
# Security Audit Report for NOAH-DAO

**Date:** Jul 10, 2023

**Version:** 1.0

**Contact**: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.07.10
16:52:24 +08'00'

# Contents

## Report Manifest

| Item | Description |
| --- | --- |
| Client | NOAH |
| Target | NOAH-DAO |

## Version History

| Version | Date | Description |
| --- | --- | --- |
| 1.0 | July 10, 2023 | First Version |

**About BlockSec** The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

# Chapter 1 Introduction

## 1.1 About Target Contracts

| Information | Description |
|---|---|
| Type | Smart Contract |
| Language | Solidity |
| Approach | Semi-automatic and manual verification |

The repository that has been audited includes noah-dap-contacts.zip.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The MD5 values of the files during the audit are shown in the following. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

**Version 1**

| File | md5 |
|---|---|
| BaseGauge.sol | eaead534557d25bf6ba7a95b80cc42f3 |
| Bribe.sol | 4151633719b32fbe92a1a5feeb8f8b37 |
| FeeVault.sol | d38fcee1dc997569e44759a8f1c55aa7 |
| EsNoah.sol | b65dc788ba9d620bec3827ee28c66cea |
| Gauge.sol | fb11a8c138272fb22e2a99f3c5785e87 |
| Minter.sol | 9baee1acd31ccd03550fe749363a6817 |
| Noah.sol | 7c182db38b27c1d698081a4b24d749dc |
| RewardsDistributor.sol | e58e5bdcdbad246f8fd7522f4d999f7a |
| SmartWalletChecker.sol | f605b646835a417981c46d8a0ff56b7f |
| Vester.sol | a52f445260a58dbecb455580ccfebf81 |
| Voter.sol | 26a31c98bf72105de84aa8a94f094222 |
| VotingEscrow.sol | 4599ae7f4f9841682e439a1a5b3c9ac2 |
| Governable.sol | fda68347f3c7603d4b8a233becc1c147 |
| BribeFactory.sol | 8c7fe6af0b58c865ec17c4d35fc0107f |
| GaugeFactory.sol | 6bb83e2490a0984f14aa6ef2a5abebe9 |
| IBribe.sol | 4a54ff55155f57b5f170a96ee0aa6d76 |
| IBribeFactory.sol | 56172e2cff78fc78ab52200f286587f2 |
| IEsNoah.sol | 61b6058048254589462e90995699b39d |
| IFeeVault.sol | 189774af2210c3bccc7ea83f8e29e916 |
| IGauge.sol | 9bc68d9f50ce66c5c3ab129dba06b494 |
| IGaugeFactory.sol | 021fb49c3957ef2e51126701abee7808 |
| IMinter.sol | c7b17a1f31fa0852619b1615f108c407 |
| INoah.sol | f46c3fcc01301f60b880474724dad25e |
| IPairFactory.sol | cccb14941fff68930545f58983d785d6 |
| IPairInfo.sol | deeb1114b0b61049dc805ce4e8673329 |

| | |
|---|---|
| IRewardsDistributor.sol | 4399b98b6650597b52a8380f7ed9f967 |
| ISmartWalletChecker.sol | 5bb7218c380c77b5e958ac6333ddcb75 |
| IVester.sol | 354443365d2b31acb8d3e1e4123aa703 |
| IVoter.sol | 76cba4b77f2caf2772a4189c44702eec |
| IVotingEscrow.sol | 1bae373a2055f6ce58237164deaf8c95 |
| IVotingEscrowCallback.sol | 5feecd2601929d72abc1f32d36319e84 |
| Math.sol | 28f2a9eb3b3403f5963b85f279c553c1 |

### Version 2

| File | md5 |
|---|---|
| Bribe.sol | b7cea8854529473a962007e68c3223a8 |
| FeeVault.sol | e9a66205d1a0ea3aa7f3abaf701e3aa2 |
| EsNoah.sol | 10dbe0109884d078f0a797bda0be8a72 |
| Gauge.sol | 435633fdb5d3fb2b4c50ca4514a15bc6 |
| Minter.sol | 49c8ee5f2bb9c109ae68a7aff4848158 |
| Noah.sol | d13cbd610049962c841e62c54f083270 |
| RewardsDistributor.sol | fa8c9a641f2a60a09779b10b9b63d885 |
| SmartWalletChecker.sol | c4d911993029f15ba6df051ff8b3b401 |
| Vester.sol | 3c120a2487d4d87afd86fe03091b4581 |
| Voter.sol | 81d107c61f6d1fb1896b9f65ac8dbcce |
| VotingEscrow.sol | b0b1c3abe6e4a3de22f6ff0dc4aba6b4 |
| Governable.sol | a25efd519bd9d030764d5f0b388e3e40 |
| BribeFactory.sol | 8e60c92ff0804f0dd7508a84df761f11 |
| GaugeFactory.sol | b2d4cb930b5538e1e070846c27136610 |
| IBribe.sol | a08f093027f02fb302fabc54d1d6ff8c |
| IBribeFactory.sol | 435da33a99d14cfa757ab7b35734ed57 |
| IEsNoah.sol | 86e21fa4b3f45b8727cf7cc847f48a61 |
| IFeeVault.sol | 94b42438bd0535f54a8ddc215b139b01 |
| IGauge.sol | ff713c71c20b4e2ac3bf14bff7a7678a |
| IGaugeFactory.sol | ef1d4d06ee7ba37c514944deb07cb03d |
| IMinter.sol | e69b46e14f731f0dc7ef2d93e8d4d805 |
| INoah.sol | 426df85b079bbe9d0d6fe5a8122aa226 |
| IPairFactory.sol | a391ffa04ae839c1d21625c2f39853a2 |
| IPairInfo.sol | f4a4bd3a5d96c05678a7b76e3d156ac5 |
| IRewardsDistributor.sol | 3168b1f567643e11bbd2e236c9930f64 |
| ISmartWalletChecker.sol | 92a322cda42a60c708c10c7827762472 |
| IVester.sol | 958de38a88ffbf3e6ed409e46ba03b58 |
| IVoter.sol | 94581fa63dc9807efb385cc2c6103b54 |
| IVotingEscrow.sol | 21f6c40e53c18f08eb618b6c1dc00ae2 |
| IVotingEscrowCallback.sol | d67bb583e8a5ae4ba894273e2045ab32 |
| Math.sol | faddfc801d4c366d01595cbd368ecca9 |

### Version 3

| File | md5 |
|---|---|
| Bribe.sol | 3d1ca22669e82a3b70a569c18f53fe99 |
| FeeVault.sol | f00d33ebab1aa920c4d88916c5554cc7 |
| EsNoah.sol | b65dc788ba9d620bec3827ee28c66cea |
| Gauge.sol | 1e03b35a37d02c4e53b3709f504dcb6f |
| Minter.sol | df7a9e9525e3671f3c0c07f23f7f3f86 |
| Noah.sol | 7c182db38b27c1d698081a4b24d749dc |
| RewardsDistributor.sol | 9ef07ce5fbf7a3270a4401de92f346b4 |
| SmartWalletChecker.sol | 85fb9a79ecf2b8237656a4fedbe8020b |
| Vester.sol | 4e680a1d281f3aa64bed73efb15f7568 |
| Voter.sol | a42c33ab6eaf683a80f3b3a4995aa83e |
| VotingEscrow.sol | fc42454dd7d81a6fef5a6409410f733e |
| Governable.sol | ed8a6447422edb04a369af39d9a17364 |
| BribeFactory.sol | 0510b25dd1c0a463625dfb41c93155f8 |
| GaugeFactory.sol | ffa396aba19188cb4267cc531e4c9251 |
| IBribe.sol | 8962a247076481df0d9f7a574d95154d |
| IBribeFactory.sol | ab816dd15588c7b60cddcf7270bdabce |
| IEsNoah.sol | 61b6058048254589462e90995699b39d |
| IFeeVault.sol | e371829da6bdcce25638fbde7f82c158 |
| IGauge.sol | f7e927ed4e56b8d77054a6afe3ee74a7 |
| IGaugeFactory.sol | 021fb49c3957ef2e51126701abee7808 |
| IMinter.sol | ab988381bb64f248318f7f358c4eeb52 |
| INoah.sol | f46c3fcc01301f60b880474724dad25e |
| IPairFactory.sol | cccb14941fff68930545f58983d785d6 |
| IPairInfo.sol | deeb1114b0b61049dc805ce4e8673329 |
| IRewardsDistributor.sol | 4399b98b6650597b52a8380f7ed9f967 |
| ISmartWalletChecker.sol | 5bb7218c380c77b5e958ac6333ddcb75 |
| IVester.sol | b18d9c5acfb5e73719ec28e3e4fcd451 |
| IVoter.sol | d33a78f54f3ae1fc3f8468b5cb7279b8 |
| IVotingEscrow.sol | f3d6f952282a4fda942010acf166fbe0 |
| IVotingEscrowCallback.sol | 5feecd2601929d72abc1f32d36319e84 |
| Math.sol | bafeb8c445ef4482e47865ac54aaf881 |

