.*

[C02] ETH can get stuck in the `GenesisGroup` contract

The `GenesisGroup` contract allows users to contribute ETH by using the `purchase` function, which mints them `FGEN` tokens to record the size of their deposit. When enough ETH has been contributed to reach the `maxGenesisPrice`, or the genesis period has ended, the `launch` function can be called to transfer all the ETH collected to `PCVDeposit` contracts, and launch the

However, if the `launch` function gets called due to the `maxGenesisPrice` being reached, the contract does not prevent `purchase` from being called by users. This would likely happen when users race to contribute to the genesis before launch. This has the following negative consequences:

- The ETH deposited by users will become locked in the `GenesisGroup` contract. This is due to the fact that `launch` is the only function that can transfer ETH out of the `GenesisGroup`, and it cannot be called a second time – enforced by `completeGenesisGroup` reverting if someone tries to call `launch` a second time.
- The additional calls to `purchase` continue to mint `FGEN` to the callers. This increases the `totalSupply` of `FGEN`, and skews the calculation in `_fgenRatio`, meaning that valid contributors to the `GenesisGroup` will not receive their rightful allocation of `TRIBE` and `FEI`.

Consider adding a condition to the `purchase` function that prevents it from being called if `hasGenesisGroupCompleted` is `true`.

Update: *Fixed in PR#24. The Fei Team decided to remove the `isAtMaxPrice` condition that allowed the protocol to be launched before the genesis period passed. Additionally, a condition was added to check whether the genesis was launched when calling the `purchase` function.*

[C03] Anyone can partially bypass the `FEI` sell penalty to earn a profit

The `UniswapIncentive` contract defines the incentives to be applied when users buy or sell `FEI` tokens in the `ETH/FEI` Uniswap pool. On the one hand, when a user sells tokens and causes a deviation in the price peg, a sell penalty is applied and the protocol burns a portion of their tokens. This penalty is calculated using the following formulas:

$$sellPenalty = sellIncentiveMultiplier * amountSold$$

$$sellIncentiveMultiplier = deviation^2 * 100$$

the user's account after the sell operation is performed.

On the other hand, when a user buys `FEI` tokens to move the price back towards the peg, a buy incentive is applied and the protocol mints tokens to them. This incentive is calculated as follows:

$$buyIncentive = min(sellIncentiveMultiplier, \frac{deviation * weight}{timeWeightGranularity}) * amountBought$$

Where `deviation` is the price deviation from the peg before buying the `amountBought` amount, `weight` is the time-weighted function, `timeGranularity` is the granularity, and `sellIncentiveMultiplier` is the sell penalty multiplier described above. Notably, when the `buyIncentiveMultiplier` is equal or greater than the `sellIncentiveMultiplier`, incentive parity is reached, and the `EthUniswapPCVController` contract may be able to do a reweight to move the price back to the peg if necessary.

Since the `sellIncentiveMultiplier` function is quadratic, performing several small sell operations results in a considerably lower total penalty than performing one large sell operation, each of which take the `FEI` price from `deviation A` to `deviation B`.
This means that an attacker would be able to buy a `X` amount of `FEI` to move the price from `deviation B` to `deviation A` and get the buy incentive described above (capped at the sell incentive multiplier), and then sell the same `X` amount of `FEI` in multiple sell operations to move the price back from `deviation A` to `deviation B`. This could significantly reduce the sell penalty, and the attacker will retain a profit. Notably, the maximum profit will be reached when the incentive parity is reached.

Moreover, since performing a buy operation in the first place will either decrease or reset the time-weight function, other users may not have enough incentive to move the price back to the peg, and since there will no longer be incentive parity, the `EthUniswapPCVController` will not be able to do a conventional reweight, instead having to use the force reweight function.

Consider revising the sell incentive multiplier formula so that multiple sell operations will not decrease the sell penalty, or consider revising the buy incentive multiplier so that even after performing multiple sell operations, the attack is not profitable.

[C04] Users can claim unreleased rewards or have their funds locked

The `FeiPool` contract allows users to deposit `FEI/TRIBE` liquidity pool (LP) tokens into the Fei Pool to accrue `TRIBE` tokens as a reward. After some time, users may want to withdraw their LP tokens and get the accrued rewards. To track users' staked amounts and the total staked amount in the pool, the contract defines the `stakedBalance` and `totalStaked` variables, respectively. These are incremented each time a user deposits liquidity pool tokens into the pool.

When a user withdraws their staked tokens and accrued rewards, only the user's `stakedBalance` is updated, while the `totalStaked` variable remains the same. Since the `redeemableRewards` function depends on the `totalStaked` variable to compute the amount of redeemable rewards in the `_totalRedeemablePoolTokens` function, the total amount of redeemable tokens for the user is miscalculated, resulting in more tokens being released than what was intended.

Additionally, since the Fei Pool's total supply of tokens decreases during claims and withdrawals, while the `totalStaked` variable is unchanged, the `balance` variable from the `_totalRedeemablePoolTokens` function could become greater than the `total` variable. This would prevent users from claiming rewards and withdrawing their stake due to a "Redeemable underflow" require statement.

Consider decrementing the `totalStaked` variable in the `_withdraw` function to accurately track the total stake in the pool and avoid these scenarios.

