PUBLIC

Code Assessment

of the AMM, Gauges and Bribes Smart Contracts

April 24, 2023

Produced for

USDFI

by

CHAINSECURITY

Contents

1 Executive Summary

Dear USDFI Team,

Thank you for trusting us to help USDFI with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of AMM, Gauges and Bribes according to Scope to support you in forming an opinion on their security risks.

USDFI implements an ecosystem, which supports exchanges, including both constant (as in Uniswap V2) and stable swap. To incentivise liquidity providers to get engaged in the system, a bribing system as well as gauges are implemented to allow staking of LP tokens.

During the review, no critical or highly severe issues were uncovered.

The most critical subjects covered in our audit are functional correctness, access control and signature malleability. The security regarding all the aforementioned subjects is high.

The general subjects covered are gas efficiency, code complexity, testing, and specification quality. Note that in the third version tests were added. The quantity and quality of tests, however, see Lack Of Testing, and gas efficiency can still be further improved. In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4
• Code Corrected	4
Low-Severity Findings	11
• Code Corrected	6
• Specification Changed	1
• Risk Accepted	2
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the AMM, Gauges and Bribes repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	6 March 2023	eb5a9257b27b7ff94a2acda1ecd9b89b38de1120	Initial Version
2	4 April 2023	3830ed03c03cdc7bffcce567544b2224850df693	Fixes
3	20 April 2023	bfc6e47890e6d7a801df6286c34c6d67d8d79c1e	Further Fixes

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

The following files are in scope:

```
Audit/bribe_factory/Bribe.sol
Audit/bribe_factory/BribeFactory.sol
Audit/bribe_factory/IGaugeFactory.sol
Audit/bribe_factory/IReferrals.sol
Audit/bribe_factory/Math.sol
Audit/gauge_factory/Gauge.sol
Audit/gauge_factory/GaugeFactory.sol
Audit/gauge_factory/IBaseBribeFactory.sol
Audit/gauge_factory/IBaseFactory.sol
Audit/gauge_factory/IBasePair.sol
Audit/gauge_factory/IBribe.sol
Audit/gauge_factory/IGaugeFactory.sol
Audit/gauge_factory/IReferrals.sol
Audit/gauge_factory/IStableMiner.sol
Audit/gauge_factory/Math.sol
Audit/gauge_factory/ProtocolGovernance.sol
Audit/pair_factory/BaseV1Factory.sol
Audit/pair_factory/BaseV1Fees.sol
Audit/pair_factory/BaseV1Pair.sol
Audit/pair_factory/IBaseV1Callee.sol
Audit/pair_factory/Math.sol
```

In **Version 2**, the follwoing contracts were included in scope:

```
Audit/pair_factory/BaseFactory.sol
Audit/pair_factory/BaseFees.sol
Audit/pair_factory/BasePair.sol
Audit/pair_factory/IBaseCallee.sol
```

while the following contracts were removed from scope:

```
Audit/pair_factory/BaseV1Factory.sol
Audit/pair_factory/BaseV1Fees.sol
Audit/pair_factory/BaseV1Pair.sol
Audit/pair_factory/IBaseV1Callee.sol
```

2.1.1 Excluded from scope

All other files are excluded from scope. The interacted with contracts are expected to work as intended.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

USDFI offers an exchange that can implement either the constant product formula (as in Uniswap V2) or a stable swap formula as its invariant for swaps. Further, USDFI implements gauges to stake LP tokens and a bribing system.

2.2.1 Exchange

The trading pairs `BaseV1Pair` are deployed by `BaseV1Factory` through `createPair()`. The factory is pausable (disables swaps on pairs) and an owned contract that implements a two-phase ownership transfer scheme. Further, it allows the governance to activate a "gas-throttle mode" for all active trading pairs and to set the starting fee parameters and system-relevant addresses.

The trading pair `BaseV1Pair` acts as an Automated Market Maker (AMM) where Liquidity Providers (LPs) pool their funds, so that traders can trade against the pool according to invariant defined during the deployment. LPs call `mint()` to receive an LP token in return. Typically, minting is done in a proportional way to the present reserves. If it is done unproportionally, the deposit will be treated as a proportional mint, which is then followed by a donation with the overhead balance. Similarly, LPs can remove liquidity with `burn()`, which burns the LP token and returns funds proportional to the share that the LP had and according to the reserve amounts. Note that for both functions it is expected that funds have been transferred directly to the contract beforehands (expected to be done through a helping router contract).