### Version 4

| File | md5 |
|---|---|
| Bribe.sol | a871ec26f04f9c916abe058647d79458 |
| FeeVault.sol | 5ebe1655ff1bcc3e0922704474e1a65d |
| EsNoah.sol | b65dc788ba9d620bec3827ee28c66cea |
| Gauge.sol | 1e03b35a37d02c4e53b3709f504dcb6f |
| Minter.sol | df7a9e9525e3671f3c0c07f23f7f3f86 |

| | |
|---|---|
| Noah.sol | 7c182db38b27c1d698081a4b24d749dc |
| RewardsDistributor.sol | 9ef07ce5fbf7a3270a4401de92f346b4 |
| SmartWalletChecker.sol | 85fb9a79ecf2b8237656a4fedbe8020b |
| Vester.sol | 9646ee240408c00c4a6d8ae58a86664e |
| Voter.sol | 4fc8bb4ffdbb578b20be52a06bfca106 |
| VotingEscrow.sol | fc42454dd7d81a6fef5a6409410f733e |
| Governable.sol | ed8a6447422edb04a369af39d9a17364 |
| BribeFactory.sol | 0510b25dd1c0a463625dfb41c93155f8 |
| GaugeFactory.sol | ffa396aba19188cb4267cc531e4c9251 |
| IBribe.sol | 9c0d9f4ea8876ad3b3c3d3fb4651df75 |
| IBribeFactory.sol | ab816dd15588c7b60cddcf7270bdabce |
| IEsNoah.sol | 61b6058048254589462e90995699b39d |
| IFeeVault.sol | e371829da6bdcce25638fbde7f82c158 |
| IGauge.sol | f7e927ed4e56b8d77054a6afe3ee74a7 |
| IGaugeFactory.sol | 021fb49c3957ef2e51126701abee7808 |
| IMinter.sol | ab988381bb64f248318f7f358c4eeb52 |
| INoah.sol | f46c3fcc01301f60b880474724dad25e |
| IPairFactory.sol | cccb14941fff68930545f58983d785d6 |
| IPairInfo.sol | deeb1114b0b61049dc805ce4e8673329 |
| IRewardsDistributor.sol | 4399b98b6650597b52a8380f7ed9f967 |
| ISmartWalletChecker.sol | 5bb7218c380c77b5e958ac6333ddcb75 |
| IVester.sol | b18d9c5acfb5e73719ec28e3e4fcd451 |
| IVoter.sol | d33a78f54f3ae1fc3f8468b5cb7279b8 |
| IVotingEscrow.sol | f3d6f952282a4fda942010acf166fbe0 |
| IVotingEscrowCallback.sol | 5feecd2601929d72abc1f32d36319e84 |
| Math.sol | bafeb8c445ef4482e47865ac54aaf881 |

## 1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

## 1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

### 1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

### 1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

### 1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

### 1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

**Note** *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

## 1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.5.

**Table 1.5:** Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Low* |
|---|---|---|
| *High* | High | Medium |
| *Low* | Medium | Low |

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

# Chapter 2 Findings

In total, we find **Twenty-one** potential issues. Besides, we have **seven** recommendations and **six** notes as follows:

- High Risk: 7
- Medium Risk: 11
- Low Risk: 3
- Recommendations: 7
- Notes: 6

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | High | Index out of Bounds for the Empty Array | Software Security | Fixed |
| 2 | Medium | Improper Use of the Keyword Memory | Software Security | Fixed |
| 3 | Low | Incorrect Index in getPriorSupplyIndex | Software Security | Fixed |
| 4 | Medium | Potential Loop from Self-Calling | Software Security | Fixed |
| 5 | Low | Incorrect Validation of Withdrawal Rate | Software Security | Fixed |
| 6 | High | Miscalculated Bribe Rewards (I) | DeFi Security | Fixed |
| 7 | High | Miscalculated Bribe Rewards (II) | DeFi Security | Fixed |
| 8 | Medium | Timely invocation of update_period() before setReleaseFactor() and setPledgeFactor() | DeFi Security | Acknowledged |
| 9 | Medium | Timely invocation of distribute() in notifyRewardAmount() | DeFi Security | Confirmed |
| 10 | Medium | Reward for Killed Gauge Being Locked | DeFi Security | Confirmed |
| 11 | Medium | Lack of Checks for Gauges that Do Not Support Voting | DeFi Security | Confirmed |
| 12 | Medium | Reward Token can be Managed by Users with Different Privileges | DeFi Security | Fixed |
| 13 | Medium | Timely invocation of claimfees() in Gauge | DeFi Security | Acknowledged |
| 14 | High | Failed to Notify Rewards due to the Reentrancy Lock | DeFi Security | Fixed |
| 15 | High | Swap Fee Rewards cannotDistribution Mechanism does not Work | DeFi Security | Fixed |
| 16 | High | Manipulated Unlocking Duration | DeFi Security | Fixed |
| 17 | Medium | Risk of Voting Power Manipulation when is_unlock is True | DeFi Security | Acknowledged |
| 18 | Medium | Lack of Check of Function withdrawToken | DeFi Security | Fixed |
| 19 | Low | Inconsistent Status Update during Voting Process | DeFi Security | Fixed |
| 20 | Medium | Miscalculated poolWeight with Duplicated Pool Voting | DeFi Security | Fixed |
| 21 | High | Incorrect Reward Calculations from Inappropriate Check | DeFi Security | Fixed |
| 22 | - | Lack of Zero Address Check | Recommendation | Confirmed |
| 23 | - | Redundant Functions | Recommendation | Fixed |
| 24 | - | Redundant Invocation of Function _updateFor | Recommendation | Fixed |
| 25 | - | Meaningless Usage of max | Recommendation | Fixed |
| 26 | - | Inappropriate Variable Naming | Recommendation | Confirmed |
| 27 | - | Lack of Check for releaseFactor and pledgeFactor | Recommendation | Confirmed |
| 28 | - | Redundant Check in Function mint_marketing | Recommendation | Fixed |
| 29 | - | Potential Centralization Problem | Note | Confirmed |
| 30 | - | Timely deployment contracts | Note | Confirmed |
| 31 | - | Non-Linear Unlocking in Multiple Claims | Note | Confirmed |
| 32 | - | Token Release for Team and VC without Time Restrictions | Note | Confirmed |
| 33 | - | Potential Inequity Function poke() of the Contract Voter | Note | Confirmed |
| 34 | - | Incompatible Tokens | Note | Confirmed |

The details are provided in the following sections.

## 2.1 Software Security

### 2.1.1 Index out of Bounds for the Empty Array

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the contract `Gauge`, the array `fees` are initialized to an empty array in the constructor. The function `_claimFees()` caches the global fees into the empty array `fees[0]` and loads the last value from it directly, which results in a revert due to an index-out-of-bounds error.

```
18    constructor(
19        address _stake,
20        address _bribe,
21        address _ve,
22        address _voter,
23        bool _forPair,
24        address[] memory _allowedRewardTokens
25    ) BaseGauge(_stake, _bribe, _ve, _voter, _forPair, _allowedRewardTokens) {
26        fees = new uint[](0);
27    }
```

**Listing 2.1:** Gauge.sol

**Impact** Fees can not be claimed as the invocation of function `notifyRewardAmount()` will revert by the index-out-of-bounds error.

**Suggestion** Revise the length of the array accordingly.

### 2.1.2 Improper Use of the Keyword Memory

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In Solidity, assignments made from memory to memory only create references. This means that changing the value of one memory pointer will also update any other references to that same memory location. In the function `_checkpoint` of the contract `VotingEscrow`, a memory pointer `initial_last_point` is created as a reference to the variable `last_point`. The `initial_last_point` variable is intended to be used as a bias to calculate the block number of subsequent checkpoints. However, due to the memory reference problem, the result of `last_point.blk` is incorrect. Specifically, the value of `initial_last_point.ts` is modified to `t_i` when `last_point.ts` is assigned as `t_i` in the loop. As a result, the assignment `last_point.blk = initial_last_point.blk + (block_slope * (t_i - initial_last_point.ts) / MULTIPLIER)` is equivalent to `last_point.blk = initial_last_point.blk`. This causes the value of each `checkpoint` to be the same as the value of the first one.

```
175     function _checkpoint(address _account, LockedBalance memory old_locked, LockedBalance memory
            new_locked) internal {
176     Point memory u_old;
177     Point memory u_new;
178     int128 old_dslope = 0;
179     int128 new_dslope = 0;
180     uint _epoch = epoch;
181
182
183     if (_account != address(0)) {
184         // Calculate slopes and biases
185         // Kept at zero when they have to
186         if (old_locked.end > block.timestamp && old_locked.amount > 0) {
187             u_old.slope = old_locked.amount / iMAXTIME;
188             u_old.bias = u_old.slope * int128(int256(old_locked.end - block.timestamp));
189         }
190         if (new_locked.end > block.timestamp && new_locked.amount > 0) {
191             u_new.slope = new_locked.amount / iMAXTIME;
192             u_new.bias = u_new.slope * int128(int256(new_locked.end - block.timestamp));
193         }
194
195
196         // Read values of scheduled changes in the slope
197         // old_locked.end can be in the past and in the future
198         // new_locked.end can ONLY by in the FUTURE unless everything expired: than zeros
199         old_dslope = slope_changes[old_locked.end];
200         if (new_locked.end != 0) {
201             if (new_locked.end == old_locked.end) {
202                 new_dslope = old_dslope;
203             } else {
204                 new_dslope = slope_changes[new_locked.end];
205             }
206         }
207     }
208
209
210     Point memory last_point = Point({bias: 0, slope: 0, ts: block.timestamp, blk: block.number});
211     if (_epoch > 0) {
212         last_point = point_history[_epoch];
213     }
214     uint last_checkpoint = last_point.ts;
215     // initial_last_point is used for extrapolation to calculate block number
216     // (approximately, for *At methods) and save them
217     // as we cannot figure that out exactly from inside the contract
218     Point memory initial_last_point = last_point;
219     uint block_slope = 0; // dblock/dt
220     if (block.timestamp > last_point.ts) {
221         block_slope = (MULTIPLIER * (block.number - last_point.blk)) / (block.timestamp -
                last_point.ts);
222     }
223     // If last point is already recorded in this block, slope=0
224     // But that's ok b/c we know the block in such case
```

```
225
226
227     // Go over weeks to fill history and calculate what the current point is
228     {
229         uint t_i = (last_checkpoint / WEEK) * WEEK;
230         for (uint i; i < 255; ++i) {
231             // Hopefully it won't happen that this won't get used in 5 years!
232             // If it does, users will be able to withdraw but vote weight will be broken
233             t_i += WEEK;
234             int128 d_slope = 0;
235             if (t_i > block.timestamp) {
236                 t_i = block.timestamp;
237             } else {
238                 d_slope = slope_changes[t_i];
239             }
240             last_point.bias -= last_point.slope * int128(int256(t_i - last_checkpoint));
241             last_point.slope += d_slope;
242             if (last_point.bias < 0) {
243                 // This can happen
244                 last_point.bias = 0;
245             }
246             if (last_point.slope < 0) {
247                 // This cannot happen - just in case
248                 last_point.slope = 0;
249             }
250             last_checkpoint = t_i;
251             last_point.ts = t_i;
252             last_point.blk = initial_last_point.blk + (block_slope * (t_i - initial_last_point.ts))
                      / MULTIPLIER;
253             _epoch += 1;
254             if (t_i == block.timestamp) {
255                 last_point.blk = block.number;
256                 break;
257             } else {
258                 point_history[_epoch] = last_point;
259             }
260         }
261     }
262
263
264     ...
265 }
```

**Listing 2.2:** VotingEscrow.sol

**Impact** Some functions that rely on the block number of the `point_history` may get unexpected results, such as the function `balanceOfAtB()`.