Update: *Fixed in PR#19. The `totalStaked` amount is now being decremented in the `_withdraw` function.*

[C05] Anyone with enough liquidity to reach the `maxGenesisPrice` can make profit from genesis

The `GenesisGroup` contract allows users to deposit ETH through the `purchase` function in exchange for `FGEN` tokens, which represent the proportion of the total ETH in the contract that the user deposited. After the genesis period has ended or after the `maxGenesisPrice` has been reached, the protocol can be launched via the `launch` function (which can be called by

Since all of the operations mentioned above can be executed in the same transaction, a flash loan could be used to steal ETH using the following attack:

1. Borrow the amount of ETH necessary to increase the average price to the `maxGenesisPrice`.
2. Launch the protocol by calling the `launch` function. This operation will:

– Initialize the bonding curve oracle, which will start with an initial price that equals the `maxGenesisPrice`.
– Purchase `FEI` in the bonding curve so that users can redeem them.
– Add liquidity to the `FEI/TRIBE` Uniswap pool at an exchange rate based on the amount of `FEI` purchased in the genesis period, the amount of `TRIBE` held by the `IDO` contract and the `exchangeRateDiscount` variable.
– Add liquidity to the `ETH/FEI` Uniswap pool.
3. Call the `redeem` function to withdraw `FEI` and `TRIBE` tokens based on the ratio of `FGEN` owned.
4. Sell all the `TRIBE` earned in exchange for `FEI` in the initialized `FEI/TRIBE` Uniswap pool.
5. Sell all the `FEI` (earned from the `redeem` function, and from selling `TRIBE`) in the initialized `ETH/FEI` pool in exchange for ETH.
6. Repay the loan and keep the remaining ETH as profit.

The amount used by the attacker should be such that they can afford the price impact on Uniswap's `ETH/FEI` pair, and still make profit. A spreadsheet that calculates this amount can be found here.

If this attack is performed, then:

- The `ETH/FEI` Uniswap pool price will be below the bonding curve oracle's peg, forcing a reweight, or other users in the system will try to restore the peg by interacting with the incentive contracts.
- The `ETH/FEI` Uniswap pool will be launched with less ETH than expected.
- The `FEI/TRIBE` pool will be launched with less `FEI` and a lower price than expected.
- The attacker will steal around 800,000 USD in ETH from the protocol.

Consider flagging the block where the `maxGenesisPrice` is reached in a global variable and restricting the `launch` function to only be called when the current block number is higher than this value. Additionally, consider revising the `exchangeRateDiscount` amount so that this scenario is not profitable for a user with enough liquidity nor for an attacker using a flash loan.

Update: *Fixed in PR#27. The flash loan attack was fixed by disallowing to launch the protocol and redeem `FEI` and `TRIBE` in the same transaction. Additionally, the initial bonidng curve oracle price has been decreased to drastically reduce the profit that users with enough liquidity would make by selling all their `FEI` and `TRIBE` tokens right after the protocol is launched.*

High severity

[H01] Users participating in Genesis can get fewer `FEI` and `TRIBE` than expected

The `GenesisGroup` contract stops the genesis period earlier than its `duration`, if enough ETH has been contributed to reach the *maximum price of `FEI` paid by the Genesis Group*. This attractive maximum price incentivizes users to contribute to the `GenesisGroup` to bootstrap the protocol.

However, the `GenesisGroup` continues to accept `purchase` calls after the `maxGenesisPrice` has been reached. Even with a solution to **ETH can get stuck in the `GenesisGroup` contract issue**, `purchase` transactions will continue to be accepted until the `launch` function is called. These additional purchases could push the average price for the `GenesisGroup` far above the `maxGenesisPrice`, causing contributors to receive fewer `FEI` and `TRIBE` tokens than expected. In extreme cases, users could end up receiving `FEI` that is valued at less than their ETH investment (measured in USD).

Consider implementing a check in the `purchase` function that prevents it from being executed if the purchase causes the `maxGenesisPrice` to be exceeded. Notably, if the maximum price has not been reached, but a purchase would cause the maximum price to be exceeded, some or all of the purchase amount should be rejected.

check whether the genesis was launched when calling the [illegible] function. Additionally, the initial `FEI` price has been capped in the `BondingCurve` contract so that it does not exceed the buffer adjusted amount, which means that if enough ETH is collected in genesis, users will not benefit from a discount in `FEI`.

[H02] Outdated oracle prices being used throughout the protocol

Throughout the system, the `UniswapOracle` contract and the `BondingCurveOracle` contract are used to `read` the current `USDC/ETH` and `FEI/ETH` prices respectively. These prices are used throughout the protocol to peg the price for `FEI`, and so are of critical importance in the system.

While the oracle is updated using the `update` function in a number of function calls, there are some calls within the system that read the price without updating it first. We understand that the Fei team is intending to manually update the oracle, however this poses a risk during times of high-volatility or high network-congestion.

Locations in the code that do not update the oracle price include:

- Within the function `reweight` in the `EthUniswapPCVController` contract, the function first determines whether `reweightEligible`. To determine this, it calculates the deviation from the current peg. However this peg is fetched without first updating the Uniswap Oracle. This means that the determination of whether to reweight is based on an outdated price, and the protocol could end up not reweighting when it should.
- The call to `isAtMaxPrice` to determine whether the `GenesisGroup` can execute `launch` reads the `peg` without updating it. This might cause the protocol to launch before the max has been reached, or to not launch when it should.
- Functions in the `BondingCurve` contract that rely on the `peg`, for example `getCurrentPrice`, and `getAdjustedAmount`.
- The `thaw` function in the `BondingCurveOracle` which calculates the thawed FEI price based on the current Uniswap price.

Consider ensuring that all function calls that read an Oracle price execute a call to `update` before reading it.

[H03] Incentive to support the peg decays with volatility

The `FEI` protocol incentivizes supporting the peg by minting extra `FEI` to users that buy `FEI` from the Uniswap pool. The incentive is a function of the growth rate, the time since the peg was last restored, and the distance from the peg. A `struct` containing a `blockNo`, `weight`, `growthRate`, and `active` flag tracks this info in the `timeWeightInfo` variable. The `timeWeightInfo.growthRate` is initialized as `333`, and only updated through governance.

The updated weight and price deviations are inputs to the `updateTimeWeight` function, which updates the `timeWeightInfo.weight` through the `_setTimeWeight` function. Whenever a buy is performed, the price deviation and updated weight is factored into the `calculateBuyIncentiveMultiplier` function to determine the incentive, whereas the `weight` is not factored into the `calculateSellPenaltyMultiplier` function which calculates the disincentive for selling. Upon each transaction, `incentivize` is called, which calls either `incentivizeBuy` or `incentivizeSell` depending on the direction of the trade. Regardless of the direction, `updateTimeWeight` is called, but this function differs for buy and sell transactions, with an asymmetric effect that reduces the incentive for buyers when there is price volatility below parity.