Traders can swap tokens with the function `swap()`. The outcoming and incoming amounts for both tokens can be arbitrary, as long as the invariant is satisfied after the fees are deducted from the incoming amounts. Hence, `swap()` can be used for flash loans. The following invariants are defined:

- `x*y >= k` for non-correlated assets
- `x^3*y + x*y^3 >= k` for correlated assets

On every pool reserve-changing action, that is first in a block, the pool keeps track of the cumulative reserves of the previous block. Further, it stores a history of these actions as observations with at least 30 minutes time distance. Using these observations, the pools implement Time Weighted Average Price (TWAP) oracles.

Note that with using `skim()` overhead balances get out of the pool, while calling `sync()` donates them to the pool.

2.2.2 Gauges and Bribes

Gauges allow users to stake their LP tokens to receive `STABLE` rewards. These rewards are allocated epoch-wise by the factory through a voting mechanism, which allows users holding a balance in the `veProxy` to vote on the `STABLE` allocations for the gauges. To incentivise voters to vote for a gauge, a bribing mechanism is introduced that rewards voters with the accrued LP fees.

A gauge for a given LP token is deployed through the gauge factory with `addGauge()`. Along with the gauge, its bribe contract is deployed. Note that with `deprecateGauge()` and `resurrectGauge()`, a gauge can be deactivated or reactivated, respectively.

Besides the administrative functionality, the gauge factory implements the voting mechanism. Namely, voters can cast their votes once per epoch with `vote()` by specifying the distribution of their voting power among gauges or with `poke()`, which recasts the previous votes with the same distribution. Note that the factory creates virtual balance for the following epoch for the voter in the corresponding bribe contracts with `Bribe._deposit()`, so that the voter becomes egligible for bribes distributed in the following epoch. If no votes are cast in the subsequent epochs, the last votes are taken into account. However, note that the power of a user remains constant by design in such cases. Votes can be always removed with `reset()`, as it is a privilieged action, only admin, governance, and the voting admin are allowed to call it.

The reward distribution of `STABLE` rewards to gauges is a two-step process. First, `preDistribute()` locks the votes for the gauges, retrieves fresh `STABLE` from its minter and starts a new epoch. Next, `distribute()` distributes the rewards to the corresponding gauges with `Gauge.notifyRewardAmount()`, which stores the rate with which the reward will be emitted over the next seven days. Note that a gauge's timing does not necessarily match its factory's timing in terms of epochs.

Users that staked their LP token with `deposit()` (or the targets of `depositFor()`) will be egligible to receive rewards according to the aforementioned rates. Bear in mind, that with `withdraw()` the LP tokens can be withdrawn. However, the rewards amongst the users will not be distributed only according to the amounts they have staked. More specifically, the deposited amount will contribute 40% to a users's derived balances for rewards. The remaining 60% of the supply are used for a boosting mechanism for stakers that have voted (users' derived balances increase according to there voting weight for the gauge) receive more rewards (voting power of a user may range from 40% to 100% of his staked amount). Note that the derived balances may be out-of-sync. Hence, they can be refreshed with `kick()` for any user.

Ultimately, rewards can be retrieved with several `getReward()` variations. Consider that anyone can claim rewards for anyone with the condition that the recipient is the address that earned the rewards or that the recipient has been whitelisted by the owner through `setWhitelisted()`. It is worth mentioning, that a certain percentage of the rewwards will be distributed to referrals.

To incentivize voters, gauges implement `claimVotingFees()` that claims the LP fees from the LP token while notifying the bribe about the new rewards with `Bribe.notifyRewardAmount()` for both tokens of the pair which adds the reward for the voters of the current epoch to be received in the next epoch (recall the voting mechanism with `_deposit()`). Besides, the underlying tokens of the LP, additional tokens could be added as reward tokens by the governance. Similar to the gauges, rewards can be retrieved with variations of `getReward()`. In contrast to gauges, the unclaimed rewards in bribes disappear after 50 epochs for a user. Further, note that governance can recover ERC-20 tokens from the bribe contract that are not reward tokens.