**Suggestion** Use deep copy for `initial_last_point` assignment.

### 2.1.3 Incorrect Index in getPriorSupplyIndex

**Severity** Low

**Status** Fixed in `Version 3`

**Introduced by** `Version 2`

**Description** In the function `getPriorSupplyIndex()` of the contract Bribe, the check statement `if (supplyCheckpoint > timestamp)` is incorrect. In the current implementation, the start index for the array `supplyCheckpoints[]` begins from 1, so the check for the point at the index 0 is meaningless.

```
126    function getPriorSupplyIndex(uint timestamp) public view returns (uint) {
127    uint nCheckpoints = supplyNumCheckpoints;
128    if (nCheckpoints == 0) {
129        return 0;
130    }
131
132
133    // First check most recent balance
134    if (supplyCheckpoints[nCheckpoints].timestamp <= timestamp) {
135        return (nCheckpoints);
136    }
137
138
139    // Next check implicit zero balance
140    if (supplyCheckpoints[0].timestamp > timestamp) {
141        return 0;
142    }
143
144
145    uint lower = 0;
146    uint upper = nCheckpoints;
147    while (upper > lower) {
148        uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
149        SupplyCheckpoint memory cp = supplyCheckpoints[center];
150        if (cp.timestamp == timestamp) {
151            return center;
152        } else if (cp.timestamp < timestamp) {
153            lower = center;
154        } else {
155            upper = center - 1;
156        }
157    }
158    return lower;
159}
```

**Listing 2.3:** Bribe.sol

```
233    function _writeSupplyCheckpoint() internal {
234    uint _nCheckPoints = supplyNumCheckpoints;
235    uint _timestamp = block.timestamp;
236
237
238    if (_nCheckPoints > 0 && supplyCheckpoints[_nCheckPoints].timestamp == _timestamp) {
239        supplyCheckpoints[_nCheckPoints].supply = totalSupply;
240    } else {
241        _nCheckPoints += 1;
```

```
242         supplyCheckpoints[_nCheckPoints] = SupplyCheckpoint(_timestamp, totalSupply);
243         supplyNumCheckpoints = _nCheckPoints;
244     }
245 }
```

**Listing 2.4:** Bribe.sol

**Impact** The check will always be false, which is meaningless.

**Suggestion** Use `supplyCheckpoints[1].timestamp` instead of `supplyCheckpoints[0].timestamp`.

### 2.1.4 Potential Loop from Self-Calling

**Severity** Medium

**Status** Fixed in `Version 3`

**Introduced by** `Version 2`

**Description** In the function `check()` of the contract `SmartWalletChecker`, there's a function call to the checker that implements the function `check()` interface. However, only the contract `SmartWalletChecker` implements the function `check()`. In this case, if the called contract is the contract `SmartWalletChecker` itself, it will result in a self-call loop and a revert due to out of gas.

```
48     function check(address _wallet) external view returns (bool) {
49     if (!isWhitelistEnabled) {
50         return true;
51     }
52
53
54     bool _check = wallets[_wallet];
55     if (_check) {
56         return _check;
57     } else {
58         if (checker != address(0)) {
59             return SmartWalletChecker(checker).check(_wallet);
60         }
61     }
62     return false;
63 }
```

**Listing 2.5:** SmartWalletChecker.sol

**Impact** The invocation of the function `check()` may revert.

**Suggestion** Add a check to prevent the `checker` from being the contract `SmartWalletChecker` itself.

### 2.1.5 Incorrect Validation of Withdrawal Rate

**Severity** Low

**Status** Fixed in `Version 4`

**Introduced by** `Version 3`

**Description** In the function `withdraw()` of the contract `Vester`, the validation of `require(rate > 0 || rate <= PRECISION, "Vester: rate invalid")` is incorrect, it should ensure both conditions are satisfied, or the check will be meaningless.

```
70    function withdraw(uint rate) external nonReentrant {
71        require(rate > 0 || rate <= PRECISION, "Vester: rate invalid");
72        _updateVesting(msg.sender, OperationType.WITHDRAW_TYPE);
73
74
75        uint balance = balances[msg.sender];
76        require(balance > 0, "Vester: vested amount is zero");
77
78
79        uint amount = (balance * rate) / PRECISION;
80        if (amount == balance) {
81            ve.unvesting(msg.sender);
82        }
83
84
85        balances[msg.sender] = balance - amount;
86        totalVesting -= amount;
87
88
89        IERC20(esToken).safeTransfer(msg.sender, amount);
90
91
92        emit Withdraw(msg.sender, amount);
93    }
```

**Listing 2.6:** Vester.sol

**Impact** The check is meaningless.

**Suggestion** Change `rate > 0 || rate <= PRECISION` to `rate > 0 && rate <= PRECISION`.

## 2.2 DeFi Security

### 2.2.1 Miscalculated Bribe Rewards (I)

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `earned()` in the contract `Bribe` is used to calculate the user's rewards in the current epoch. It will synchronize rewards starting from the `checkpoint` where the user last claimed the reward, traversing to the latest `checkpoint`. This process is divided into two steps. In the first step, the traversal only goes up to the second-to-last `checkpoint`, while the second step updates only the last `checkpoint`.

However, based on the current implementation, we have observed that the first step always misses the accumulation of rewards for the second-to-last `checkpoint`. Specifically, the calculation of reward quantity is based on the voting amount recorded for the user in the last `checkpoint` of each epoch. Each iteration

calculates the rewards for the corresponding epoch and then adds them to the total rewards the user should receive in the next iteration. This results in the rewards for the epoch corresponding to the second-to-last `checkpoint` being calculated but not included in the rewards available to the user.

```
235     function earned(address token, address account) public view returns (uint) {
236     uint _startTimestamp = lastEarn[token][account];
237     if (numCheckpoints[account] == 0) {
238         return 0;
239     }
240
241
242     uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
243     uint _endIndex = numCheckpoints[account] - 1;
244
245
246     uint reward = 0;
247     // you only earn once per epoch (after it's over)
248     Checkpoint memory prevRewards; // reuse struct to avoid stack too deep
249     prevRewards.timestamp = _bribeStart(_startTimestamp);
250     uint _prevSupply = 1;
251
252
253     if (_endIndex > 0) {
254         for (uint i = _startIndex; i <= _endIndex - 1; i++) {
255             Checkpoint memory cp0 = checkpoints[account][i];
256             uint _nextEpochStart = _bribeStart(cp0.timestamp);
257             // check that you've earned it
258             // this won't happen until a week has passed
259             if (_nextEpochStart > prevRewards.timestamp) {
260                 reward += prevRewards.balanceOf;
261             }
262
263
264             prevRewards.timestamp = _nextEpochStart;
265             _prevSupply = supplyCheckpoints[getPriorSupplyIndex(_nextEpochStart + DURATION)].supply
                    ;
266             prevRewards.balanceOf = (cp0.balanceOf * tokenRewardsPerEpoch[token][_nextEpochStart])
                    / _prevSupply;
267         }
268     }
269
270
271     Checkpoint memory cp = checkpoints[account][_endIndex];
272     uint _lastEpochStart = _bribeStart(cp.timestamp);
273     uint t_i = _bribeStart(Math.max(_startTimestamp, _lastEpochStart));
274     {
275         while (true) {
276             t_i += DURATION;
277             if (t_i > block.timestamp) {
278                 break;
279             }
280             reward +=
281                 (cp.balanceOf * tokenRewardsPerEpoch[token][t_i - DURATION]) /
```

```
282                 supplyCheckpoints[getPriorSupplyIndex(t_i)].supply;
283             }
284         }
285         return reward;
286     }
```

**Listing 2.7:** Bribe.sol

**Impact** The rewards for the second-to-last `checkpoint` of the user will never be claimed.

**Suggestion** Include the rewards of the second-to-last `checkpoint`.

### 2.2.2 Miscalculated Bribe Rewards (II)

**Severity** High

**Status** Fixed in in `Version 2`

**Introduced by** `Version 1`

**Description** As mentioned in `Issue-1`, the function calculates rewards for voting users in two steps. In the first step, it traverses from the `checkpoint` where the user last claimed the reward to the latest `checkpoint`. However, this implementation does not take into account the possibility of epochs between two checkpoints where no `checkpoint` exists.

Specifically, the loop only calculates rewards for each epoch based on the `checkpoint` it belongs to, and any epochs without a corresponding checkpoint are skipped altogether, which results in the loss of rewards for users in epochs without a corresponding `checkpoint`.

```
181   function earned(address token, address account) public view returns (uint) {
182     uint _startTimestamp = lastEarn[token][account];
183     if (numCheckpoints[account] == 0) {
184         return 0;
185     }
186
187
188     uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
189     uint _endIndex = numCheckpoints[account] - 1;
190
191
192     uint reward = 0;
193     // you only earn once per epoch (after it's over)
194     Checkpoint memory prevRewards; // reuse struct to avoid stack too deep
195     prevRewards.timestamp = _bribeStart(_startTimestamp);
196     uint _prevSupply = 1;
197
198
199     if (_endIndex > 0) {
200         for (uint i = _startIndex; i <= _endIndex - 1; i++) {
201             Checkpoint memory cp0 = checkpoints[account][i];
202             uint _nextEpochStart = _bribeStart(cp0.timestamp);
203             // check that you've earned it
204             // this won't happen until a week has passed
205             if (_nextEpochStart > prevRewards.timestamp) {
206                 reward += prevRewards.balanceOf;
```

```
207             }
208
209
210             prevRewards.timestamp = _nextEpochStart;
211             _prevSupply = supplyCheckpoints[getPriorSupplyIndex(_nextEpochStart + DURATION)].supply
                    ;
212             prevRewards.balanceOf = (cp0.balanceOf * tokenRewardsPerEpoch[token][_nextEpochStart])
                    / _prevSupply;
213         }
214     }
215
216
217     Checkpoint memory cp = checkpoints[account][_endIndex];
218     uint _lastEpochStart = _bribeStart(cp.timestamp);
219     uint t_i = _bribeStart(Math.max(_startTimestamp, _lastEpochStart));
220     {
221         while (true) {
222             t_i += DURATION;
223             if (t_i > block.timestamp) {
224                 break;
225             }
226             reward +=
227                 (cp.balanceOf * tokenRewardsPerEpoch[token][t_i - DURATION]) /
228                 supplyCheckpoints[getPriorSupplyIndex(t_i)].supply;
229         }
230     }
231     return reward;
232 }
```

**Listing 2.8:** Bribe.sol

**Impact** Users will receive less rewards than expected.

**Suggestion** Implement corresponding logic to sum up the rewards of epochs that have no corresponding `checkpoints`.

### 2.2.3 Timely invocation of update_period() before setReleaseFactor() and setPledgeFactor()

**Severity** Medium

**Status** Acknowledged

**Introduced by** `Version 1`

**Description** The contract `Minter` is designed to periodically mint and distribute system rewards (i.e. `NOAH` and `esNOAH` tokens). The amount of weekly rewards of `esNOAH` is calculated based on the constant `releaseFactor`, `totalSupply` of `NOAH` and `esNOAH`. These rewards will be allocated to the LPs of various `Gauges`, while the remaining rewards will be distributed as incentives for users who lock their `NOAHs` in the contract `VotingEscrow`. For the rewards of `LPs`, it's also calculated based on `totalSupply` of `NOAH` and `esNOAH`, but using another constant `pledgeFactor`.