Suppose, for example, the protocol starts at parity, which sets the `weight` to `0`. Then the following sequence of transactions occur:

1. A large sell pushes the price down to `0.9`. First `getTimeWeight` will be called, and return `0`, because no blocks have passed since the peg was last at parity. In the same transaction, `updateTimeWeight` is called with parameters `weight = 0`, `initialDeviation = 0`, `finalDeviation = 0.1`. As a result, `updateTimeWeight` calls `_setTimeWeight` to update `timeWeightInfo.weight` to `0` and `timeWeightInfo.blockNo` to the current block height.
2. Next, a hundred blocks later, a ‘partial’ buy is submitted, which would bring the price up to `0.95`. First, `getTimeWeight` will be called. The `blockDelta` will now be `100`, and

the parameters: `weight = 33300`, `initialDeviation = 0.10`, and `finalDeviation = 0.05`. Since the peg was only partially restored, `updateTimeWeight` reduces the weight by `remainingRatio = finalDeviation / initialDeviation = 0.5`. Then, when `_setTimeWeight` is called, `timeWeightInfo.weight` is updated to `33300 * 0.5 = 16650`, and the current block height is stored in `timeWeightInfo.blockNo`.

3. One block later, another sell happens, pushing the price back down to `0.9`. This time, when `getTimeWeight` is called, the `blockDelta` will be `1` and `timeWeightInfo.weight` will be `16650`, so the function returns a weight of `16650 + 1 * 333 = 16983`. In the same sell transaction, `_setTimeWeight` is called, updating `timeWeightInfo.weight` to `16983` and updating `timeWeightInfo.blockNo`.
4. One block later, with the price back down to `0.9`, suppose a buyer makes a purchase of any size. When `getTimeWeight` is called, the `blockDelta` is `1`, and the `timeWeightInfo.weight` is `16983`, so the function returns `16983 + 1 * 333 = 17316`.

The weight is factored into the incentive for buyers. The higher the weight, the greater the incentive. When the first buyer submitted an order where the initial price was `0.9`, the weight was `33300`. When the second buyer submitted an order two blocks later with the same initial price of `0.9`, the incentive had dropped to `17316`. This is because partial buys reduce the `timeWeightInfo.weight` by the ratio of the partial buy, but sells do *not* increase the `timeWeightInfo.weight` by the inverse ratio. Buyers only realize the full incentive if the *first* buyer makes a purchase that *fully* restores the peg to parity, or if sells never happen after partial buys. Otherwise, the incentive for buyers to support the peg decays with price volatility below parity.

If the incentive decays such that buyers are no longer incentivized to support the peg, the `PCVController` would have to step in to re-establish parity using `forceReweight`.

Consider updating the sell functions to re-incentivize subsequent buying to support the peg.

Update: *Acknowledged. The Fei Team states that this is an expected behavior of the protocol.*

The `GenesisGroup` contract calculates whether or not the `maxGenesisPrice` has been reached using `address(this).balance`. While this balance is usually indicative of how much ETH has been deposited using the `purchase` function, malicious actors can increase the contract's ETH balance directly using `selfdestruct`. Performing a `selfdestruct` allows the attacker to bypass the checks and token minting performed in an execution of `purchase`, merely altering the balance directly.

While incredibly unlikely, if an attacker were to `selfdestruct` enough ETH into the `GenesisGroup` for the contract to reach the `maxGenesisPrice`, they could call `launch` to bootstrap the protocol while leaving the `FGEN.totalSupply` at 0. With 0 `FGEN` in existence, no users would be able to exeucte `redeem` to extract the `FEI` and `TRIBE` balances from the contract, leaving them locked inside. If an attacker instead deposited a smaller amount of ETH using `selfdestruct`, they would cause each valid contributor to receive more `FEI` and `TRIBE` than they were intended to receive.

While such an attack has no clear motivation, it can still affect the state of the system. Consider performing price calculations using `FGEN.totalSupply` instead of `address(this).balance` in both `isAtMaxPrice` and `launch` so that a `selfdestruct` would not affect the system.

Update: *Not fixed. In the words of the Fei team: "Did not fix because it is an obscure vector which is pure self-inflicted loss".*

[M02] Inverse price fetched from Uniswap

The `UniswapOracle` contract keeps track of the Uniswap price of `USDC` for use throughout the protocol and stores the price as USDC-per-WETH. The `update` function checks the price on Uniswap and, subject to some conditions, updates the price in the oracle. When checking the Uniswap price, Uniswap returns `price0Cumulative` and `price1Cumulative`, which contain the WETH-per-USDC and USDC-per-WETH prices, respectively. The boolean `isPrice0` is then used to flag whether Uniswap's `price0Cumulative` is the correct price to use in the oracle – and if `isPrice0` is `false`, the oracle instead uses `price1Cumulative`.

Due to the fact that the update function later performs an inversion, the final price is correctly stored as USDC-per-WETH. However, the inversion may cause a loss of accuracy on the USDC-per-WETH rate. Furthermore, the code is confusing to understand due to the false statements about what is being fetched from Uniswap and the lack of explanation regarding the inversion.

Consider fetching the USDC-per-WETH from Uniswap, as implied by the code, and updating the logic to no longer invert the fetched price.

Update: *Fixed in PR#21.*

[M03] Rounding errors in `Roots` library reduce `FEI` received from bonding curve

The `Roots` library implements a `cubeRoot` function. It then implements a `twoThirdsRoot` function which calls `cubeRoot`. This `twoThirdsRoot` function can have large rounding errors due to the order of operations. The `cubeRoot` function returns a truncated integer, which is then squared in the `twoThirdsRoot` function. The effect is that the `twoThirdsRoot` function is biased downwards, potentially significantly depending on the scale. For example, `twoThirdsRoot(124)` returns 16, whereas the true value is approximately 24.87.