2.2.3 Trust Model & Roles

- Governance: Fully trusted as they set meaningful parameters in the interest of the protocol and set up priveleged roles to trusted addresses.
- Users: Untrusted. Further, users are expected to understand how the system works (e.g. use meaningful pools and interact with helper contracts (e.g. routers) to perform actions on the pool).

- Tokens: Trusted. Expected to be non-rebasing tokens for the exchange. However, a malicious token could, in the worst case, drain the pairs in which it is present.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

- Distributing Multiple Times Is Possible Even in the Same Block Acknowledged
- Lost Rewards Acknowledged
- Poking May Revert Risk Accepted
- burn() Read-Only Reentrancy Risk Accepted

5.1 Distributing Multiple Times Is Possible Even in the Same Block

Correctness Low Version 1 Acknowledged

ISSUEIDPREFIX-001

The documentation specifies that the next distribution must be `7 days` away from the last distribution. However, this is not necessarily the case. Consider the following example:

1. `lastDistribute` is `block.timestamp - 2 weeks - 1`
2. `preDistribute()` is called. `lastDistribute` is now `block.timestamp - 1 weeks - 1`
3. Optionally, `distribute()` is called.
4. `preDistribute()` is called. `lastDistribute` is now `block.timestamp - 1`
5. Optionally, `distribute()` is called.

That may occur if the initial `lastDistribute` value is low (lack of sanity check in the constructor) or if the distribution is not called regularly. Ultimately, the specification is violated.

Further, the behaviour in such cases is unspecified.

Acknowledged:

USDFI has acknowledged this issue stating that:

> *Emergency option to resync time in case of epoch desync (ie, force majeur event)*

5.2 Lost Rewards

Design Low Version 1 Acknowledged

ISSUEIDPREFIX-002

The reward rate in gauges is typically computed as `reward / DURATION`. If `reward < DURATION` holds, the reward rate will be 0. Ultimately, the rewards are lost as they will not be accounted in in the future.

Acknowledged:

USDFI replied

> *Rewards must be lost to prevent griefing attacks that can occur due to non-shareable numbers!*

However, it could be possible to cache the non-distributable rewards so that they could be accounted for in the future.

5.3 Poking May Revert

Design Low Version 1 Risk Accepted

ISSUEIDPREFIX-003

Poking can revert when `_prevWeight * _weight < _prevUsedWeight` holds. That may be the case when the user had a big decrease in balance. Another scenario could be when the user had specified a small amount of weight allocated to a gauge (e.g. 1, balance decrease from 100 to 99). The revert occurs in the bribe where `_deposit()` and `_withdraw()` revert with zero amounts.

Ultimately, a user may not be poked anymore, which could lead to reverts in the gauge's `updateReward` modifier. However, the issue may be repaired by using `vote()` or `reset()`.

Risk accepted:

USDFI states:

> *Smaller amounts are not allowed by the voter, because of the linear drop no fast drops are possible (poking reversion is completely ruled out by the vote escrow function in the vote escrow contract (outside of audit scope)).*

5.4 `burn()` Read-Only Reentrancy

Design Low Version 1 Risk Accepted

ISSUEIDPREFIX-004

The `burn()` function in the pair contract first reduces the total supply and the user's balance, then transfers the underlying tokens, and last reduces the stored reserves. If the transferred token is a reentrant token, a state inconsistency between the supply and the stored reserves is created.

Note that any computation based on the current supply and the underlying reserves may return wrong results.

Risk accepted:

USDFI has accepted the risk to keep their implementation closer to Uniswap V2.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	4

- Contracts Do Not Extend the Interfaces Code Corrected
- DOS on Gauges When Derived Supply Is 0 Code Corrected
- ERC-2612 Violations Code Corrected
- Lack of Testing Code Corrected

Low-Severity Findings	7

- Different Library Versions Code Corrected
- Fees Claimable After 50 Weeks Specification Changed
- Initial Referral Fee Code Corrected
- Lack of Events Code Corrected
- Maximum Referral Fee Code Corrected
- Used Weights Are Not Reset Code Corrected
- recoverERC20() Allows Recovering Arbitrary Tokens Code Corrected

6.1 Contracts Do Not Extend the Interfaces

Design Medium Version 1 Code Corrected

ISSUEIDPREFIX-012

Most of the contracts interact with each other based on the interface definitions. For example, `GaugeFactory` relies on `IBribe` to handle the calls. However the `Bribe` contract itself does not explicitly implement the `IBribe` interface (`addReward` is missing).