The aforementioned two constants `releaseFactor` and `pledgeFactor` are allowed to be modified by the team through privileged functions `setReleaseFactor()` and `setPledgeFactor()`. However, before the update, the contract will not invoke the function `update_period()` to update and distribute rewards of the last epoch, which could result in the previous epoch's rewards being directly changed. It's unfair to the contract users.

```
163    function update_period() external returns (uint) {
164        uint _period = esnoah_mining_active_period;
165        uint _esnoah_minted = esnoah_minted;
166        uint _esnoah_mining_weekly = esnoah_mining_weekly;
167        if (_esnoah_mining_weekly == 0) return _period;
168
169
170        if (block.timestamp >= _period + WEEK && initializer == address(0)) {
171            // only trigger if new week
172            _period = (block.timestamp / WEEK) * WEEK;
173            esnoah_mining_active_period = _period;
174
175
176            uint _left = ESNOAH_MINING_CAP - _esnoah_minted;
177            if (_esnoah_mining_weekly > _left) {
178                _esnoah_mining_weekly = _left;
179                esnoah_mining_weekly = 0;
180            } else {
181                esnoah_mining_weekly = (_esnoah_mining_weekly * EMISSION) / PRECISION;
182            }
183
184
185            // minted
186            esnoah_minted = _esnoah_minted + _esnoah_mining_weekly;
187
188
189            uint _balanceOf = _esNoah.balanceOf(address(this));
190            if (_balanceOf < _esnoah_mining_weekly) {
191                _esNoah.mint(address(this), _esnoah_mining_weekly - _balanceOf);
192            }
193
194
195            uint weekly_reward = calculate_reward(_esnoah_mining_weekly);
196            uint weekly_burn = _esnoah_mining_weekly - weekly_reward;
197            uint weekly_liquidity = calculate_liquidity(weekly_reward);
198            uint weekly_reward_distributor = weekly_reward - weekly_liquidity;
199
200
201            // burn
202            require(_esNoah.transfer(BLACK_HOLE, weekly_burn));
203            // reward stake noah
204            require(_esNoah.transfer(address(_rewards_distributor), weekly_reward_distributor));
205            _rewards_distributor.checkpoint_token(); // checkpoint token balance that was just
                   minted in rewards distributor
206            _rewards_distributor.checkpoint_total_supply(); // checkpoint supply
207            // liquidity
```

```
208             _voter.notifyRewardAmount(weekly_liquidity);
209
210
211             emit Mint(msg.sender, _esnoah_mining_weekly, weekly_liquidity,
                    weekly_reward_distributor, weekly_burn);
212         }
213         return _period;
214     }
```

**Listing 2.9:** Minter.sol

```
102     function setReleaseFactor(uint _releaseFactor) external override {
103         require(msg.sender == team, "not team");
104         releaseFactor = _releaseFactor;
105     }
```

**Listing 2.10:** Minter.sol

```
107     function setPledgeFactor(uint _pledgeFactor) external override {
108     require(msg.sender == team, "not team");
109     pledgeFactor = _pledgeFactor;
110 }
```

**Listing 2.11:** Minter.sol

**Impact** Users may receive less rewards than expected.

**Suggestion** Invoke the function `update_period()` before modifying the `releaseFactor` and `pledgeFactor`.

**Feedback from the Project** The logic of the function `update_period()` determines the number of mining and staking rewards in esNOAH for the week, and it will be executed at the start of each epoch (i.e., after 0:00 on every Thursday UTC). Executing this function at other times during the same epoch will not take effect. While adjustments to the parameters are typically made within the current epoch, the new parameters will only take effect after the start of the next epoch (i.e., the next Thursday). It is not necessary to execute the function `update_period()` every time the parameters are adjusted.

### 2.2.4 Timely invocation of distribute() in notifyRewardAmount()

**Severity** Medium

**Status** Confirmed

**Introduced by** `Version 1`

**Description** According to the design, the distribution of `esNOAHs` among various `Gauges` is determined by the voting results of users holding `veNOAH` in the contract `Voter`. This portion of rewards is transferred by the contract `Minter` via the function `notifyRewardAmount()`. However, based on the current implementation, these rewards are not directly settled and distributed to each `Gauge` based on the current votes after being transferred to the contract `Voter`. In this case, a malicious user is able to frontrun the invocation of the function `distribute()` to distribute the rewards to a specific `Gauge`, and vote for another `Gauge` right after that, which votes twice with one ballot. Besides, although the function `distribute()` will be triggered when the user claims rewards, the rewards may be delayed.

```
338    function distribute(address _gauge) public lock {
339        IMinter(minter).update_period();
340        _updateFor(_gauge); // should set claimable to 0 if killed
341        uint _claimable = claimable[_gauge];
342        if (_claimable > IGauge(_gauge).left(base) && _claimable / DURATION > 0) {
343            claimable[_gauge] = 0;
344            IGauge(_gauge).notifyRewardAmount(base, _claimable);
345            emit DistributeReward(msg.sender, _gauge, _claimable);
346        }
347    }
```

**Listing 2.12:** Voter.sol

```
320    function notifyRewardAmount(uint amount) external {
321    _safeTransferFrom(base, msg.sender, address(this), amount); // transfer the distro in
322    uint _ratio = (amount * 1e18) / totalWeight; // 1e18 adjustment is removed during claim
323    if (_ratio > 0) {
324        index += _ratio;
325    }
326    emit NotifyReward(msg.sender, base, amount);
327}
```

**Listing 2.13:** Voter.sol

**Impact** The reward distribution may be delayed, resulting in loss of rewards for certain users to experience loss.

**Suggestion** Invoke the function `distribute()` directly after the original logic in the function `notifyRewardAmount()` is executed.

**Feedback from the Project** In order to reduce the gas costs incurred by users when invoking the function, the `team` will also promptly call the function `distribute()` after the start of each epoch, in the same manner as regular users.

### 2.2.5 Reward for Killed Gauge Being Locked

**Severity** Medium

**Status** Confirmed

**Introduced by** `Version 1`

**Description** In contract `Voter`, users can vote for each `Gauge`. The reward of each epoch will be allocated to the corresponding gauge according to the proportion of votes in each pool. The `Gauge` can be disabled and enabled through the function `killGauge()` and `reviveGauge()` by the privileged account `emergencyCouncil`. However, a disabled `Gauge` is still votable, and is included in the calculation of the reward distribution, but not claimable.

```
359    function _updateFor(address _gauge) internal {
360        address _pool = poolForGauge[_gauge];
361        uint _supplied = weights[_pool];
362        if (_supplied > 0) {
363            uint _supplyIndex = supplyIndex[_gauge];
```

```
364             uint _index = index; // get global index0 for accumulated distro
365             supplyIndex[_gauge] = _index; // update _gauge current position to global position
366             uint _delta = _index - _supplyIndex; // see if there is any difference that need to be
                    accrued
367             if (_delta > 0) {
368                 uint _share = (uint(_supplied) * _delta) / 1e18; // add accrued difference for each
                        supplied token
369                 if (isAlive[_gauge]) {
370                     claimable[_gauge] += _share;
371                 }
372             }
373         } else {
374             supplyIndex[_gauge] = index; // new users are set to the default global state
375         }
376     }
```

**Listing 2.14:** Voter.sol

**Impact** Users who vote for "killed" `Gauges` will receive no rewards.

**Suggestion** Restrict users from voting for "killed" `Gauges`.

**Feedback from the Project** Which `pool` to vote for is entirely decided by the users, and the `team` will not restrict users' voting behavior. However, the frontend page will provide information on whether a `pool` is voteable, to prevent users from voting for `pools` that are not voteable.