The `Roots` library also implements a `threeHalfsRoot` function which suffers the same downward bias caused by incorrect order of operations. This `threeHalfsRoot` function calls the `sqrt` function which returns a truncated value. The `threeHalfsRoot` function then raises the truncated value to the power of three. Again, results may significantly deviate from expected values. For example, `threeHalfsRoot(8)` returns 8, whereas the true value is approximately 22.63.

The `threeHalfsRoot` and `twoThirdsRoot` functions are called from the `_getBondingCurveAmountOut` function, which is used to calculate the `amountOut` for every purchase made before the protocol reaches scale via the `getAmountOut` function.

As is, the `twoThirdsRoot` and `threeHalfsRoot` functions scale well, with relatively small percentage errors on big numbers. By changing the order of operations in the `twoThirdsRoot`

test fails after increasing the accuracy of the `twoThirdsRoot` and `threeHalfsRoot` functions.

Consider combining the arithmetic functions and changing the order of operations such that truncating steps are performed last while being mindful of potential overflows.

Update: *Fixed in PR#31. In the words of the Fei team: "This change reduced the error by 1-2 orders of magnitude on the number ranges in this test class".*

[M04] Incorrect proposal and quorum thresholds for voting

`Tribe` tokens are used for governance, with the `totalSupply` set to 1 billion tokens with `18 decimals`. The amount of `Tribe` required to reach the `proposalThreshold` for a vote is only `0.01%` of `Tribe` supply, although the comments indicate this should be `1%`. Likewise, in order for a vote to succeed, the `quoromVotes` requires only `0.1%` of `Tribe` supply, although the comments indicate this should be `10%`. These inconsistencies should be resolved.

Update: *Fixed in PR#. The amount of quorum votes was changed to 25,000,000 (2.5% of Tribe's total supply), and the proposal threshold was changed to 2,500,000 (0.25% of Tribe's total supply).*

[M05] ETH and `FEI` can get locked in `EthUniswapPCVDeposit`

The `deposit` function in `EthUniswapPCVDeposit` takes an ETH amount, mints an amount of `FEI`, and uses these to add liquidity to the ETH/FEI Uniswap Pool. The precise amount of `FEI` to mint is calculated in the `_getAmountFeiToDeposit` function.

When accepting liquidity, the Uniswap contract calculates the precise ratio needed to maintain the current price, and will not accept liquidity in a different ratio. In the case of a token, this means Uniswap fetches the exact token amount using `transferFrom`, and, in the case of ETH, Uniswap returns any excess ETH deposited to the sender. The `deposit` function does not handle the case where Uniswap does not accept 100% of the ETH and `FEI` as liquidity. This means that if `_getAmountFeiToDeposit` calculates the incorrect amount of `FEI` and ETH to deposit, `FEI` or ETH will become locked inside the `EthUniswapPCVDeposit` contract.

- Fei's `_getAmountFeiToDeposit` and Uniswap's `_addLiquidity` calculate the amounts to deposit slightly differently: Fei calculates the `FEI` needed given the reserve sizes and an ETH amount, whereas Uniswap calculates the ETH needed given the reserve sizes and a `FEI` amount. While this difference is minimal, it leads to small differences in the amounts to deposit.
- `_getAmountFeiToDeposit` does not always use the actual reserve amounts from Uniswap. The function `getReserves` alters the `FEI` reserve amount used in calculations if the Uniswap Pair's `FEI` balance is larger than the reserve amount, leading to a difference in the ratio used in calculations.

Consider implementing logic to handle any `FEI` or ETH that remains in the `EthUniswapPCVDeposit` contract after adding liquidity to Uniswap. Alternatively, consider altering the calculations to exactly mirror those performed in Uniswap.

Update: *Fixed in PR#30. The `EthUniswapPCVDeposit` now burns the dust `FEI` after adding liquidity to the ETH/FEI Liquidity Pool, and dust ETH is tacked onto the next deposit.*

[M06] Incentive does not initialize when Ether depreciates

Once `FEI` reaches parity with `USDC`, there are 2 ways for the `FEI` price to fall below the peg: either by users selling `FEI` (which the protocol handles) or by ETH depreciating against `USDC`. There is no incentive to support the peg if ETH depreciates against `USDC`, because:

- When the price is above the peg, `timeWeightInfo.active` is set to false.
- If the `FEI` price naturally drops without users trading (i.e., by ETH depreciation against `USDC`), nothing triggers updates to the `timeWeightInfo` variables.
- The `getTimeWeight` function returns `0` when `timeWeightInfo.active` is false.
- The `calculateBuyIncentiveMultiplier` function returns `0` when weight is `0`, regardless of the deviation.

Nothing forces the time-weighted incentive to become active except trading. Therefore, if `FEI` depreciates because Ether has depreciated against `USDC`, nothing incentivizes buyers to support the peg. Some trading activity would need to happen (i.e., selling, or an unincentivized buy) to start the time-weighted incentive. In such a case where a trade happens after Ether depreciates, the

intending to back-run those transactions to capture an incentive.

Consider adding an incentive for buys when the initial price is below the peg even if the time-weighted period has not yet been initialized.

Update: *Acknowledged. In the words of the Fei Team: We acknowledge that this is an issue but we expect natural arbitrage and trading activity to initialize it within a reasonable time frame.*

Low severity

[L01] Approved addresses can forfeit user rewards

Holders of `FPOOL` tokens are entitled to rewards from the `FeiPool` contract. When the rewards have been collected using the `claim` function, the user's `FPOOL` tokens are burned. The situation is analogous with `FGEN` tokens in `GenesisGroup`, with rewards collected using `redeem`, which burns the `FGEN` tokens.