The following is an incomplete list of further examples:

- `BaseV1Pair` does not implement `IBasePair`
- `BaseV1Factory` does not implement `IBaseV1Factory`
- `GaugeFactory` does not implement `IGaugeFactory`
- Similarly, this is true for other contracts.

Without typing, there are no compile-time guarantees that the contract will be compatible with the calls to the functions that the interface defines. This can lead to potential runtime errors and exceptions that are hard to debug. It is important to explicitly define that the contracts implement the corresponding interfaces, to minimize such errors.

Code corrected:

The contracts are implementing now the corresponding interfaces.

6.2 DOS on Gauges When Derived Supply Is 0

Design Medium Version 1 Code Corrected

ISSUEIDPREFIX-008

The subtraction

```
(rewardPerToken() - userRewardPerTokenPaid[account])
```

in `earned()` implies that the `rewardPerToken()` function should be an increasing function. Otherwise, the subtraction will revert. However, its value could decrease when the derived supply goes to zero.

Consider the following scenario:

1. Assume a user for whom `userRewardPerTokenPaid[userA] > 0` holds. The derived supply is 0. The reward per token is 0.
2. A deposit to address 0 is made by an attacker with 1 wei. `rewardPerTokenStored` is set to 0.
3. User A wants to deposit again. The subtraction above reverts.

Note that a scenario for step 2 could occur when user A gets poked first so that his weight changes to 0 (and the total weight too). Then he could get kicked as his derived balance is 0. Ultimately, he will not be able to receive any rewards anymore.

Ultimately, user A will not be able to perform any actions on the gauge anymore.

Code corrected:

Code returns `0` if `derivedSupply == 0`. Hence, the aforementioned scenario cannot revert `earned()`.

6.3 ERC-2612 Violations

Correctness Medium Version 1 Code Corrected

ISSUEIDPREFIX-017

The `permit()` functionality is defined in ERC-2612 which is based on ERC-712. Note that there are two violations of the standards:

1. The `PERMIT_TYPEHASH` violates EIP-712 which describes the typehash to be the keccak256 of the encoded struct. However, its value does not match the hash.
2. The lack of an external `DOMAIN_SEPARATOR()` function violates the ERC-2612 standard.

Ultimately, the implemented standard is violated.

Code corrected:

The code has been corrected.

6.4 Lack of Testing

Design Medium Version 1 Code Corrected

ISSUEIDPREFIX-014

The codebase does not include any tests. Note that it is highly recommended to properly test intended and unintended behaviour with unit and end-to-end tests. These tests help can build an understanding of undocumented functionality.

Code corrected:

USDFI has implemented a testing infrastructure using Hardhat. Please note that tests are considered out-of-scope. Hence, their correctness and/or coverage are not reviewed.

6.5 Different Library Versions

Design Low Version 1 Code Corrected

ISSUEIDPREFIX-011

The libraries have different versions. For example, the `Audit/gauge_factory/Address.sol` and `Audit/bribe_factory/Address.sol` files have different versions.

Code corrected:

The issue has been addressed.

6.6 Fees Claimable After 50 Weeks

Correctness Low Version 1 Specification Changed

ISSUEIDPREFIX-018

The documentation specifies that fees are not claimable after 50 weeks. However, the code implements this differently.

Consider, the scenario where `userTimestamp` is 100 weeks ago. Then, the iteration in `earned()` will start at the `userTimestamp` and iterate for 50 weeks. Afterwards, the `userTimestamp` is updated to the current epoch. Hence, it forfeits the new rewards.

Specification changed:

USDFI has decided to change his documentation, claiming that:

> *We believe that it's not logical to assume that a user actively votes in the protocol’s governance, but does not claim his rewards for 50 consecutive weeks which are visibly on display every time he votes using the frontend.*

Note that during the fix window, we have not received any formal documentation (rather than code comments) from the USDFI to validate this specification has been changed.