### 2.2.6 Lack of Checks for Gauges that Do Not Support Voting

**Severity** Medium

**Status** Confirmed

**Introduced by** `Version 1`

**Description** In the contract `Voter`, the user is allowed to vote for various `Gauges` via the function `vote()`. The function will allocate the user's existing `veNOAH` based on the voting weights set by the user for `Gauges`. If a `Gauge` does not support voting, the function will skip it, resulting in the votes of this portion of `veNOAH` not being utilized. The user has to wait until the next epoch (up to a maximum of 7 days) to vote for the other pools. In this case, it's suggested to revert when the user tries to vote on the `Gauge` that is not supporting voting.

```
218     function _vote(address _account, address[] memory _poolVote, uint[] memory _weights) internal
            {
219     _reset(_account);
220     uint _poolCnt = _poolVote.length;
221     uint _weight = IVotingEscrow(_ve).balanceOf(_account);
222     uint _totalVoteWeight = 0;
223     uint _totalWeight = 0;
224     uint _usedWeight = 0;
225
226
227     for (uint i = 0; i < _poolCnt; i++) {
228         _totalVoteWeight += _weights[i];
229     }
```

```
230
231
232     for (uint i = 0; i < _poolCnt; i++) {
233         address _pool = _poolVote[i];
234         address _gauge = gauges[_pool];
235         if (isVotableGauge[_gauge]) {
236             uint _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
237             require(votes[_account][_pool] == 0);
238             require(_poolWeight != 0);
239             _updateFor(_gauge);
240
241
242             poolVote[_account].push(_pool);
243
244
245             uint _newWeights = weights[_pool] + _poolWeight;
246             weights[_pool] = _newWeights;
247             votes[_account][_pool] += _poolWeight;
248             IBribe(bribes[_gauge])._deposit(uint(_poolWeight), _account);
249             _usedWeight += _poolWeight;
250             _totalWeight += _poolWeight;
251             emit Voted(_account, _poolWeight);
252             emit PoolVoted(_pool, _newWeights);
253         }
254     }
255     if (_usedWeight > 0) IVotingEscrow(_ve).voting(_account);
256     uint newTotalWeight = totalWeight + uint(_totalWeight);
257     totalWeight = newTotalWeight;
258     usedWeights[_account] = uint(_usedWeight);
259     emit TotalWeight(newTotalWeight);
260 }
```

**Listing 2.15:** Voter.sol

**Impact** Users have to wait for 7 days before they can vote again to correct any erroneous votes.

**Suggestion** Restrict users from voting for non votable `Gauge`.

**Feedback from the Project** Voting for project `Gauges` is controlled by a whitelist, if the code prevents `Gauge` that are not in the whitelist from voting, it may cause anomalies when updating users who have already voted. For example, if `Gauge` A is open for voting for the first three weeks, but disables voting for the fourth week, users who have already voted for `Gauge` A will encounter errors when the function `poke()` is triggered.

### 2.2.7 Reward Token can be Managed by Users with Different Privileges

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The contract `Bribe` plays a role in recording and distributing rewards to the voting users for incentivizing more users to participate in voting. The rewards can be any token listed on the whitelist set by

the gov. After each distribution of a new type of token as a reward through the function notifyRewardAmount(), the function will record it in the mapping isReward[]. This allows skipping unnecessary whitelist checks in the future. However, the team is able to directly modify the mapping isReward[] via the privileged function swapOutRewardToken(), without the need for whitelist checks for newly added reward tokens.

```
313    function swapOutRewardToken(uint i, address oldToken, address newToken) external {
314    require(msg.sender == IVotingEscrow(_ve).team(), "only team");
315    require(rewards[i] == oldToken);
316    isReward[oldToken] = false;
317    isReward[newToken] = true;
318    rewards[i] = newToken;
319}
```

**Listing 2.16:** Bribe.sol

```
249    function whitelist(address _token) public onlyGov {
250    _whitelist(_token);
251}
```

**Listing 2.17:** Voter.sol

```
253    function _whitelist(address _token) internal {
254        require(!isWhitelisted[_token]);
255        isWhitelisted[_token] = true;
256        emit Whitelisted(msg.sender, _token);
257    }
```

**Listing 2.18:** Voter.sol

```
290    function notifyRewardAmount(address token, uint amount) external lock {
291    require(amount > 0);
292    if (!isReward[token]) {
293        require(IVoter(voter).isWhitelisted(token), "bribe tokens must be whitelisted");
294        require(rewards.length < MAX_REWARD_TOKENS, "too many rewards tokens");
295    }
296    // bribes kick in at the start of next bribe period
297    uint adjustedTstamp = getEpochStart(block.timestamp);
298    uint epochRewards = tokenRewardsPerEpoch[token][adjustedTstamp];
299
300
301    _safeTransferFrom(token, msg.sender, address(this), amount);
302    tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards + amount;
303
304
305    periodFinish[token] = adjustedTstamp + DURATION;
306
307
308    if (!isReward[token]) {
309        isReward[token] = true;
310        rewards.push(token);
311    }
312
313
```

```
314     emit NotifyReward(msg.sender, token, adjustedTstamp, amount);
315 }
```

**Listing 2.19:** Bribe.sol

**Impact** The team can bypass the check of whitelist by modifying the mapping `isReward[]` via the privileged function `swapOutRewardToken()`.

**Suggestion** Add the check to ensure the newly added reward token is included in the whitelist.

### 2.2.8 Timely invocation of claimfees() in Gauge

**Severity** Medium

**Status** Acknowledged

**Introduced by** `Version 1`

**Description** According to the design, a portion of transaction fee will be distributed to the contract `Bribe` to incentivize the voting. However, this has to be manually triggered by someone via the function `claimFees()`. In this case, the users may receive less rewards than expected.

```
21     function _claimFees() internal virtual override returns (uint[] memory claimed) {
22         claimed = new uint[](1);
23         if (!isForPair) {
24             return claimed;
25         }
26         IVoter _voter = IVoter(voter);
27         IFeeVault _feeVault = IFeeVault(_voter.feeVault());
28
29
30         claimed[0] = _feeVault.claimFees(stake);
31         if (claimed[0] == 0) {
32             return claimed;
33         }
34
35
36         address _bribe = bribe;
37         // no body vote
38         if (IBribe(_bribe).totalSupply() == 0 || !_voter.isVotableGauge(address(this))) {
39             _safeTransferFrom(stake, address(this), _feeVault.feeTo(), claimed[0]);
40         } else {
41             uint _fees0 = fees[0] + claimed[0];
42             address _token0 = stake;
43             if (_fees0 > IBribe(_bribe).left(_token0) && _fees0 / DURATION > 0) {
44                 fees[0] = 0;
45                 _safeApprove(_token0, _bribe, _fees0);
46                 IBribe(_bribe).notifyRewardAmount(_token0, _fees0);
47             } else {
48                 fees[0] = _fees0;
49             }
50         }
51
52
53         emit ClaimFees(msg.sender, claimed);
```

```
54    }
```

**Listing 2.20:** Gauge.sol

**Impact** Rewards for voters are delayed, and what's worse, voters may lose the rewards.

**Suggestion** Ensure the function `claimFees()` will be triggered by the team periodically and timely.

**Feedback from the Project** To reduce the gas costs incurred by users when invoking the function, the `team` will promptly call the function `claimFees()` in the same manner as regular users after the start of each epoch.

### 2.2.9 Failed to Notify Rewards due to the Reentrancy Lock

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `notifyRewardAmount()` in the contract `Gauge` contains a reentrancy guard (i.e. the modifier `lock`), and it claims swap fees from the contract `FeeVault` via the call stack `nofityRewardAmount` -> `_claimFees` -> `_feeVault.claimFees`. However, the function `claimFees()` in the contract `FeeVault` re-enters the original function `notifyRewardAmount()` in the caller `Gauge` when sufficient swap fees accumulated. This can result in the revert of transaction as the original function `notifyRewardAmount()` is in a lock state.

```
545   function notifyRewardAmount(address token, uint amount) external lock {
546   IVoter _voter = IVoter(voter);
547   require(_voter.isGaugeHandler(msg.sender));
548   require(token != stake);
549   require(amount > 0);
550   if (!isReward[token]) {
551       require(rewards.length < MAX_REWARD_TOKENS, "too many rewards tokens");
552   }
553   if (rewardRate[token] == 0) _writeRewardPerTokenCheckpoint(token, 0, block.timestamp);
554   (rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRewardPerToken(token, type(uint)
          .max, true);
555   _claimFees();
556
557
558   if (block.timestamp >= periodFinish[token]) {
559       _safeTransferFrom(token, msg.sender, address(this), amount);
560       rewardRate[token] = amount / DURATION;
561   } else {
562       uint _remaining = periodFinish[token] - block.timestamp;
563       uint _left = _remaining * rewardRate[token];
564       require(amount > _left);
565       _safeTransferFrom(token, msg.sender, address(this), amount);
566       rewardRate[token] = (amount + _left) / DURATION;
567   }
568   require(rewardRate[token] > 0);
569   uint balance = IERC20(token).balanceOf(address(this));
570   require(rewardRate[token] <= balance / DURATION, "Provided reward too high");
```

```
571     periodFinish[token] = block.timestamp + DURATION;
572     if (!isReward[token]) {
573         isReward[token] = true;
574         rewards.push(token);
575     }
576
577
578     emit NotifyReward(msg.sender, token, amount);
579 }
```

**Listing 2.21:** BaseGauge.sol

```
29      function _claimFees() internal virtual override returns (uint[] memory claimed) {
30          claimed = new uint[](1);
31          if (!isForPair) {
32              return claimed;
33          }
34          IVoter _voter = IVoter(voter);
35          IFeeVault _feeVault = IFeeVault(_voter.feeVault());
36
37
38          claimed[0] = _feeVault.claimFees(stake);
39          if (claimed[0] == 0) {
40              return claimed;
41          }
42
43
44          address _bribe = bribe;
45          // no body vote
46          if (IBribe(_bribe).totalSupply() == 0 || !_voter.isVotableGauge(address(this))) {
47              _safeTransferFrom(stake, address(this), _feeVault.feeTo(), claimed[0]);
48          } else {
49              uint _fees0 = fees[0] + claimed[0];
50              address _token0 = stake;
51              if (_fees0 > IBribe(_bribe).left(_token0) && _fees0 / DURATION > 0) {
52                  fees[0] = 0;
53                  _safeApprove(_token0, _bribe, _fees0);
54                  IBribe(_bribe).notifyRewardAmount(_token0, _fees0);
55              } else {
56                  fees[0] = _fees0;
57              }
58          }
59
60
61          emit ClaimFees(msg.sender, claimed);
62      }
```

**Listing 2.22:** Gauge.sol

```
45      function claimFees(address token) external returns (uint forVote) {
46      require(voter.poolForGauge(msg.sender) == token);
47      uint balance = IERC20(token).balanceOf(address(this));
48      if (balance > 0) {
```

```
49        forVote = (balance * (PRECISION - teamRate)) / PRECISION;
50
51
52        if (forVote > 0) {
53            IERC20(token).approve(msg.sender, forVote);
54            IGauge(msg.sender).notifyRewardAmount(token, forVote);
55        }
56
57
58        IERC20(token).safeTransfer(feeTo, balance - forVote);
59    }
60}
```

**Listing 2.23:** FeeVault.sol

**Impact** Invoking the function `notifyRewardAmount()` within the `Gauge` will result in a revert due to the inappropriate reentrancy lock, thus preventing the distribution of rewards to the `Gauge`.