As with all ERC20 tokens, holders of `FPOOL` and `FGEN` can `approve` other addresses to transfer their tokens on their behalf. However, with `FPOOL` and `FGEN`, these approved addresses are given an extra power: they can burn the user's tokens, forfeiting reward collection. This forfeiture of user rewards also increases the amount of rewards other token holders are entitled to, as rewards are split proportionally between holders.

While it is assumed that users trust the addresses that they are approving, these external `burnFrom` functions seem to serve no purpose in the system, and so the increased risk could be removed.

Consider changing the `FPOOL` and `FGEN` tokens to be `ERC20` instead of `ERC20Burnable`. Functions that need to perform token burns can use the internal `_burn` function to do so, and additional `allowance` checks can be added if a third party is calling the function.

Update: *Fixed in PR#32. The `FeiPool` contract no longer inherits from the `ERC20Burnable` contract. Additionally, the `burnFrom` public function was removed, and a new `_burnFrom` function was added to be used in the `_withdraw` function.*

The Protocol uses a "Burner" role to manage permissions around token burning, which the incentivization model requires.

The `burnFrom` function in the `Fei` contract uses the internal `_burn` function from the OpenZeppelin `ERC20` implementation instead of using the `burnFrom` function from the OpenZeppelin `ERC20Burnable` implementation.

The `_burn` function from `ERC20` does not check allowances before performing the burn action. Currently, calling the `burnFrom` function from the `Fei` contract will allow a Burner to burn *any amount* of `FEI` from *any account* without restriction. In the event that a Burner role address gets compromised, this could present more risk to the protocol than necessary. On the other hand, the more restrictive `burnFrom` function from `ERC20Burnable` considers allowances before burning tokens, which could help lessen the impact of a rogue Burner.

For the `FEI` token, consider using the `burnFrom` function from `ERC20Burnable` rather than the `_burn` function from the `ERC20` contract to minimize risk.

Update: *Acknowledged. In the words of the Fei team: "We feel that this risk is appropriate as the burner should not need ERC20 approval and is only given to one contract".*

[L03] Core contract cannot allocate all Tribe tokens

The function `allocateTribe` in the `Core` contract is used to distribute `TRIBE` tokens to specified addresses. A percentage of `TRIBE` tokens remains undistributed as a treasury. According to the Fei whitepaper, *the community can distribute this as it sees fit as the protocol develops.*

However, in the allocation process, the function requires that the amount to be distributed is *strictly* less than the contract's `TRIBE` balance. This means that the contract cannot allocate all of the `TRIBE` tokens it holds, and a single unit of `TRIBE` will be locked in the contract.

Consider changing the require statement to use >= instead of >, so that all of the `TRIBE` it holds can be allocated.

Update: *Fixed in PR#33.*

`truffle-config` file specifies version `0.6.6`, which was released on April 6, 2020. Throughout the codebase there are also different versions of Solidity being used. For example, most of the Solidity files specify supported versions `^0.6.0`, while the following files specify supported versions `^0.6.2`:

- `IPCVDeposit.sol`
- `IUniswapPCVController.sol`
- `IFei.sol`
- `IIncentive.sol`
- `IUniswapIncentive.sol`

As Solidity is now under a fast release cycle, consider using a more recent version of the compiler, such as version `0.7.6`. In addition, to avoid unexpected behavior, consider specifying explicit Solidity versions in pragma statements.

Update: *Not fixed. In the words of the Fei Team: "It is not possible due to dependence on solidity compiler version 0.6.6 due to Uniswap interfaces".*

[L05] Not checking for `0` addresses

Some constructors do not check that an initialized `address` is not `0`. This could result in loss of control or locked funds. Examples include:

- `beneficiary` address in the `LinearTokenTimelock` constructor
- `admin` address in the `Timelock` constructor

Consider validating that addresses are not `0` to ensure contracts operate as intended.

Update: *Fixed in PR#39.*

[L06] Not using `SafeMath` and `SafeCast`

Throughout the codebase there are math operations that are not checked for overflow or underflow using the `SafeMath` library. Examples can be found in the following locations, however this list is not exhaustive:

- The `_beforeTokenTransfer` function
- The `checkAllocation` function
- The `getFinalPrice` function

Although we did not observe instances which appear at high risk of overflow, consider checking all math operations using `SafeMath` as a best practice. In addition, the `getFinalPrice` function casts an `int256` value to `uint256` without any checks. Consider using the `SafeCast` library to check casting operations.

Update: *Partially fixed in PR#37 and PR#67. There are still some ocurrencies where `SafeMath` and `SafeCast` are not being used.*

[L07] `queueTransaction` does not check transaction value

The `queueTransaction` function uses ETH in the `Timelock` contract to execute transactions. The contract is funded using the `receive` function, which can then be used to execute queued transactions. However, there is no easy way to withdraw ETH from the contract, and `queueTransaction` does not validate that ETH being sent in a queued transaction was provided for that purpose.

Consider making `queueTransaction` a `payable` function and validating `msg.value == value` to ensure proper allocation of funds. In addition, consider implementing a way to withdraw any remaining ETH from the contract after confirming success in the `executeTransaction` function.

Update: *Not fixed. In the words of the Fei Team: "We want to minimize changes to forked DAO contracts".*

[L08] Re-implementing ECDSA signature recovery

The following functions include implementations of the ECDSA signature recovery function:

- `castVoteBySig` in the `GovernorAlpha` contract
- `delegateBySig` in the `Tribe` contract
- `permit` in the `Tribe` contract

constantly reviewed by the community. Consider importing and using the recover function from OpenZeppelin's `ECDSA` library not only to benefit from bug fixes to be applied in future releases, but also to reduce the code's attack surface.