6.7 Initial Referral Fee

Correctness Low Version 1 Code Corrected

ISSUEIDPREFIX-016

The documentation specifies that the default base referral fee shall be 2%. However, it is 0%.

Code Corrected:

USDFI has correctly hardcoded `baseReferralFee` to 2000.

6.8 Lack of Events

Design Low Version 1 Code Corrected

ISSUEIDPREFIX-015

Events are not emitted always emitted on important state-modifying actions. Note that this is the case across all contracts. The following is an incomplete list of functions that lack event emissions:

- `BaseV1Factory.setBaseStableFee()`
- `BaseV1Factory.setBaseVariableFee()`
- `BaseV1Factory.setShouldGasThrottleAndMaxGasPrice()`
- `BaseV1Factory.setOwner()`
- `BaseV1Factory.acceptOwner()`
- `BaseV1Factory.setPause()`
- `BaseV1Factory.setProtocolAddress()`
- `BaseV1Factory.setAdmins()`
- `BaseV1Factory.setPause()`
- `BaseV1Factory.setPause()`
- `BaseV1Factory.setPause()`
- `BaseV1Pair.setFee()`
- `GaugeFactory.preDistribute()`
- `GaugeFactory.updateVeProxy()`
- `GaugeFactory.updatePokeDelay()`
- `GaugeFactory.updateMaxVotesToken()`
- `GaugeFactory.updateReferrals()`
- `ProtocolGovernance.setGovernance()`
- `ProtocolGovernance.acceptGovernance()`
- `ProtocolGovernance.setAdminAndVoter()`
- `ProtocolGovernance.setStableMiner()`
- `ProtocolGovernance.updateBaseReferrals()`
- `Gauge.kick()`
- `Gauge.updateReferral()`
- `Gauge.setWhitelisted()`

- `BribeFactory.createBribe()`
- `Bribe.addRewardtoken()`
- `Bribe.setWhitelisted()`
- `Bribe.updateReferral()`

Emitting events could ease following the state of the contract.

Code corrected:

USDFI has added emitting relevant events to the functions of the above list.

6.9 Maximum Referral Fee

Correctness | Low | Version 1 | Code Corrected

ISSUEIDPREFIX-007

The documentation specifies that the referral fee for gauges and bribes is at most 10%. However, that is not enforced, and; hence, the referral fee may exceed 10%.

Code corrected:

USDFI has corrected the code by enforcing this limit:

```
require((_baseReferralFee <= 10000), "must be lower 10%");
```

6.10 Used Weights Are Not Reset

Correctness | Low | Version 1 | Code Corrected

ISSUEIDPREFIX-013

When weights are reset, `usedWeight` is not set to 0. While this has no impact on voting or execution, the automatic getter for `usedWeight` may return outdated values.

Code corrected:

USDFI has successfully corrected the code by setting `usedWeights[_owner]` to zero.

6.11 `recoverERC20()` Allows Recovering Arbitrary Tokens

Correctness | Low | Version 1 | Code Corrected

ISSUEIDPREFIX-020

The `recoverERC20()` function should according to the documentation allow the governance to withdraw non-reward tokens from the bribe contract. However, reward tokens can be withdrawn, too.

Code corrected:

USDFI correctly has changed to code to check that the token to be withdrawn is not a reward token.

6.12 Comments With Errors

Informational Version 1 Code Corrected

ISSUEIDPREFIX-009

The code contains some comments with errors. Examples are:

1. The comment for the `stable` parameter of `BaseV1Pair` specifies that is not immutable, while it is.
2. The comment for the `baseStableFee` of `BaseV1Factory` mentions that it is 0.04%. However, it is 0.05%.

Code corrected:

USDFI has successfully resolved both the aforementioned errors.