**Suggestion** Ensure proper use of the re-entrancy lock.

### 2.2.10 Swap Fee Rewards cannotDistribution Mechanism does not Work

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The contract `Gauge` claims fees from the contract `FeeVault` via the external call `_feeVault.claimFees(sta` in the function `_claimFees()`. The contract `FeeVault` will then invoke the function `notifyRewardAmount()` to send the reward. However, in the function `notifyRewardAmount()` of contract `Gauge`, there is a requirement `token != stake` that will lead to revert of the transaction.

```
29    function _claimFees() internal virtual override returns (uint[] memory claimed) {
30        claimed = new uint[](1);
31        if (!isForPair) {
32            return claimed;
33        }
34        IVoter _voter = IVoter(voter);
35        IFeeVault _feeVault = IFeeVault(_voter.feeVault());
36
37
38        claimed[0] = _feeVault.claimFees(stake);
39        if (claimed[0] == 0) {
40            return claimed;
41        }
42
43
44        address _bribe = bribe;
45        // no body vote
46        if (IBribe(_bribe).totalSupply() == 0 || !_voter.isVotableGauge(address(this))) {
47            _safeTransferFrom(stake, address(this), _feeVault.feeTo(), claimed[0]);
48        } else {
49            uint _fees0 = fees[0] + claimed[0];
```

```
50            address _token0 = stake;
51            if (_fees0 > IBribe(_bribe).left(_token0) && _fees0 / DURATION > 0) {
52                fees[0] = 0;
53                _safeApprove(_token0, _bribe, _fees0);
54                IBribe(_bribe).notifyRewardAmount(_token0, _fees0);
55            } else {
56                fees[0] = _fees0;
57            }
58        }
59
60
61        emit ClaimFees(msg.sender, claimed);
62    }
```

**Listing 2.24:** Gauge.sol

```
45    function claimFees(address token) external returns (uint forVote) {
46    require(voter.poolForGauge(msg.sender) == token);
47    uint balance = IERC20(token).balanceOf(address(this));
48    if (balance > 0) {
49        forVote = (balance * (PRECISION - teamRate)) / PRECISION;
50
51
52        if (forVote > 0) {
53            IERC20(token).approve(msg.sender, forVote);
54            IGauge(msg.sender).notifyRewardAmount(token, forVote);
55        }
56
57
58        IERC20(token).safeTransfer(feeTo, balance - forVote);
59    }
60}
```

**Listing 2.25:** FeeVault.sol

```
548   function notifyRewardAmount(address token, uint amount) external lock {
549   IVoter _voter = IVoter(voter);
550   require(_voter.isGaugeHandler(msg.sender));
551   require(token != stake);
552   require(amount > 0);
553   if (!isReward[token]) {
554       require(rewards.length < MAX_REWARD_TOKENS, "too many rewards tokens");
555   }
556   if (rewardRate[token] == 0) _writeRewardPerTokenCheckpoint(token, 0, block.timestamp);
557   (rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRewardPerToken(token, type(uint)
          .max, true);
558   _claimFees();
559
560
561   if (block.timestamp >= periodFinish[token]) {
562       _safeTransferFrom(token, msg.sender, address(this), amount);
563       rewardRate[token] = amount / DURATION;
564   } else {
```

```
565         uint _remaining = periodFinish[token] - block.timestamp;
566         uint _left = _remaining * rewardRate[token];
567         require(amount > _left);
568         _safeTransferFrom(token, msg.sender, address(this), amount);
569         rewardRate[token] = (amount + _left) / DURATION;
570     }
571     require(rewardRate[token] > 0);
572     uint balance = IERC20(token).balanceOf(address(this));
573     require(rewardRate[token] <= balance / DURATION, "Provided reward too high");
574     periodFinish[token] = block.timestamp + DURATION;
575     if (!isReward[token]) {
576         isReward[token] = true;
577         rewards.push(token);
578     }
579
580
581     emit NotifyReward(msg.sender, token, amount);
582 }
```

**Listing 2.26:** BaseGauge.sol

**Impact** The rewards cannot be distributed to the `Gauge` due to the revert caused by improper require statement.

**Suggestion** Ensure proper use of the re-entrancy lock.

### 2.2.11 Manipulated Unlocking Duration

**Severity** High

**Status** Fixed in `Version 3`

**Introduced by** `Version 2`

**Description** The function `_updateVesting()` within the contract `Vester` is implemented to record and update the status of user-locked assets. If the `OperationType` is `DEPOSIT_TYPE` or `SYNC_TYPE`, the unlocking duration will be updated with the current `timestamp` correspondingly. This design is intended to synchronize the changes in the amount of `veNOAH` of the user caused by the user-related actions in the contract `VotingEscrow`.

However, the function `deposit_for()` allows other users to increase the locking amount of a specific user, thereby updating the unlocking duration for that user. Since the unlocking duration is calculated based on the latest `timestamp`, the unlocking duration will be extended. In this case, a malicious user can simply deposit 1 `wei` for others to manipulate their unlocking duration.

```
134     function _updateVesting(address account, OperationType _type) private {
135     uint amount = _getNextClaimableAmount(account);
136     Vesting storage _vesting = _lastVesting[account];
137     IVotingEscrow.LockedBalance memory lockedBalance = ve.locked(account);
138     _vesting.lastTime = block.timestamp;
139     _vesting.lockedAmount = uint(int256(lockedBalance.amount));
140     if (_type == OperationType.DEPOSIT_TYPE || _type == OperationType.SYNC_TYPE) {
141         uint unlock_duration = YEAR;
142         if (lockedBalance.end > block.timestamp) {
```

```
143             unlock_duration = YEAR - (lockedBalance.end - block.timestamp) / 6;
144         }
145         // The maximum unlocking duration is 1 year. The minimum unlocking duration is 6 months
146         _vesting.duration = unlock_duration;
147     }
148
149
150     if (amount == 0) {
151         return;
152     }
153
154
155     balances[account] -= amount;
156     totalVesting -= amount;
157
158
159     claimableAmounts[account] += amount;
160     IEsNoah(esToken).burn(amount);
161 }
```

**Listing 2.27:** Vester.sol

```
111     function syncWithVotingEscrow(address account) external {
112     require(msg.sender == address(ve), "no voting escrow");
113     _updateVesting(account, OperationType.SYNC_TYPE);
114 }
```

**Listing 2.28:** Vester.sol

```
408     function increase_amount(uint _value) external nonreentrant {
409     assert_not_contract(msg.sender);
410     LockedBalance memory _locked = locked[msg.sender];
411
412
413     assert(_value > 0); // dev: need non-zero value
414     require(_locked.amount > 0, "No existing lock found");
415     require(_locked.end > block.timestamp, "Cannot add to expired lock. Withdraw");
416
417
418     _deposit_for(msg.sender, _value, 0, _locked, DepositType.INCREASE_LOCK_AMOUNT);
419 }
```

**Listing 2.29:** VotingEscrow.sol

```
314     function _deposit_for(
315     address _account,
316     uint _value,
317     uint unlock_time,
318     LockedBalance memory locked_balance,
319     DepositType deposit_type
320 ) internal {
321     LockedBalance memory _locked = locked_balance;
322     uint supply_before = supply;
```

```
323
324
325     supply = supply_before + _value;
326     LockedBalance memory old_locked;
327     (old_locked.amount, old_locked.end) = (_locked.amount, _locked.end);
328     // Adding to existing lock, or if a lock is expired - creating a new one
329     _locked.amount += int128(int256(_value));
330     if (unlock_time != 0) {
331         _locked.end = unlock_time;
332     }
333     locked[_account] = _locked;
334
335
336     // Possibilities:
337     // Both old_locked.end could be current or expired (>/< block.timestamp)
338     // value == 0 (extend lock) or value > 0 (add to lock or extend lock)
339     // _locked.end > block.timestamp (always)
340     _checkpoint(_account, old_locked, _locked);
341
342
343     address from = msg.sender;
344     if (_value != 0) {
345         assert(IERC20(token).transferFrom(from, address(this), _value));
346     }
347
348
349     _syncWithVotingEscrow(_account);
350
351
352     emit Deposit(from, _account, _value, _locked.end, deposit_type, block.timestamp);
353     emit Supply(supply_before, supply_before + _value);
354 }
```

**Listing 2.30:** VotingEscrow.sol

```
351     function _syncWithVotingEscrow(address _account) internal {
352         EnumerableSet.AddressSet storage gauges = _attachments[_account];
353         uint _count = gauges.length();
354         for (uint i; i < _count; i++) {
355             IVotingEscrowCallback(gauges.at(i)).syncWithVotingEscrow(_account);
356         }
357         if (vest[_account]) {
358             IVotingEscrowCallback(vester).syncWithVotingEscrow(_account);
359         }
360         if (voted[_account]) {
361             IVoter(voter).poke(_account);
362         }
363     }
```

**Listing 2.31:** VotingEscrow.sol

**Impact** The user's unlocking duration can be extended by a malicious user.

**Suggestion** Set a minimum deposit value in the function `deposit_for()`.

### 2.2.12 Risk of Voting Power Manipulation when is_unlock is True

**Severity** Medium

**Status** Acknowledged

**Introduced by** Version 2

**Description** In the current implementation, the user is allowed to withdraw their locked tokens before the lock end time if the global variable `is_unlocked` is set as true. However, the user is also allowed to lock their tokens in this situation, which poses a risk of potential manipulation of the user's voting power.

```
437    function withdraw(address _account) external nonreentrant {
438    LockedBalance memory _locked = locked[_account];
439    require(block.timestamp >= _locked.end || is_unlocked, "The lock didn't expire and funds are
           not unlocked");
440    uint value = uint(int256(_locked.amount));
441
442
443    locked[_account] = LockedBalance(0, 0);
444    uint supply_before = supply;
445    supply = supply_before - value;
446
447
448    // old_locked can have either expired <= timestamp or zero end
449    // _locked has only 0 end
450    // Both can have >= 0 amount
451    _checkpoint(_account, _locked, LockedBalance(0, 0));
452
453
454    uint time_expire = msg.sender != _account && block.timestamp >= _locked.end + WEEK
455        ? block.timestamp - _locked.end - WEEK
456        : 0;
457    uint penalty_ratio = Math.min(
458        (MULTIPLIER * penalty_factor) / 1000,
459        (MULTIPLIER * time_expire) / MAX_PENALTY_TIME
460    );
461    uint penalty = (value * penalty_ratio) / MULTIPLIER;
462    if (penalty != 0) assert(IERC20(token).transfer(msg.sender, penalty));
463
464
465    assert(IERC20(token).transfer(_account, value - penalty));
466
467
468    _syncWithVotingEscrow(_account);
469
470
471    emit Withdraw(msg.sender, _account, value, penalty, block.timestamp);
472    emit Supply(supply_before, supply_before - value);
473}
```

**Listing 2.32:** VotingEscrow.sol

**Impact** If `is_unlock` is set as true, a malicious user could manipulate their own voting power via `Flashloan`.

**Suggestion** Set all users' voting power to zero when `is_unlock` is true.

**Feedback from the Project** Function `is_unlocked` can only be modified by the privileged role `admin`. The purpose of setting this state variable is to provide an actionable plan for the `admin` during emergencies or exceptional situations, allowing users who have staked for a long period of time to retrieve their assets. Once this state is activated, the project will consider migrating to a new contract.