Update: *Not fixed. In the words of the Fei Team: "We want to minimize changes to forked DAO contracts".*

[L09] Transfers are not checked for success

The `transfer` and `transferFrom` functions return `true` upon success, but the return values are not checked when these functions are called. For example, `transferFrom` is called from the `_deposit` function and `transfer` is called from the `_withdraw` function, but these result are not checked for success. Currently, the protocol only handles ETH, `FEI`, and `TRIBE` tokens. However, if planning to incorporate other tokens in the future, they may not revert on failed transfers, thereby causing these functions to continue operation when they should abort. Consider using `SafeERC20` to catch failed transfers.

Update: *Partially fixed in PR#12. An assertion was added when calling the `transfer` function from the `WETH` contract. The Fei Team decided not to check the `FEI` and `TRIBE` `transfer` and `transferFrom` return values since they inherit their behavior from the openzeppelin-contracts' `ERC20.sol` contract, which will revert if something goes wrong.*

[L10] `UniswapOracle` does not allow overflow

The `update` function in the `UniswapOracle` contract does not allow the cumulative price variables to overflow, against the guidance of the UniswapV2Pair oracle specification. This means the `update` function will eventually fail, despite the fact that Uniswap is designed to handle this overflow correctly. Consider permitting overflows to conform to the Uniswap Oracle specification.

Update: *Fixed in PR#37.*

Notes & Additional Information

[N01] Commented out code

implementation, it may be better to track them in a separate document for discussion.

Update: *Fixed in PR#34.*

[N02] Functions `return` without parameters specified

The `delegate` and `delegateBySig` functions within `Tribe.sol` execute the `_delegate` function in `return` statements. However, `delegate`, `delegateBySig`, and `_delegate` do not have `return` parameters, so the use of `return` in these locations is confusing and misleading. Consider replacing these `return` statements with simple function calls.

Update: *Acknowledged. In the words of the Fei Team: "Want to minimize changes to forked DAO contracts".*

[N03] `GenesisGroup` is a token unnecessarily

The `GenesisGroup` contract is a token, however this token's use will last a maximum of 3 days. The `GenesisGroup` contract `mints` FGEN `tokens` to users that deposit collateral, and then burns all `FGEN` tokens when users `redeem` them for `FEI` and `TRIBE` tokens. Consider accounting for this GenesisGroup token allocation using a simple mapping, rather than by creating a separate token.

Update: *Acknowledged. The Fei Team decided to keep the `GenesisGroup` as a token, since they expect secondary markets to exist to allow users exiting Genesis by selling their `FGEN` tokens.*

[N04] Improper use of `require`

As outlined in the Solidity docs, `require` statements are to *ensure valid conditions that cannot be detected until execution time*, whereas `assert` statements should be *to check invariants*. Properly functioning code should never reach a failing `assert` statement.

The codebase includes several `require` statements used to check for conditions that should never happen. For example, `require` statements are used to check whether the `total` is

would constitute a failure of the internal system, this condition would correctly be checked by an `assert` statement. Consider updating error handling statements that check for properly functioning code to use `assert` statements instead of `require`.

Update: *Fixed in PR#51.*

[N05] Inconsistencies around time

While much of the codebase uses timestamps to measure time, some time intervals are measured instead as a number of blocks. In these instances, comments are used to indicate the time interval desired that led to the specified number of blocks, however, these comments use inconsistent methods of estimation.

For instance, in `GovernorAlpha` the `votingPeriod` value, 17280, is accompanied by inline documentation that assumes 15-second block times. In `CoreOrchestrator`, the `INCENTIVE_GROWTH_RATE` value has inline documentation that assumes 12-second block times. The whitepaper and other project documentation generally match this inline documentation.

Even when a number of blocks is converted to a time interval using a consistent block time assumption, those approximations can deviate from reality. Inconsistent block time assumptions only make things more confusing. Block times can and will be variable, and the actual current block times of the Ethereum network have been closer to ~13s. This means that the actual time intervals observed in production could deviate from the documentation by as much as 13%. For example, the `votingPeriod` value is documented as being equivalent to approximately three days, when, in reality, it would be closer to two days and fourteen hours – a significant deviation that could result in user confusion.

To reduce confusion and increase the predictability of time intervals, consider using block timestamps for time intervals where possible. Alternatively, document assumptions about block times clearly and consistently, and be sure to explicitly reflect the variability of the time intervals they represent.

Update: *Fixed in PR#50. Now, block times are assumed to be of ~13s.*

[N06] Inconsistent coding style

While most `public` function names do not contain an underscore, some begin with one underscore and others begin with *two* underscores. For example:

- The `_feiTribeExchangeRate` function
- The `__acceptAdmin` function
- The `__abdicate` function

Some `internal` function names start with an underscore, while others do not. For example:

- `_writeCheckpoint` and `safe32` are both `internal` functions in the `Tribe` contract.
- `_setBeneficiary` and `setLockedToken` are both `internal` functions in the `LinearTokenTimelock` contract.

Some parameters end with an underscore, while most do not. For example:

- The `delay_` parameter in the `setDelay` function
- The `pendingAdmin_` parameter in the `setPendingAdmin` function

Some lines of code are very long. For example:

- The `propose` function definition is 139 characters long.
- The `getActions` function definition is 164 characters long.

Some string literals are surrounded by double quotes (`"`), while others are surrounded by single quotes (`'`). For example:

- The `Fei` constructor uses double quotes to pass strings to the `ERC20` function.
- The `Fei` constructor uses single quotes to pass a string to the `keccak256` function.

There is mixed use of spaces and tabs for indentation. For example:

- The `incentivize` function uses a mixture of spaces and tabs for indentation, sometimes on the same line.

Some functions use named return variables, while others do not. For example:

variable for the returned value.

Consider enforcing a standard coding style, such as that provided by the Solidity Style Guide, to improve the project's overall legibility. Also consider using a linter like Solhint to define a style and analyze the codebase for style deviations.