6.13 Gas Optimisation

Informational Version 1 Code Corrected

ISSUEIDPREFIX-019

The codebase has several inefficiencies in terms of gas costs when deploying and executing smart contracts. Here, we report a list of non-exhaustive possible gas optimisations:

1. `Bribe.constructor`: To deploy a bribe contract, in order to set `firstBribeTimestamp`, `referralContract`, and `referralFee`, multiple queries to the `gaugeFactory` are made. However, the memory variable `_gaugeFactory` has the same address and using it instead of `gaugeFactory` reduces number of storage reads.
2. `getRewardForOwnerToOtherOwnerSingleToken` of `Bribe` has a visibility of public, with input parameter in the memory. As this function is only called externally, its visibility can be changed to external and consequently the input parameters to calldata.
3. The visibility of `updateReferral` in `Bribe` can be changed to external and its input parameters to calldata.
4. `BribeFactory` defines a storage variable named `last_bribe`, which is set but never read. Apart from that, the constructor returns this storage variable, although a local memory variable `lastBribe` holds the same address and returning it is more gas efficient.
5. `Gauge` defines two state variables `token` and `TOKEN`, which necessarily hold the same address once set. Following the code path, they never get out of sync and are always equal. Hence, holding just one of them and cast it whenever neccesary.
6. `gaugeFactory` in `Gauge` is never modified after being set in the constructor. Hence, it can be defined as immutable. Apart from that, they way it is set in the current implementation, it holds the same value as `DISTRIBUTION`.
7. `Gauge._claimVotingFees` calculates `bribe` even in the case, where neither `claimed0` nor `claimed1` is non-zero. Calculating `bribe` inside the if-statement makes it more gas efficient.
8. `Gauge._deposit` defines a local memory variable `userAmount`, which holds the value of input parameter `amount`. It seems to be unnecessary.

9. `GaugeFactory._vote` updates `totalWeight` incementally in each iteration of the loop. By accumulating all the changes at updating `totalWeight` after the last iteration, gas consumption can be reduced significantly.
10. `GaugeFactory.addGauge` writes to the last element of `maxVotesToken`. However, this last element has the same value as the input parameter `_tokenLP`.
11. `GaugeFactory.preDistribute` inside the loop, makes multiple accesses to the storage variable `lockedWeights[_tokens[i]]`. Doing the intermediate calculations in the memory and writing them back to storage is more gas efficient.
12. `GaugeFactory.updateReferrals` can be defined as external along its input parameters as calldata.
13. `GaugeFactory.delay` can be defined as constant, as its value never changes later.
14. `GaugeFactory.STABLE` after being assigned a value in the constructor never gets modified. Hence, it can be defined as immutable.
15. `BaseV1Pair.permit` recalculates `DOMAIN_SEPARATOR` for every call. It makes sense as a prevention mechanism against forks. The storage write could potentially be done only if the chain id changed.
16. `BaseV1Pair` defines an event named `Claim`. It is called only once with `sender` and `recipient` holding the same address.
17. `BaseV1Factory.constructor` sets the storage variable `isPaused` to false, which is the default value for any boolean values.
18. `BaseV1Factory` defines three storage variables `_temp0`, `_temp1`, and `_temp`. These variables are used solely as input parameters when deploying a `BaseV1Pair`. Instead of occupying extra storage for these variables, they can easily be defined as input parameters to the `BaseV1Pair`.
19. The modifier `BaseV1Pair.gasThrottle` can be implemented more optimised, by assigning `maxGasPrice == 0` as should not throttle, which consequently removes the need to define state variable `shouldGasThrottel` in `BaseV1Factory`.

Code corrected:

USDFI has correctly addresses most of the aforementioned gas inefficiencies. The following inefficiencies are going to be reviewed later by USDFI:

1. `GaugeFactory.divisor` can be defined as constant. Its value is not later set during deployment through constructor.
2. `BaseV1Factory` defines three storage variables `_temp0`, `_temp1`, and `_temp`. These variables are used solely as input parameters when deploying a `BaseV1Pair`. Instead of occupying extra storage for these variables, they can easily be defined as input parameters to the `BaseV1Pair`.

6.14 No NatSpec

Informational **Version 1** **Code Corrected**

ISSUEIDPREFIX-021

While documentation was provided, the individual functions are not documented in the code. It is highly recommended to at least describe each entry point with descriptive comments (e.g. NatSpec for all functions).

Code corrected:

Comments have been added. However, no NatSpec was used.

6.15 Referrals Default Values

Version 1 Code Corrected

ISSUEIDPREFIX-010

The `referralContract` by default is the 0-address, indicating that this feature is deactivated. However, the code will revert if governance does not have a valid referral contract.

Further, if there is no `mainRefFeeReceiver` specified, the 0-address receives funds.