### 2.2.13 Lack of Check of Function withdrawToken

**Severity** Medium

**Status** Fixed in `Version 4`

**Introduced by** `Version 3`

**Description** The function `withdrawToken()` allows the privileged role `team` to rescue the stuck tokens of the contract for users who accidentally transfer their tokens in. However, as the staking token, `esNOAH` is also allowed to be withdrawn in this function, which is risky for the stakers.

```
47    // to help users who accidentally send their tokens to this contract
48function withdrawToken(address _token, address account, uint amount) external {
49    require(msg.sender == ve.team(), "only team");
50    IERC20(_token).safeTransfer(account, amount);
51}
```

**Listing 2.33:** Vester.sol

**Impact** Team can transfer all the staked `esNOAH` tokens via the function `withdrawToken()`.

**Suggestion** Add check to ensure `esNOAH` can not be withdrawn via the function `withdrawToken()`.

### 2.2.14 Inconsistent Status Update during Voting Process

**Severity** Low

**Status** Fixed in `Version 4`

**Introduced by** `Version 3`

**Description** The internal function `_vote()` of the contract `Voter` is designed to synchronize the voting weight based on the changing `veNOAH` balance of users and the predefined voting weights for each pool. If the balance of `veNOAH` decreases to zero, the internal function `_reset()` will be invoked to clear all the voting weights from the user to all pools. In this case, the function `abstain()` should also be invoked to update the status of the account.

```
207    function _vote(address _account, address[] memory _poolVote, uint[] memory _weights) internal
           {
208    IVotingEscrow ve = IVotingEscrow(_ve);
209    uint _weight = ve.balanceOf(_account);
210    if (_weight == 0) {
211        _reset(_account);
212        return;
213    }
214
215
216    _update_period();
217    uint _poolCnt = _poolVote.length;
```

```
218     uint _totalVoteWeight;
219     uint _usedWeight;
220     uint _oldVotingWeight;
221
222
223     for (uint i; i < _poolCnt; i++) {
224         _totalVoteWeight += _weights[i];
225     }
226
227
228     delete poolVote[_account];
229     address account = _account; //stack too deep
230     for (uint i; i < _poolCnt; i++) {
231         address _pool = _poolVote[i];
232         uint _oldVotes = votes[msg.sender][_pool];
233         if (isVotablePool[_pool]) {
234             uint _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
235             require(_poolWeight != 0);
236
237
238             poolVote[account].push(_pool);
239
240
241             if (_oldVotes == _poolWeight) continue;
242
243
244             _usedWeight += _poolWeight;
245             _oldVotingWeight += _oldVotes;
246
247
248             address _gauge = gauges[_pool];
249             _updateFor(_gauge);
250
251
252             uint _newWeights = weights[_pool] - _oldVotes + _poolWeight;
253             weights[_pool] = _newWeights;
254             votes[account][_pool] = _poolWeight;
255
256
257             IBribe(IGauge(_gauge).bribe())._voted(_poolWeight, account);
258
259
260             emit Voted(account, _pool, _poolWeight);
261             emit PoolVoted(_pool, _newWeights);
262         } else {
263             votes[account][_pool] = 0;
264             emit Abstained(account, _oldVotes);
265         }
266     }
267     if (_oldVotingWeight == 0 && _usedWeight > 0) ve.voting(_account);
268     uint newTotalWeight = totalWeight + _usedWeight - _oldVotingWeight;
269     totalWeight = newTotalWeight;
270     emit TotalWeight(newTotalWeight);
```

```
271 }
```

**Listing 2.34:** Voter.sol

```
152     function reset() external onlyNewEpoch {
153     lastVoted[msg.sender] = block.timestamp;
154     _reset(msg.sender);
155     IVotingEscrow(_ve).abstain(msg.sender);
156 }
```

**Listing 2.35:** Voter.sol

**Impact** The statuses of users may be incorrect.

**Suggestion** Invoke the function `abstain()` after `_reset()` in the function `_vote()`.

### 2.2.15 Miscalculated poolWeight with Duplicated Pool Voting

**Severity** Medium

**Status** Fixed in `Version 4`

**Introduced by** `Version 3`

**Description** In the function `_vote()`, the voting weights for users in the pool are not assigned cumulatively. This leads to inaccuracies when a user casts multiple votes for the same pool during the voting process, resulting in an incorrect voting weight for that pool.

```
207     function _vote(address _account, address[] memory _poolVote, uint[] memory _weights) internal
            {
208     IVotingEscrow ve = IVotingEscrow(_ve);
209     uint _weight = ve.balanceOf(_account);
210     if (_weight == 0) {
211         _reset(_account);
212         return;
213     }
214
215
216     _update_period();
217     uint _poolCnt = _poolVote.length;
218     uint _totalVoteWeight;
219     uint _usedWeight;
220     uint _oldVotingWeight;
221
222
223     for (uint i; i < _poolCnt; i++) {
224         _totalVoteWeight += _weights[i];
225     }
226
227
228     delete poolVote[_account];
229     address account = _account; //stack too deep
230     for (uint i; i < _poolCnt; i++) {
231         address _pool = _poolVote[i];
232         uint _oldVotes = votes[msg.sender][_pool];
```

```
233         if (isVotablePool[_pool]) {
234             uint _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
235             require(_poolWeight != 0);
236
237
238             poolVote[account].push(_pool);
239
240
241             if (_oldVotes == _poolWeight) continue;
242
243
244             _usedWeight += _poolWeight;
245             _oldVotingWeight += _oldVotes;
246
247
248             address _gauge = gauges[_pool];
249             _updateFor(_gauge);
250
251
252             uint _newWeights = weights[_pool] - _oldVotes + _poolWeight;
253             weights[_pool] = _newWeights;
254             votes[account][_pool] = _poolWeight;
255
256
257             IBribe(IGauge(_gauge).bribe())._voted(_poolWeight, account);
258
259
260             emit Voted(account, _pool, _poolWeight);
261             emit PoolVoted(_pool, _newWeights);
262         } else {
263             votes[account][_pool] = 0;
264             emit Abstained(account, _oldVotes);
265         }
266     }
267     if (_oldVotingWeight == 0 && _usedWeight > 0) ve.voting(_account);
268     uint newTotalWeight = totalWeight + _usedWeight - _oldVotingWeight;
269     totalWeight = newTotalWeight;
270     emit TotalWeight(newTotalWeight);
271 }
```

**Listing 2.36:** Voter.sol

**Impact** User's vote weight may be incorrect.

**Suggestion** Update the vote weight cumulatively instead of the direct assignment.

### 2.2.16 Incorrect Reward Calculations from Inappropriate Check

**Severity** High

**Status** Fixed in `Version 4`

**Introduced by** `Version 3`

**Description** In the function `_vote()` of the contract `Voter`, the `_poolWeight` may be 0 due to arithmetic round down. In this scenario, any attempt to invoke the function `_vote()` for this account will result in a failure. A malicious user can craft a `_poolWeight` that decreases to zero during the voting process, thereby ensuring that `poke()` (which invokes `_vote()`) calls can not be made to his/her account until the voting power decreases to zero.

As a result, the user's voting record within the contract `Bribe` will remain static across multiple epochs, which allows him/her to gain more rewards than anticipated.

```
207    function _vote(address _account, address[] memory _poolVote, uint[] memory _weights) internal
           {
208    IVotingEscrow ve = IVotingEscrow(_ve);
209    uint _weight = ve.balanceOf(_account);
210    if (_weight == 0) {
211        _reset(_account);
212        return;
213    }
214
215
216    _update_period();
217    uint _poolCnt = _poolVote.length;
218    uint _totalVoteWeight;
219    uint _usedWeight;
220    uint _oldVotingWeight;
221
222
223    for (uint i; i < _poolCnt; i++) {
224        _totalVoteWeight += _weights[i];
225    }
226
227
228    delete poolVote[_account];
229    address account = _account; //stack too deep
230    for (uint i; i < _poolCnt; i++) {
231        address _pool = _poolVote[i];
232        uint _oldVotes = votes[msg.sender][_pool];
233        if (isVotablePool[_pool]) {
234            uint _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
235            require(_poolWeight != 0);
236
237
238            poolVote[account].push(_pool);
239
240
241            if (_oldVotes == _poolWeight) continue;
242
243
244            _usedWeight += _poolWeight;
245            _oldVotingWeight += _oldVotes;
246
247
248            address _gauge = gauges[_pool];
249            _updateFor(_gauge);
```

```
250
251
252             uint _newWeights = weights[_pool] - _oldVotes + _poolWeight;
253             weights[_pool] = _newWeights;
254             votes[account][_pool] = _poolWeight;
255
256
257             IBribe(IGauge(_gauge).bribe())._voted(_poolWeight, account);
258
259
260             emit Voted(account, _pool, _poolWeight);
261             emit PoolVoted(_pool, _newWeights);
262         } else {
263             votes[account][_pool] = 0;
264             emit Abstained(account, _oldVotes);
265         }
266     }
267     if (_oldVotingWeight == 0 && _usedWeight > 0) ve.voting(_account);
268     uint newTotalWeight = totalWeight + _usedWeight - _oldVotingWeight;
269     totalWeight = newTotalWeight;
270     emit TotalWeight(newTotalWeight);
271 }
```

**Listing 2.37:** Voter.sol

```
195     function poke(address _account) external {
196     address[] memory _poolVote = poolVote[_account];
197     uint _poolCnt = _poolVote.length;
198     uint[] memory _weights = new uint[](_poolCnt);
199
200
201     for (uint i; i < _poolCnt; i++) {
202         _weights[i] = votes[_account][_poolVote[i]];
203     }
204
205
206     _vote(_account, _poolVote, _weights);
207 }
```

**Listing 2.38:** Voter.sol

```
250     function getReward(address account, address[] memory tokens) external lock {
251     address _voter = voter;
252     require(msg.sender == account || msg.sender == _voter);
253
254
255     IVoter(_voter).poke(account);
256     _claimFees();
257
258
259     uint length = tokens.length;
260     for (uint i; i < length; i++) {
261         _claim(tokens[i], account);
```

```
262     }
263 }
```

**Listing 2.39:** Bribe.sol

**Impact** Malicious users are able to earn more rewards than expected.

**Suggestion** Remove the redundant check.

## 2.3 Additional Recommendation

### 2.3.1 Lack of Zero Address Check

**Status** Confirmed

**Introduced by** Version 1

**Description** Lack of zero address check before updating address variables in multiple places, such as function `setEmergencyCouncil()` and `constructor()` in contract `Voter`.

```
100     function setEmergencyCouncil(address _council) public {
101     require(msg.sender == emergencyCouncil);
102     emergencyCouncil = _council;
103 }
```

**Listing 2.40:** Voter.sol

```
161     constructor(address __ve, address _base) {
162     _ve = __ve;
163     base = _base;
164     minter = msg.sender;
165     emergencyCouncil = msg.sender;
166     isGaugeHandler[address(this)] = true;
167 }
```

**Listing 2.41:** Voter.sol

**Suggestion** Add zero address checks accordingly.

### 2.3.2 Redundant Functions

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** In the contract `Voter`, there are two identical functions with different names, one named `distro()` while the other named `distribute()`.

```
399     function distro() external {
400     distribute(0, pools.length);
401 }
```

**Listing 2.42:** Voter.sol

```
403    function distribute() external {
404    distribute(0, pools.length);
405}
```

**Listing 2.43:** Voter.sol

**Suggestion** Remove the redundant function.

### 2.3.3 Redundant Invocation of Function _updateFor

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `setVotableGauge()` allows the privileged `gov` to enable a `Gauge` that can not be voted for, or disable a votable `Gauge`. When the `gov` tries to disable a votable `Gauge`, the function will first distribute unreleased rewards by invoking the function `distribute()`, and invoke the function `_updateFor()` after that to update the reward of the `Gauge`. However, the function `distribute()` itself has already invoked the function `_updateFor()`, thus the function `_updateFor()` in line 126 is redundant.

```
119    function setVotableGauge(address _gauge, bool _isActive) external onlyGov {
120        require(isGauge[_gauge]);
121        require(isVotableGauge[_gauge] != _isActive);
122        isVotableGauge[_gauge] = _isActive;
123        if (!_isActive) {
124                distribute(_gauge);
125
126            _updateFor(_gauge);
127            address _pool = poolForGauge[_gauge];
128            uint newTotalWeight = totalWeight - weights[_pool];
129            totalWeight = newTotalWeight;
130            weights[_pool] = 0;
131
132
133            emit PoolVoted(_pool, 0);
134            emit TotalWeight(newTotalWeight);
135        }
136        emit SetVotableGauge(_gauge, _isActive);
137    }
```

**Listing 2.44:** Voter.sol

```
388    function distribute(address _gauge) public lock {
389    IMinter(minter).update_period();
390    _updateFor(_gauge); // should set claimable to 0 if killed
391    uint _claimable = claimable[_gauge];
392    if (_claimable > IGauge(_gauge).left(base) && _claimable / DURATION > 0) {
393        claimable[_gauge] = 0;
394        IGauge(_gauge).notifyRewardAmount(base, _claimable);
395        emit DistributeReward(msg.sender, _gauge, _claimable);
396    }
397}
```

**Listing 2.45:** Voter.sol

**Suggestion** Remove the redundant function.

### 2.3.4 Meaningless Usage of max

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** There are several meaningless usages of function `Max()` in the contract `RewardDistributor`. The return value of `Max(uint(X), 0)` will always be `X` itself.

```
134     return Math.max(uint(int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.ts))))), 0);
```

**Listing 2.46:** RewardDistributor.sol

```
153     ve_supply[t] = Math.max(uint(int256(pt.bias - pt.slope * dt)), 0);
```

**Listing 2.47:** RewardDistributor.sol

```
203     uint balance_of = Math.max(uint(int256(old_user_point.bias - dt * old_user_point.slope)), 0);
```

**Listing 2.48:** RewardDistributor.sol

```
260     uint balance_of = Math.max(uint(int256(old_user_point.bias - dt * old_user_point.slope)), 0);
```

**Listing 2.49:** RewardDistributor.sol

**Suggestion** Return the X directly instead of invoking the function `Max()`.

### 2.3.5 Inappropriate Variable Naming

**Status** Confirmed

**Introduced by** `Version 1`

**Description** The name of variable `bribeForGauge` in the contract `Voter` is confusing because it uses the key of `bribe` to index the `gauge` in the function `registerGauge`. This is semantically inconsistent with the variable `poolForGauge`, which uses `gauge` as the key to index the `pool`.

**Suggestion** Change `bribeForGauge` to `gaugeForBribe`.

### 2.3.6 Lack of Check for releaseFactor and pledgeFactor

**Status** Confirmed

**Introduced by** `Version 1`

**Description** In the contract `Minter`, the increase of `releaseFactor` and `pledgeFactor` will increase the number of tokens minted in each epoch. They can be updated via the function `setReleaseFactor()` and `setPledgeFactor()` respectively by the privileged role `Owner`. However, there is no check to limit the maximum value of them.

```
130    function setReleaseFactor(uint _releaseFactor) external override onlyOwner {
131    releaseFactor = _releaseFactor;
132}
```

**Listing 2.50:** Minter.sol

```
134    function setPledgeFactor(uint _pledgeFactor) external override onlyOwner {
135    pledgeFactor = _pledgeFactor;
136}
```

**Listing 2.51:** Minter.sol

**Suggestion** Add a check to ensure the `releaseFactor` and `pledgeFactor` will never exceed a reasonable maximum value.

### 2.3.7 Redundant Check in Function mint_marketing

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the function `mint_marketing()` of the contract `Minter`, there are two identical checks (i.e., `require(amount > 0)`; and `require(amount > 0, "fully minted")`;) , which are redundant.

```
138    function mint_marketing(uint amount, address receiver) external override {
139    require(msg.sender == marketer, "not marketer");
140    require(receiver != address(0), "zero address");
141    require(amount > 0);
142
143
144    uint _marketing_minted = marketing_minted;
145    uint _left = MARKETING - _marketing_minted;
146    if (amount > _left) {
147        amount = _left;
148    }
149    require(amount > 0, "fully minted");
150    marketing_minted = _marketing_minted + amount;
151    _noah.mint(receiver, amount);
152    emit MarketingMint(msg.sender, amount);
153}
```

**Listing 2.52:** Minter.sol

**Suggestion** Remove the redundant check.

## 2.4 Notes

### 2.4.1 Potential Centralization Problem

**Status** Confirmed

**Introduced by** `version 1`

**Description** This project has potential centralization problems. The privileged role `team` can change the `releaseFactor` and `pledgeFactor` impacts the volume of `esNoah` minted in each epoch . Meanwhile, it can also change the `teamRate` affects the transaction fee rewards received by users. The privileged role `gov` can change the whitelist token, which changes the reward token users receive in the contract `Bribe`. We suggest these roles should be in multi-signature. and they are out of scope for auditing.

### 2.4.2 Timely deployment contracts

**Status** Confirmed
**Introduced by** `version 1`

**Description** The contract `VotingEscrow`, `FeeVault`, `Minter`, and `RewardDistributor` within the project all initialize the relevant variables in the contracts using the current `block.timestamp` at the time of deployment. The system's token minting and distribution depend on the time variables in these contracts being synchronized within the same week. If they are not consistent, users may receive an incorrect number of rewards.

### 2.4.3 Non-Linear Unlocking in Multiple Claims

**Status** Confirmed
**Introduced by** `version 1`

**Description** The function `_getNextClaimableAmount()` in the contract `vester` is designed to compute the amount of tokens that a user can unlock at the moment. As per the system's design, linear unlocking operates in a linear fashion for a single claim cycle, but it does not maintain this linearity when the user makes multiple claims. It unlocks linearly based on the quantity of remaining `esNOAH` that are yet to be unlocked.

### 2.4.4 Token Release for Team and VC without Time Restrictions

**Status** Confirmed
**Introduced by** `version 2`

**Description** In the contract `Minter`, the function `mint_team()` and `mint_vc()` are utilized to mint `NOAH` and `esNOAH` tokens for the `team` and the `vc` respectively. However, these functions lack time-staggered batches unlocking checks. Consequently, the corresponding privileged role `team` has the ability to mint `NOAH` and `esNOAH` tokens to anyone at any time.

### 2.4.5 Potential Inequity Function poke() of the Contract Voter

**Status** Confirmed
**Introduced by** `version 1`

**Description** The public function `poke()` in the contract `Voter` allows anyone to update votes of any account according to the original proportion as `veNOAH` decreases linearly over time.It will also update the voting amount of the account in the contract `Bribe`. Since the contract `Bribe` calculates the rewards of the voter based on the latest votes in the epoch, the final voting rewards that the voter can receive depend on not only their `veNOAH` amount but also on whether they have been "poked" by others in that epoch.

This mechanism may indeed appear unfair to ordinary users who are unaware of its workings, as they may be poked by others before the end of an epoch while some others are not. It introduces an additional layer of subjectivity and potential bias. The fairness and transparency of the mechanism could be compromised if it relies on manual intervention.

### 2.4.6 Incompatible Tokens

**Status** Confirmed

**Introduced by** `version 1`

**Description** Elastic supply tokens are not compatible with the protocol. They could dynamically adjust their price, supply, user's balance, etc. Such as inflation tokens, deflation tokens, rebasing tokens, and so forth. The inconsistency could result in security impacts if some critical operations are based on the recorded amount of transferred tokens.