Update: *Fixed in PR#54.*

[N07] Inconsistent error message format

Error messages in the codebase follow different formats. In particular, messages from the `dao` contracts conform to the format `Contract_Name::Function_Name: message`, while other messages (like those in the `GenesisGroup` contract) conform to the format `Contract_Name: message`. To improve readability and facilitate debugging, consider following a consistent format across all error messages.

In addition, some error messages reference an incorrect function name, such as those found in the `transferFrom` function, the `_moveDelegates` function, and the `Timelock` constructor. These should be resolved.

Update: *Fixed in PR#47.*

[N08] Inconsistent methods for retrieving cumulative price from Uniswap oracle

The `update` function in the `UniswapOracle` contract uses the `currentCumulativePrice` function of the `UniswapV2OracleLibrary` library to retrieve the cumulative prices and timestamp from the Uniswap oracle, whereas the `_init` function in the same contract retrieves the same values using separate individual functions. To improve legibility and facilitate refactoring, consider using a consistent method for retrieving cumulative prices from the Uniswap oracle.

Update: *Fixed in PR#46.*

[N09] Incorrect GovernorAlpha name constant

governance token.

Update: *Fixed in PR#35.*

[N10] Interfaces omit some external functions

Throughout the codebase there are instances of `interface` contracts omitting some of the `public` or `external` functions that their corresponding implementation contracts define. Some examples include:

- `IBondingCurveOracle` omits the `initialPrice` function that is implemented in `BondingCurveOracle`.
- `IFei` omits the `permit` function that is implemented in `Fei`.
- `IGenesisGroup` omits the `burnFrom` function that is implemented in `GenesisGroup`.
- `IPool` omits the `burnFrom` function that is implemented in `Pool`.
- `IUniswapPCVController` omits the `minDistanceForReweight` that is implemented in `EthUniswapPCVController`.

Incomplete interfaces may introduce confusion for users, developers, and auditors alike. To improve overall code legibility and minimize confusion, consider modifying the interface contracts to reflect all of the `public` and `external` functions from their respective implementation contracts.

Update: *Fixed in PR#53.*

[N11] Uninitializable global variable in `LinearTimelockToken`

The `lockedToken` global variable in the `LinearTokenTimelock` contract is only initializable by the `internal` `setLockedToken` function. As is, the `lockedToken` variable can only be initialized by inheriting the contract and calling the `setLockedToken` function. Consider making the contract `abstract` to indicate that inheriting contracts need to complete the implementation. Alternatively, consider initializing the `lockedToken` variable in the `LinearTokenTimelock` constructor.

Update: *Fixed in PR#39.*

include:

- The literal value `10000` on line 22 of `BondingCurveOrchestrator.sol`
- The literal value `-1` that is often used to represent an approval of 'infinite tokens' in `Tribe.sol` and throughout the codebase
- The literal value `2**112` on line 60 of `UniswapOracle.sol`

Literal values in the codebase without an explained meaning make the code harder to read, understand and maintain for developers, auditors, and external contributors alike.

Consider defining a `constant` variable for every magic value used, giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended.

Update: *Partially fixed in PR#41. Only the occurrences of magic constants mentioned above were explicitly declared.*

[N13] Missing and incomplete event emissions

Several constructors do not emit events after initializing sensitive variables in the system, but when those variables are updated using setter functions, an event is emitted. For example:

- The `Core` contract constructor does not emit the `FeiUpdate` event.
- The `OracleRef` contract constructor does not emit the `OracleUpdate` event.
- The `BondingCurve` contract constructor does not emit the `ScaleUpdate` event.

Some setters, like the `setGenesisGroup` function, do not emit events. Whereas others, like the `setFei` function from the same contract, do emit an event.

Consider emitting events for all state changing functions, including those in contract constructors. In addition, consider emitting the old and new values in these `XUpdate` events to help track changes (e.g. `ScaleUpdate(uint256 _oldScale, uint256 _newScale)`).

Update: *Partially fixed in PR#52. The Fei team elected not to change events to include old update values.*

Our suggestions are to rename:

- `state` to `getProposalState`.
- `releaseWindow` to `releaseWindowDuration`.
- `timestamp` to `timeSinceStart`.
- `d` and `t` to `duration` and `timePassed`.
- `dst` throughout Tribe.sol to `destination`.
- `price0` in the `PCVDepositOrchestrator` and corresponding `interface` to `isPrice0`.
- `_twfb` to `_timeWeightedFinalBalance`.
- `threeHalfsRoot` to `threeHalvesPower` or `twoThirdsRoot`.
- `twoThirdsRoot` to `twoThirdsPower` or `threeHalvesRoot`.
- `calculateDeviation` to `deviationBelowPeg`.

Consider renaming these parts of the contracts to increase overall code clarity.

Update: *Fixed in PR#43 and PR#31. Some of the suggestions above were implemented by the Fei team.*

[N15] NatSpec comments missing

Many functions do not have NatSpec comments (such as those in the `Timelock` and `LinearTokenTimelock` contracts). Furthermore, some functions do not have any comments, for example those in the `BondingCurve` contract. While many of these functions implement an interface, where the interface *does* include NatSpec comments, there are several exceptions leaving some code undocumented. In addition, it may improve readability to provide NatSpec comments on the implemented function, rather than on the interface definition. Consider adding NatSpec comments to all public and external functions, and including more comments throughout function implementations.

Update: *Partially fixed in PR#56 NatSpec comments were moved from the interfaces to the implementations, but some of these are still incomplete.*

[N16] Using `now` instead of `block.timestamp`

Solidity. This is observed in the following contracts:

- Line 55 of `Core.sol`
- Line 123 of `GenesisGroup.sol`
- Line 40 and line 46 of `Timed.sol`
- Line 175 and line 251 of `Tribe.sol`

Consider using `block.timestamp` for clarity and to facilitate future upgrades.

Update: *Fixed in PR#42.*

[N17] Proposals can be canceled in states that are unintuitive

The `GovernorAlpha` contract is a fork of a Compound contract by the same name. It is responsible for governance proposals, including proposal vote collection, creation, cancellation and execution.