Code corrected:

Initial values need to be provided on construction now.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Magic Values

Informational **Version 1** **Code Partially Corrected**

ISSUEIDPREFIX-005

The use of magic numbers in the codebase is not recommended, they should be replaced by variables with a self-explanatory name. Examples are:

- The scaling factor `1e18`
- `1000`
- `10000`
- `100000`
- `50` (number of maximum iterations in `Bribe.earned`)

Code partially corrected:

Some of the numbers have been replaced by variables with a self-explanatory name.

7.2 Voting Twice per Epoch Possible

Informational **Version 1** **Acknowledged**

ISSUEIDPREFIX-006

`poke()` can be considered as a voting function. However, it ignores `lastVote`. Hence, it is technically possible to revote with the same allocations.

Acknowledged:

USDFI has acknowledged this issue stating that:

> *This is by design since poke may be executed voluntarily at any time. Please note that re-poking and re-voting without acquiring more veTokens is only to the detriment of the user himself as the passage of time always reduces his voting power (veTokens) and is also costly*

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 DOS Possibilities

Note Version 1

If governance adds to many gauges, `preDistribute()` may hit the gas limit. Additionally, users could vote for too many gauges so that they DOS themselves.

8.2 Deprecated Gauges May Have Locked Rewards

Note Version 1

The function `preDistribute()` considers the weight of the deprecated gauges for the total amount of votes. However, the `distribute()` function will never distribute such rewards. Hence, as long as there are any deprecated gauges with remaining votes, tokens will be locked in the contract. It is worth mentioning that this is required since, otherwise, the distribution could be DOSed, if a gauge was resurrected between predistribution and distribution.

8.3 Low Default Fee Value for LPs

Note Version 1

LPs should be aware that by default only 20% percent of the accrued fees will be allocated to them.

8.4 Reward Mechanics

Note Version 1

An LP with high liquidity may provide liquidity to gauges, when he sees a high rate. Other users, who bribed the voters to receive rewards, may end up getting less profit compared to just receiving fees.

8.5 Sandwiching

Note Version 1

Users should be aware that sandwiching swaps is possible (similar to Uniswap V2 like pools). As described in System Overview, users should use helper contracts that allow specifying slippage protection so that sandwich attacks are limited to some degree.

8.6 What Should Happen With Voteless Bribes?

Note | Version 1

If a bribe is notified about a reward, it stores the reward for the next epoch. However, if no voter voted for the allocations, and hence did not receive bribe shares, no one will be able to claim the bribes. Note that it is assumed that at least a voter will cast a vote to claim the rewards. However, gas fees could be higher than the reward's value.

8.7 `distribute()` Without `preDistribute()`

Note | Version 1

When a gauge is added, `hasDistributed` for the LP token will be `false`. Hence, it is technically possible to distribute rewards without `preDistribute()`, although its reward will be zero.

8.8 `gasThrottle`

Note | Version 1

The `gasThrottle` modifier is applied to the `swap()` function to reduce the impact of front-running by specifying a maximum gas price an action should be able to have. However, such a mechanism brings certain risks. The following is an incomplete list of examples:

1. Gas price changes naturally and exceeds the maximum. Swaps can be broken temporarily until the maximum price is updated.
2. It is still be possible to front-run users with mints so that the price is affected. Consider the example, in which only one LP exists. Minting does not yield a loss for the user in that scenario but can change the price significantly.
3. Some liquidators may want to exchange the liquidated funds against the repayment currency so that the profit in the repayment currency is guaranteed. Since liquidations are time-sensitive, higher gas prices are set. `gasThrottle` could make it undesirable for liquidators to use the exchange in such scenarios.

Further, note that MEV often is done by transferring ETH to the block creator directly. Hence, in such cases the mechanism is ineffective.

8.9 `refLevelPercent` Is Ended With 0 Elements

Note | Version 1

Users should be aware that a zero element in `refLevelPercent` will break the fee distribution for the subsequent elements.

8.10 `rewardsPerToken()` Can Return `rewardsPerEpoch`

Note | Version 1

The function `rewardsPerToken()` returns the ratio of the rewards and the supply at a given epoch number. However, if the total supply is zero, `rewardsPerEpoch` is returned.