The public `cancel` function allows a proposal to be canceled only if certain conditions are met. Specifically, if a proposal has already been executed, then it cannot be canceled. However, a proposal that has *already* been canceled can be canceled again, so too can a proposal that's been defeated or has already expired.

To ensure proposal states align with user expectations, and to avoid any confusion for outside observers, consider also disallowing cancellations for proposals that are canceled, defeated or expired.

Update: *Fixed in PR#61. Now,* `GovernorAlpha` *proposals can only be cancelled in* `Active` *or* `Pending` *states.*

[N18] Proposal struct storage is inefficient

The `Proposal` struct uses 13 storage slots. To improve gas efficiency, consider reorganizing the attributes within the struct to reduce the storage to 12 slots, for example, by moving the `bool` `canceled` and `executed` attributes immediately after the `proposer` `address`.

[N19] Redundant event definition

The `KillSwitchUpdate` `event` is defined on line 23 of the `BondingCurveOracle` contract. That contract inherits from `IBondingCurveOracle`, which inherits from `IOracle` where the same `KillSwitchUpdate` `event` is defined on line 12.

Consider removing redundant event definitions to improve overall code clarity and maintainability.

Update: *Fixed in PR#52.*

[N20] Some interfaces are not inherited

There are numerous interfaces that are not inherited by relevant contracts. Some examples include:

- The Orchestrator interfaces in the `CoreOrchestrator` contract. For instance, the `IIDOOrchestrator` and the `IGenesisOrchestrator` interfaces, which are not inherited by the `IDOOrchestrator` or `GenesisOrchestrator` contracts, respectively.
- The `TimelockInterface` interface, which is not inherited by the `Timelock` contract.
- The `CompInterface` interface, which is not inherited by the `Tribe` contract.

Consider having contracts inherit from relevant interfaces wherever possible.

Update: *Fixed in PR#7 and PR#10. The* `orchestrator` *variable was removed from the* `GenesisGroup` *contract, and the* `operator` *variable is now being used to check that the operator is allowed to perform sell operations.*

[N21] Some interfaces are unnecessary or inconsistent with implementation

There are some interfaces used that may be unnecessary or are inconsistent with their implementations.

- For instance, the `IOrchestrator` interface is defined and used for the `orchestrator` variable, but this `orchestrator` variable is never used.

Consider either removing unused interfaces and arguments, fully implementing them, or modifying them to be consistent with their implementations where appropriate.

Update: *Fixed in PR#53.*

[N22] Test and production constants in the same codebase

The `CoreOrchestrator` contract defines the `TEST_MODE` boolean variable which is used to define several constants in the system. This decreases legibility of production code, and makes the system's integral values more error-prone. Consider instead having different environments for production and testing, with different contracts.

Update: *Fixed in PR#38. All the test constants were removed from the master branch.*

[N23] Unnecessarily small integer sizes

In Solidity, using integers smaller than 256 bits tends to increase gas costs because the Ethereum Virtual Machine must perform additional operations to zero out the unused bits. This can be justified by savings in storage costs in some scenarios, however, that is not generally the case in this codebase. In several contracts, unsigned integers are being unnecessarily sized less than 256 bits. Examples include:

- The `uint128` variables in the `Pool` contract
- The `uint96` variables in the `Tribe` contract
- The `uint32` variables in the `Timed` contract

In some instances, these smaller integer sizes can cause function reverts earlier than necessary. In particular, since the `_timestamp` and `_initTimed` functions cast block times to `uint32` values, functions using the `Timed` contract to manage control flow will revert beginning February 7, 2106 at 6:28:16 AM GMT.

Consider using integers of size 256 bits to improve gas efficiency and mitigate function reverts.

possible.

[N24] Unnecessary if statement

The `availableForRelease` function fetches the elapsed time since the start timestamp using the `timestamp` function inherited from the `Timed` contract. It then checks if this is larger than `duration`, and if so, caps the value at `duration`. However, the `timestamp` function from the `Timed` contract already caps the result to `duration`, so this `if` statement will never be executed and is unnecessary. Consider removing this `if` statement.

Update: *Fixed in PR#8.*

[N25] Unnecessary imports

Some Solidity files are unnecessarily imported.
– `Core.sol` imports `IFei.sol` unnecessarily, because `Fei.sol` already imports `IFei.sol`.
– `Core.sol` imports `IERC20.sol` unnecessarily, because `IFei.sol` already imports `IERC20.sol`.
– `Pool.sol` imports `SafeMathCopy.sol` without using it.
– `BondingCurve.sol` imports `AccessControl.sol` without using it.
– `EthUniswapPCVController.sol` imports `IOracle.sol` without using it.
– `UniswapIncentive.sol` imports `IOracle.sol` without using it.

Consider removing redundant and unused imports to improve legibility.

Update: *Fixed in PR#45.*

[N26] Unreachable, incorrect error message

In the `calculateDeviation` function in the `UniRef` contract there is a subtraction with a provided error message argument. This error message is logically unreachable, but it is also incorrect. It currently reads “UniRef: price exceeds peg”, but should read “UniRef: *peg* exceeds *price*“.

Consider removing or correcting the error message to improve code clarity.

The following contracts inherit both `ERC20` and `ERC20Burnable`:

- `Fei`
- `Pool`
- `GenesisGroup`

Inheriting `ERC20` is unnecessary where `ERC20Burnable` is inherited, because `ERC20Burnable` already inherits `ERC20`. Inheriting both contracts can be confusing, especially where calls to `super` functions are made. To improve readability of these contracts, consider removing the `ERC20` inheritance.

Update: *Fixed in PR#32.*

[N28] Use of `uint` instead of `uint256`

Across the codebase, there are hundreds of instances of `uint`, as opposed to `uint256`. In favor of explicitness, consider replacing all instances of `uint` with `uint256`.

Update: *Fixed in PR#54.*

Conclusions

5 critical and 3 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits