Security Audit Report for Stratum Contract

Date: Feb 07, 2024

Version: 1.1

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Stratum
Target	Stratum Contract

Version History

Version	Date	Description
1.0	Sep 21, 2023	First Version
1.1	Feb 07, 2024	First Version (Extension)

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 Stratum [1].

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 commit SHA values 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.

Project		Commit SHA
Stratum	`Version 1`	`ebebe8065759c4544689f503ad955c7184d35216`
	`Version 2`	`0b07d28f063306f636876a47a1dd97942c43f63e`
	`Version 3`	`13b048f1d5175d10a7ae8c6a1a51a04afdcbb0fb`
	`Version 4`	`a0d712e8ac35a36dd1554e1f4083deb57660b03d`
	`Version 5`	`f00f885806805c83652357325b68d5f3c39a3ff4`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **stratum-exchange/v1** folder contract only. Specifically, the files covered in this audit include:

- contracts/factories/PairFactory.sol
- contracts/Gauge.sol
- contracts/Minter.sol
- contracts/Router.sol
- contracts/Stratum.sol
- contracts/Voter.sol
- contracts/VotingEscrow.sol
- contracts/WrappedExternalBribe.sol
- contracts/MetaBribe.sol
- contracts/multipool/AmplificationUtils.sol
- contracts/multipool/LPToken.sol
- contracts/multipool/MathUtils.sol
- contracts/multipool/OwnerPausable.sol
- contracts/multipool/Swap.sol
- contracts/multipool/SwapUtils.sol

[1] https://github.com/stratum-exchange/v1

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 [2] and Common Weakness Enumeration [3]. 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 1.1.

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.

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

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

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **twenty-three** potential issues. Besides, we have **five** recommendations and **four** notes as follows:

- High Risk: 4
- Medium Risk: 18
- Low Risk: 1
- Recommendations: 5
- Notes: 4

ID	Severity	Description	Category	Status
1	Medium	Improper Use of the Keyword Memory	Software Security	Fixed
2	Medium	Improper Use of the Math Library	Software Security	Fixed
3	Medium	Reward Token can be Managed by Users with Different Privileges	DeFi Security	Fixed
4	Medium	Incorrect Update on the checkpoints.timestamp	DeFi Security	Acknowledged
5	Medium	Lack of Check for the Parameter _expiresAt	DeFi Security	Fixed
6	Medium	Lack of Check for the Loan Restriction when _noPullback is False.	DeFi Security	Fixed
7	Medium	Locked Reward for Killed Gauge	DeFi Security	Fixed
8	Low	Lack of Checks for Gauges that Do Not Support Voting	DeFi Security	Acknowledged
9	High	Timely Invocation of distribute() in notifyRewardAmount()	DeFi Security	Fixed
10	Medium	Lack of Maximum Cap on Total Supply	DeFi Security	Fixed
11	Medium	Missed Claim Fees for 3pool	DeFi Security	Fixed
12	High	Lack of Access Control in Function removePartnerToken()	DeFi Security	Fixed
13	Medium	Manipulated bribes_value with Spot Price	DeFi Security	Fixed
14	High	Improper Calculation of Total Bribes Value	DeFi Security	Fixed
15	Medium	Incorrect Calculation of the Claimable Amount	DeFi Security	Fixed
16	Medium	Delayed Epoch Rewards	DeFi Security	Acknowledged
17	Medium	Locked Reward for the Expired NFT	DeFi Security	Confirmed
18	High	Incorrect Calculation of Reward in earned()	DeFi Security	Fixed
19	Medium	Timely Update Rewards in Function killGauge()	DeFi Security	Fixed
20	Medium	Potential DoS in check_total_bribes_value()	DeFi Security	Fixed
21	Medium	Incorrect Calculation of the Bribe Value	DeFi Security	Fixed
22	Medium	The Return Value of the Function estimateValue() can be Manipulated	DeFi Security	Fixed
23	Medium	Incorrect Calculation of the MetaBribe_Weight	DeFi Security	Fixed
24	-	Lack of Zero Address Check	Recommendation	Confirmed
25	-	Redundant Functions	Recommendation	Fixed
26	-	Lack of Check for _swapAddress	Recommendation	Fixed
27	-	Lack of Check for _transferFrom	Recommendation	Fixed
28	-	Incorrect Error Message	Recommendation	Fixed
29	-	Potential Centralization Problem	Note	-
30	-	Incompatible Tokens	Note	-
31	-	Timely Pull Back Borrowed NFT for Lenders	Note	-
32	-	Invocation of Function poke() in One Epoch	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 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 the 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 results in the value of each checkpoint to be the same as the value of the first one.

```
600  function _checkpoint(
601  uint _tokenId,
602  LockedBalance memory old_locked,
603  LockedBalance memory new_locked
604) internal {
605  if (_tokenId != 0) {
606      // Calculate slopes and biases
607      // Kept at zero when they have to
608      if (old_locked.end > block.timestamp && old_locked.amount > 0) {
609        u_old.slope = old_locked.amount / iMAXTIME;
610        u_old.bias =
611          u_old.slope *
612          int128(int256(old_locked.end - block.timestamp));
613      }
614      if (new_locked.end > block.timestamp && new_locked.amount > 0) {
615        u_new.slope = new_locked.amount / iMAXTIME;
616        u_new.bias =
617          u_new.slope *
618          int128(int256(new_locked.end - block.timestamp));
619      }
620
621      // Read values of scheduled changes in the slope
622      // old_locked.end can be in the past and in the future
623      // new_locked.end can ONLY by in the FUTURE unless everything expired: than zeros
624      old_dslope = slope_changes[old_locked.end];
625      if (new_locked.end != 0) {
626        if (new_locked.end == old_locked.end) {
```

```
627         new_dslope = old_dslope;
628       } else {
629         new_dslope = slope_changes[new_locked.end];
630       }
631     }
632   }
633 Point memory last_point = Point({
634   bias: 0,
635   slope: 0,
636   ts: block.timestamp,
637   blk: block.number
638 });
639 if (_epoch > 0) {
640   last_point = point_history[_epoch];
641 }
642 uint last_checkpoint = last_point.ts;
643 // initial_last_point is used for extrapolation to calculate block number
644 // (approximately, for *At methods) and save them
645 // as we cannot figure that out exactly from inside the contract
646 Point memory initial_last_point = last_point;
647 uint block_slope = 0; // dblock/dt
648 if (block.timestamp > last_point.ts) {
649   block_slope =
650     (MULTIPLIER * (block.number - last_point.blk)) /
651     (block.timestamp - last_point.ts);
652 }
653 // If last point is already recorded in this block, slope=0
654 // But that's ok b/c we know the block in such case
655
656
657 // Go over weeks to fill history and calculate what the current point is
658 {
659   uint t_i = (last_checkpoint / WEEK) * WEEK;
660   for (uint i = 0; i < 255; ++i) {
661     // Hopefully it won't happen that this won't get used in 5 years!
662     // If it does, users will be able to withdraw but vote weight will be broken
663     t_i += WEEK;
664     int128 d_slope = 0;
665     if (t_i > block.timestamp) {
666       t_i = block.timestamp;
667     } else {
668       d_slope = slope_changes[t_i];
669     }
670     last_point.bias -=
671       last_point.slope *
672       int128(int256(t_i - last_checkpoint));
673     last_point.slope += d_slope;
674     if (last_point.bias < 0) {
675       // This can happen
676       last_point.bias = 0;
677     }
678     if (last_point.slope < 0) {
679       // This cannot happen - just in case
```

```
680         last_point.slope = 0;
681       }
682       last_checkpoint = t_i;
683       last_point.ts = t_i;
684       last_point.blk =
685         initial_last_point.blk +
686         (block_slope * (t_i - initial_last_point.ts)) /
687         MULTIPLIER;
688       _epoch += 1;
689   ***
690     }
691   }
692 }
```

Listing 2.1: VotingEscrow.sol

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

Suggestion Use deep copy for `initial_last_point` assignment.

2.1.2 Improper Use of the Math Library

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MetaBribe`, the statement `ve_supply[t] = Math.max(uint(int256(pt.bias - pt.slope * dt)), 0);` is supposed to return zero if the calculated result is a negative number. However, in the current implementation, it casts the negative result to an unsigned integer (uint), which makes it underflow without reverting. As a result, it will return a large number that exceeds expectations.

```
246     function ve_for_at(
247         uint _tokenId,
248         uint _timestamp
249       ) external view returns (uint) {
250         address ve = voting_escrow;
251         uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
252         uint epoch = _find_timestamp_user_epoch(
253           ve,
254           _tokenId,
255           _timestamp,
256           max_user_epoch
257         );
258         IVotingEscrow.Point memory pt = IVotingEscrow(ve).user_point_history(
259           _tokenId,
260           epoch
261         );
262         return
263           Math.max(
264             uint(int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.ts))))),
265             0
```

```
266         );
267     }
```

Listing 2.2: MetaBribe.sol

```
278   function _checkpoint_total_supply() internal {
279       address ve = voting_escrow;
280       uint t = time_cursor;
281       uint rounded_timestamp = (block.timestamp / WEEK) * WEEK;
282       IVotingEscrow(ve).checkpoint();
283
284
285       for (uint i = 0; i < 20; i++) {
286         if (t > rounded_timestamp) {
287           break;
288         } else {
289           uint epoch = _find_timestamp_epoch(ve, t);
290           IVotingEscrow.Point memory pt = IVotingEscrow(ve).point_history(epoch);
291           int128 dt = 0;
292           if (t > pt.ts) {
293             dt = int128(int256(t - pt.ts));
294           }
295           ve_supply[t] = Math.max(uint(int256(pt.bias - pt.slope * dt)), 0);
296         }
297         t += WEEK;
298       }
299       time_cursor = t;
300     }
```

Listing 2.3: MetaBribe.sol

Impact It will overflow to an enormous value when `(pt.bias - pt.slope * dt)` is negative.

Suggestion Check whether `(pt.bias - pt.slope * dt)` is negative before casting it.

2.2 DeFi Security

2.2.1 Reward Token can be Managed by Users with Different Privileges

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `ExternalBribe` 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 `governor`. 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. The same problem exists in the contracts `WrappedExternalBribe`, `internalBribe` and `Gauge`.

```
335  function swapOutRewardToken(
336  uint i,
337  address oldToken,
338  address newToken
339) external {
340  require(msg.sender == IVotingEscrow(_ve).team(), "only team");
341  require(rewards[i] == oldToken);
342  isReward[oldToken] = false;
343  isReward[newToken] = true;
344  rewards[i] = newToken;
345}
```

Listing 2.4: ExternalBribe.sol

```
309  function notifyRewardAmount(address token, uint amount) external lock {
310  require(amount > 0);
311  if (!isReward[token]) {
312    require(
313      IVoter(voter).isWhitelisted(token),
314      "bribe tokens must be whitelisted"
315    );
316    require(rewards.length < MAX_REWARD_TOKENS, "too many rewards tokens");
317  }
318  // bribes kick in at the start of next bribe period
319  uint adjustedTstamp = getEpochStart(block.timestamp);
320  uint epochRewards = tokenRewardsPerEpoch[token][adjustedTstamp];
321
322
323  _safeTransferFrom(token, msg.sender, address(this), amount);
324  tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards + amount;
325
326
327  periodFinish[token] = adjustedTstamp + DURATION;
328
329
330  if (!isReward[token]) {
331    isReward[token] = true;
332    rewards.push(token);
333  }
334
335
336  emit NotifyReward(msg.sender, token, adjustedTstamp, amount);
337}
```

Listing 2.5: ExternalBribe.sol

```
248    function whitelist(address _token) public {
249        require(msg.sender == governor);
250        _whitelist(_token);
251      }
252
253
254      function _whitelist(address _token) internal {
```

```
255         require(!isWhitelisted[_token]);
256         isWhitelisted[_token] = true;
257         emit Whitelisted(msg.sender, _token);
258       }
```

Listing 2.6: Voter.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.

2.2.2 Incorrect Update on the checkpoints.timestamp

Severity Medium

Status Acknowledged

Introduced by `Version 1`

Description In the contract `VotingEscrow`, the global variable `checkpoints` are modified by the function `_moveTokenDelegate()` and `_moveAllDelegates()`. The function `_findWhatCheckpointToWrite()` uses a comparison between `checkpoints.timestamp` and `block.timestamp` to determine which `checkpoint` should be updated. However, it is worth noting that `checkpoints.timestamp` has never been modified and retains its default value of zero.

```
1381  function _findWhatCheckpointToWrite(
1382    address account
1383  ) internal view returns (uint32) {
1384    uint _timestamp = block.timestamp;
1385    uint32 _nCheckPoints = numCheckpoints[account];
1386
1387
1388    if (
1389      _nCheckPoints > 0 &&
1390      checkpoints[account][_nCheckPoints - 1].timestamp == _timestamp
1391    ) {
1392      return _nCheckPoints - 1;
1393    } else {
1394      return _nCheckPoints;
1395    }
1396  }
```

Listing 2.7: VotingEscrow.sol

```
1334  function _moveTokenDelegates(
1335    address srcRep,
1336    address dstRep,
1337    uint _tokenId
1338  ) internal {
1339    if (srcRep != dstRep && _tokenId > 0) {
1340      if (srcRep != address(0)) {
1341        uint32 srcRepNum = numCheckpoints[srcRep];
1342        uint[] storage srcRepOld = srcRepNum > 0
```

```
1343          ? checkpoints[srcRep][srcRepNum - 1].tokenIds
1344          : checkpoints[srcRep][0].tokenIds;
1345        uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
1346        uint[] storage srcRepNew = checkpoints[srcRep][nextSrcRepNum].tokenIds;
1347        // All the same except _tokenId
1348        for (uint i = 0; i < srcRepOld.length; i++) {
1349          uint tId = srcRepOld[i];
1350          if (tId != _tokenId) {
1351            srcRepNew.push(tId);
1352          }
1353        }
1354
1355
1356        numCheckpoints[srcRep] = srcRepNum + 1;
1357      }
1358
1359
1360      if (dstRep != address(0)) {
1361        uint32 dstRepNum = numCheckpoints[dstRep];
1362        uint[] storage dstRepOld = dstRepNum > 0
1363          ? checkpoints[dstRep][dstRepNum - 1].tokenIds
1364          : checkpoints[dstRep][0].tokenIds;
1365        uint32 nextDstRepNum = _findWhatCheckpointToWrite(dstRep);
1366        uint[] storage dstRepNew = checkpoints[dstRep][nextDstRepNum].tokenIds;
1367        // All the same plus _tokenId
1368        require(
1369          dstRepOld.length + 1 <= MAX_DELEGATES,
1370          "dstRep would have too many tokenIds"
1371        );
1372        for (uint i = 0; i < dstRepOld.length; i++) {
1373          uint tId = dstRepOld[i];
1374          dstRepNew.push(tId);
1375        }
1376        dstRepNew.push(_tokenId);
1377
1378
1379        numCheckpoints[dstRep] = dstRepNum + 1;
1380      }
1381    }
1382  }
```

Listing 2.8: VotingEscrow.sol

```
1397  function _moveAllDelegates(
1398    address owner,
1399    address srcRep,
1400    address dstRep
1401  ) internal {
1402    // You can only redelegate what you own
1403    if (srcRep != dstRep) {
1404      if (srcRep != address(0)) {
1405        uint32 srcRepNum = numCheckpoints[srcRep];
1406        uint[] storage srcRepOld = srcRepNum > 0
```

```
1407          ? checkpoints[srcRep][srcRepNum - 1].tokenIds
1408          : checkpoints[srcRep][0].tokenIds;
1409        uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
1410        uint[] storage srcRepNew = checkpoints[srcRep][nextSrcRepNum].tokenIds;
1411        // All the same except what owner owns
1412        for (uint i = 0; i < srcRepOld.length; i++) {
1413          uint tId = srcRepOld[i];
1414          if (idToOwner[tId] != owner) {
1415            srcRepNew.push(tId);
1416          }
1417        }
1418
1419
1420        numCheckpoints[srcRep] = srcRepNum + 1;
1421      }
1422
1423
1424      if (dstRep != address(0)) {
1425        uint32 dstRepNum = numCheckpoints[dstRep];
1426        uint[] storage dstRepOld = dstRepNum > 0
1427          ? checkpoints[dstRep][dstRepNum - 1].tokenIds
1428          : checkpoints[dstRep][0].tokenIds;
1429        uint32 nextDstRepNum = _findWhatCheckpointToWrite(dstRep);
1430        uint[] storage dstRepNew = checkpoints[dstRep][nextDstRepNum].tokenIds;
1431        uint ownerTokenCount = ownerToNFTokenCount[owner];
1432        require(
1433          dstRepOld.length + ownerTokenCount <= MAX_DELEGATES,
1434          "dstRep would have too many tokenIds"
1435        );
1436        // All the same
1437        for (uint i = 0; i < dstRepOld.length; i++) {
1438          uint tId = dstRepOld[i];
1439          dstRepNew.push(tId);
1440        }
1441        // Plus all that's owned
1442        for (uint i = 0; i < ownerTokenCount; i++) {
1443          uint tId = ownerToNFTokenIdList[owner][i];
1444          dstRepNew.push(tId);
1445        }
1446
1447
1448        numCheckpoints[dstRep] = dstRepNum + 1;
1449      }
1450    }
1451  }
```

Listing 2.9: VotingEscrow.sol

Impact The functions `_moveTokenDelegates()` and `_moveAllDelegates()` update the `checkpoints` inaccurately with incorrect indexes.

Suggestion Update `checkpoints.timestamp` properly.

Feedback from the Project This is in the "DAO voting logic" section in the contract `VotingEscrow`. The

team states that there will **NOT** going to have a `DAO` nor delegations and this issue will not going to affect the protocol.

2.2.3 Lack of Check for the Parameter _expiresAt

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `lend()` of the contract `VotingEscrow`, the parameter `_expiresAt` is utilized to specify the loan's expiration time. However, there is no validation to ensure that the expiration time is later than the current time.

```
1538    function lend(
1539        uint _tokenId,
1540        address _to,
1541        uint _expiresAt,
1542        bool _noPullback,
1543        bool _noWithdraw,
1544        bool _noApprove,
1545        bool _noIncreaseUnlockTime,
1546        bool _noMerge
1547      ) external nonreentrant {
1548        // Check requirements: msg.sender is the owner, approved for _tokenId or operator of owner
1549        require(
1550          _isApprovedOrOwner(msg.sender, _tokenId),
1551          "not owner, approved or operator"
1552        );
1553
1554
1555        // there may be no active restrictions (i.e. veSTRAT is borrowed)
1556        require(!isBorrowed(_tokenId), "token already lent");
1557
1558
1559        // check whats required for lender to be able to pull back veSTRAT later
1560        if (!_noPullback) {   // enforce those restrictions that could otherwise lead to loss
1561          // of 'claim' to that veSTRAT token
1562          require(_noApprove, "pullback requires disabling approve");
1563          require(_noMerge, "pullback requires disabling merge");
1564        }
1565
1566
1567        address owner = ownerOf(_tokenId);
1568
1569
1570        restrictions[_tokenId] = LendingRestrictions({
1571          lender: owner,
1572          expiresAt: _expiresAt,
1573          noPullback: _noPullback,
1574          noWithdraw: _noWithdraw,
1575          noApprove: _noApprove,
1576          noIncreaseUnlockTime: _noIncreaseUnlockTime,
```

```
1577          noMerge: _noMerge
1578        });
1579
1580
1581        // Transfer to borrower
1582        _transferFrom(owner, _to, _tokenId, msg.sender);
1583
1584
1585        if (!_noPullback) {
1586          // The 'return ticket': Approve for current owner (=lender), so lender will be
1587          // able to actively pull back the veSTRAT using pullBack()
1588          idToApprovals[_tokenId] = owner;
1589          emit Approval(owner, _to, _tokenId);
1590        }
1591      }
```

Listing 2.10: VotingEscrow.sol

```
1605    function isBorrowed(uint _tokenId) public view returns (bool) {
1606        // no restrictions if expired (or never set at all)
1607        if (block.timestamp >= restrictions[_tokenId].expiresAt) {
1608          return false;
1609        }
1610
1611
1612        // no restrictions if lender is still the owner: By intention, the lender sets
1613        // the restrictions while owning the veSTRAT, then transfers it to the borrower, where
1614        // the restrictions become active.
1615        if (ownerOf(_tokenId) == restrictions[_tokenId].lender) {
1616          return false;
1617        }
1618
1619
1620        return true;
1621      }
```

Listing 2.11: VotingEscrow.sol

Impact If `_expiresAt` is less than `block.timestamp`, the borrower can immediately bypass all loan restrictions.

Suggestion Add the check to ensure the `_expiresAt` is greater than the `block.timestamp`.

2.2.4 Lack of Check for the Loan Restriction when _noPullback is False.

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description During the lending process, the lender needs to configure the parameter `_noPullback`. If it is set to false, it means that the lender can retrieve the borrowed `NFT` and fully control it. In this scenario, the contract checks the parameters `_noApprove` and `_noMerge` to disable the `borrower's` ability to approve

and merge the borrowed NFT, as both of these actions would permanently transfer ownership of the NFT to the borrower. Based on this design, the borrower should also not be allowed to perform operations such as withdrawing the NFT.

```
1538    function lend(
1539        uint _tokenId,
1540        address _to,
1541        uint _expiresAt,
1542        bool _noPullback,
1543        bool _noWithdraw,
1544        bool _noApprove,
1545        bool _noIncreaseUnlockTime,
1546        bool _noMerge
1547      ) external nonreentrant {
1548        // Check requirements: msg.sender is the owner, approved for _tokenId or operator of owner
1549        require(
1550          _isApprovedOrOwner(msg.sender, _tokenId),
1551          "not owner, approved or operator"
1552        );
1553
1554
1555        // there may be no active restrictions (i.e. veSTRAT is borrowed)
1556        require(!isBorrowed(_tokenId), "token already lent");
1557
1558
1559        // check whats required for lender to be able to pull back veSTRAT later
1560        if (!_noPullback) {   // enforce those restrictions that could otherwise lead to loss
1561          // of 'claim' to that veSTRAT token
1562          require(_noApprove, "pullback requires disabling approve");
1563          require(_noMerge, "pullback requires disabling merge");
1564        }
1565
1566
1567        address owner = ownerOf(_tokenId);
1568
1569
1570        restrictions[_tokenId] = LendingRestrictions({
1571          lender: owner,
1572          expiresAt: _expiresAt,
1573          noPullback: _noPullback,
1574          noWithdraw: _noWithdraw,
1575          noApprove: _noApprove,
1576          noIncreaseUnlockTime: _noIncreaseUnlockTime,
1577          noMerge: _noMerge
1578        });
1579
1580
1581        // Transfer to borrower
1582        _transferFrom(owner, _to, _tokenId, msg.sender);
1583
1584
1585        if (!_noPullback) {
1586          // The 'return ticket': Approve for current owner (=lender), so lender will be
```

```
1587          // able to actively pull back the veSTRAT using pullBack()
1588          idToApprovals[_tokenId] = owner;
1589          emit Approval(owner, _to, _tokenId);
1590        }
1591      }
```

Listing 2.12: VotingEscrow.sol

Impact The borrower may withdraw the `NFT` before the lender pulls back it.

Suggestion Add the check to ensure the borrower cannot withdraw the NFT when `_noPullback` is true.

2.2.5 Locked Reward for Killed Gauge

Severity Medium

Status Fixed in `Version 2`

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, in the current implementation, a disabled `Gauge` is still votable, and is included in the calculation of the reward distribution, but not claimable.

```
473    function _updateFor(address _gauge) internal {
474        address _pool = poolForGauge[_gauge];
475        uint256 _supplied = weights[_pool];
476        if (_supplied > 0) {
477          uint _supplyIndex = supplyIndex[_gauge];
478          uint _index = index; // get global index0 for accumulated distro
479          supplyIndex[_gauge] = _index; // update _gauge current position to global position
480          uint _delta = _index - _supplyIndex; // see if there is any difference that need to be
                 accrued
481          if (_delta > 0) {
482            uint _share = (uint(_supplied) * _delta) / 1e18; // add accrued difference for each
                   supplied token
483            if (isAlive[_gauge]) {
484              claimable[_gauge] += _share;
485            }
486          }
487        } else {
488          supplyIndex[_gauge] = index; // new users are set to the default global state
489        }
490      }
```

Listing 2.13: Voter.sol

Impact Users who vote for “killed” `Gauges` will receive no rewards.

Suggestion Restrict users from voting for “killed” `Gauges`.

2.2.6 Lack of Checks for Gauges that Do Not Support Voting

Severity Low

Status Acknowledged

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 `NFT` based on the voting weights set by the user for `Gauges`. If a `Gauge` has not been registered in the protocol, the function will skip it, resulting in the votes of this portion 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.

```
176   function _vote(
177       uint _tokenId,
178       address[] memory _poolVote,
179       uint256[] memory _weights
180     ) internal {
181       _reset(_tokenId);
182       uint _poolCnt = _poolVote.length;
183       uint256 _weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);
184       uint256 _totalVoteWeight = 0;
185       uint256 _totalWeight = 0;
186       uint256 _usedWeight = 0;
187
188
189       for (uint i = 0; i < _poolCnt; i++) {
190         _totalVoteWeight += _weights[i];
191       }
192
193
194       for (uint i = 0; i < _poolCnt; i++) {
195         address _pool = _poolVote[i];
196         address _gauge = gauges[_pool];
197
198
199         if (isGauge[_gauge]) {
200           uint256 _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
201           require(votes[_tokenId][_pool] == 0);
202           require(_poolWeight != 0);
203           _updateFor(_gauge);
204
205
206           poolVote[_tokenId].push(_pool);
207
208
209           weights[_pool] += _poolWeight;
210           votes[_tokenId][_pool] += _poolWeight;
211           IBribe(internal_bribes[_gauge])._deposit(
212             uint256(_poolWeight),
213             _tokenId
214           );
```

```
215           IBribe(external_bribes[_gauge])._deposit(
216             uint256(_poolWeight),
217             _tokenId
218           );
219           _usedWeight += _poolWeight;
220           _totalWeight += _poolWeight;
221           emit Voted(msg.sender, _tokenId, _poolWeight);
222         }
223       }
224       if (_usedWeight > 0) IVotingEscrow(_ve).voting(_tokenId);
225       totalWeight += uint256(_totalWeight);
226       usedWeights[_tokenId] = uint256(_usedWeight);
227     }
```

Listing 2.14: Voter.sol

Impact User's voting power can be wasted.

Suggestion Restrict users from voting for unregistered `Gauge`.

Feedback from the Project Team will set up a front-end that has only valid gauges.

2.2.7 Timely Invocation of distribute() in notifyRewardAmount()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the design, the distribution of `Stratum` among various `Gauges` is determined by the voting results of users 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.

```
542   function distribute(address _gauge) public lock {
543       IMinter(minter).update_period();
544       _updateFor(_gauge); // should set claimable to 0 if killed
545       uint _claimable = claimable[_gauge];
546       if (_claimable > IGauge(_gauge).left(base) && _claimable / DURATION > 0) {
547         claimable[_gauge] = 0;
548         if (is3poolGauge[_gauge]) {
549           IGauge(_gauge).notifyRewardAmount(base, _claimable, true);
550         } else {
551           IGauge(_gauge).notifyRewardAmount(base, _claimable, false);
552         }
553         emit DistributeReward(msg.sender, _gauge, _claimable);
554       }
555     }
```

Listing 2.15: Voter.sol

```
444    function notifyRewardAmount(uint amount) external {
445        _safeTransferFrom(base, msg.sender, address(this), amount); // transfer the distro in
446        uint256 _ratio = (amount * 1e18) / totalWeight; // 1e18 adjustment is removed during claim
447        if (_ratio > 0) {
448          index += _ratio;
449        }
450        emit NotifyReward(msg.sender, base, amount);
451      }
```

Listing 2.16: 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.

2.2.8 Lack of Maximum Cap on Total Supply

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Stratum`, the function `mint()` serves the purpose of minting new tokens. However, the minting process does not implement the check for an upper limit on the total supply of tokens. This allows the privileged role to mint unlimited tokens.

```
94     function mint(address account, uint amount) external returns (bool) {
95         require(msg.sender == minter);
96         _mint(account, amount);
97         return true;
98       }
```

Listing 2.17: Stratum.sol

```
100    function claim(address account, uint amount) external returns (bool) {
101        require(msg.sender == redemptionReceiver || msg.sender == merkleClaim);
102        _mint(account, amount);
103        return true;
104      }
```

Listing 2.18: Stratum.sol

Impact The `Stratum` tokens can be minted unlimited with no capped.

Suggestion Establish a reasonable maximum cap on the quantity of tokens and implement checks during the minting process.

2.2.9 Missed Claim Fees for 3pool

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `notifyRewardAmount()` of the contract `Gauge`, it will claim fees for contracts that are not `3pool` by invoking the function `_claimFees()`. However, for `3pool` contracts, claiming fees can only be done manually by invoking the function `claimFeesFor3Pool()`.

```
742   function notifyRewardAmount(
743       address token,
744       uint amount,
745       bool _is3pool
746     ) external lock {
747       require(msg.sender == voter, "!allowed");
748       if (!_is3pool) {
749         _claimFees();
750       }
751       require(token != stake);
752       require(amount > 0);
753       if (!isReward[token]) {
754         require(
755           IVoter(voter).isWhitelisted(token),
756           "rewards tokens must be whitelisted"
757         );
758         require(rewards.length < MAX_REWARD_TOKENS, "too many rewards tokens");
759       }
760       if (rewardRate[token] == 0)
761         _writeRewardPerTokenCheckpoint(token, 0, block.timestamp);
762       (
763         rewardPerTokenStored[token],
764         lastUpdateTime[token]
765       ) = _updateRewardPerToken(token, type(uint).max, true);
766
767
768       if (block.timestamp >= periodFinish[token]) {
769         _safeTransferFrom(token, msg.sender, address(this), amount);
770         rewardRate[token] = amount / DURATION;
771       } else {
772         uint _remaining = periodFinish[token] - block.timestamp;
773         uint _left = _remaining * rewardRate[token];
774         require(amount > _left);
775         _safeTransferFrom(token, msg.sender, address(this), amount);
776         rewardRate[token] = (amount + _left) / DURATION;
777       }
778       require(rewardRate[token] > 0);
779       uint balance = IERC20(token).balanceOf(address(this));
780       require(
781         rewardRate[token] <= balance / DURATION,
782         "Provided reward too high"
783       );
784       periodFinish[token] = block.timestamp + DURATION;
785       if (!isReward[token]) {
786         isReward[token] = true;
787         rewards.push(token);
```

```
788         }
789
790
791         emit NotifyReward(msg.sender, token, amount);
792       }
```

Listing 2.19: Gauge.sol

Impact The mechanism for charging fees is inconsistent.

Suggestion Implement the corresponding logic to invoke the function `claimFeesFor3Pool()` for `3pool` in the function `notifyRewardAmount()`.

2.2.10 Lack of Access Control in Function removePartnerToken()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MetaBribe`, the function `removePartToken()` lacks access control, allowing anyone to remove the partner token through this function.

```
138     function removePartnerToken(address _partner, uint _tokenId) external {
139         for (uint i = 0; i < partnerToTokenIds[_partner].length; i++) {
140           if (partnerToTokenIds[_partner][i] == _tokenId) {
141             partnerToTokenIds[_partner][i] = 0;
142             return;
143           }
144         }
145         revert("not found");
146       }
```

Listing 2.20: MetaBribe.sol

Impact This function allows anyone to remove the partner token.

Suggestion Add access control to the function.

2.2.11 Manipulated bribes_value with Spot Price

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MetaBribe`, the function `check_total_bribes_value_in()` and `check_user_bribes_value_in()` is used to compute the bribe value of the `NFT` for the corresponding epoch owned by the user. When `_tokenAddresses` is not equivalent to `_currency`, it determines the associated value via the function `getAmountOut()` in the contract `Router`. This function calculates the value based on the amount of reserves in the `Pair`, which can be manipulated easily with the help of `flashloan`. As a result, the malicious user can inflate their bribe value artificially to profit.

```
313   function check_user_bribes_value_in(
314       uint tokenId,
315       uint _ts,
316       address _currency
317     ) public view returns (uint) {
318       if (!isPartner(IVotingEscrow(voting_escrow).ownerOf(tokenId))) {
319         return 0;
320       }
321       uint bribes_value = 0;
322       uint pools_len = voter.length();
323       for (uint i = 0; i < pools_len; i++) {
324         address _wxBribe = getWrappedExternalBribeByPool(i);
325         (address[] memory _tokenAddresses, uint[] memory _amounts, , ) = IWrappedExternalBribe(
326           _wxBribe
327         ).getMetaBribe(tokenId, _ts);
328
329
330         for (uint j = 0; j < _tokenAddresses.length; j++) {
331           if (_amounts[j] > 0) {
332             if (_tokenAddresses[j] == _currency) {
333               bribes_value += _amounts[j];
334             } else {
335               (uint value,) = router.getAmountOut(_amounts[j], _tokenAddresses[j], _currency);
336               bribes_value += value;
337             }
338           }
339         }
340       }
341       return bribes_value;
342     }
```

Listing 2.21: MetaBribe.sol

```
349   function check_total_bribes_value_in(address _currency) public view returns (uint) {
350       uint bribes_value = 0;
351       uint pools_len = voter.length();
352       for (uint i = 0; i < pools_len; i++) {
353         address wxBribe = getWrappedExternalBribeByPool(i);
354         uint rewardsListLength = IWrappedExternalBribe(wxBribe)
355           .rewardsListLength();
356         for (uint j = 0; j < rewardsListLength; j++) {
357           address reward = IWrappedExternalBribe(wxBribe).getRewardByIndex(j);
358           uint token_balance = IERC20(reward).balanceOf(wxBribe);
359           if (token_balance > 0) {
360             if (reward == _currency) {
361               bribes_value += token_balance;
362             } else {
363               (uint value, ) = router.getAmountOut(token_balance, reward, _currency);
364               bribes_value += value;
365             }
366           }
367         }
368       }
```

```
369         return bribes_value;
370       }
```

Listing 2.22: MetaBribe.sol

Impact Malicious users can manipulate the return value of the function `getAmountOut()` to inflate their own `bribes_value`.

Suggestion Calculate `bribes_value` with a reliable price source.

2.2.12 Improper Calculation of Total Bribes Value

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MetaBribe`, the function `get_metabribe_weight()` calculates the weight of `Gauge` by determining the ratio of the bribe value in one epoch to the total bribe value. For the calculation of the total bribe value, the function `check_total_bribes_value()` will iterate each `wxBribe` to get `token_balance`, and accumulate the value calculated in a specific `currency`. However, the `token_balance` is obtained directly through the function `IERC20(reward).balanceOf(wxBribe)`, which represents the amount of reward tokens in the current block instead of the value recorded in the corresponding epoch.

```
349     function check_total_bribes_value_in(address _currency) public view returns (uint) {
350         uint bribes_value = 0;
351         uint pools_len = voter.length();
352         for (uint i = 0; i < pools_len; i++) {
353           address wxBribe = getWrappedExternalBribeByPool(i);
354           uint rewardsListLength = IWrappedExternalBribe(wxBribe)
355             .rewardsListLength();
356           for (uint j = 0; j < rewardsListLength; j++) {
357             address reward = IWrappedExternalBribe(wxBribe).getRewardByIndex(j);
358             uint token_balance = IERC20(reward).balanceOf(wxBribe);
359             if (token_balance > 0) {
360               if (reward == _currency) {
361                 bribes_value += token_balance;
362               } else {
363                 (uint value, ) = router.getAmountOut(token_balance, reward, _currency);
364                 bribes_value += value;
365               }
366             }
367           }
368         }
369         return bribes_value;
370       }
```

Listing 2.23: MetaBribe.sol

```
385     function get_metabribe_weight(
386         uint _tokenId,
387         address _gauge,
388         uint week_cursor
```

```
389     ) public view returns (uint) {
390       uint user_bribes_value = check_user_bribes_value(_tokenId, week_cursor);
391       uint total_bribes_value = check_total_bribes_value();
392
393
394       address pool = voter.poolForGauge(_gauge);
395       uint pool_votes = voter.weights(pool);
396       uint partner_votes = check_partner_votes();
397       uint votesByNFTAndPool = voter.votesByNFTAndPool(_tokenId, pool);
398       if (
399         total_bribes_value > 0 &&
400         pool_votes > 0 &&
401         partner_votes > 0 &&
402         votesByNFTAndPool > 0
403       ) {
404         uint weight = ((2 * user_bribes_value * 1000) / total_bribes_value) +
405           ((pool_votes * 1000) / partner_votes);
406         return weight;
407       } else return 0;
408     }
```

Listing 2.24: MetaBribe.sol

```
430   function _claim(
431       uint _tokenId,
432       address ve,
433       uint _last_token_time,
434       address _gauge
435     ) internal returns (uint) {
436       uint user_epoch = 0;
437       uint rebase = 0;
438
439
440       uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
441       uint _start_time = start_time;
442
443
444       if (max_user_epoch == 0) return 0;
445
446
447       uint week_cursor = time_cursor_of[_tokenId];
448       if (week_cursor == 0) {
449         user_epoch = _find_timestamp_user_epoch(
450           ve,
451           _tokenId,
452           _start_time,
453           max_user_epoch
454         );
455       } else {
456         user_epoch = user_epoch_of[_tokenId];
457       }
458
459
```

```
460       if (user_epoch == 0) user_epoch = 1;
461
462
463       IVotingEscrow.Point memory user_point = IVotingEscrow(ve)
464         .user_point_history(_tokenId, user_epoch);
465
466
467       if (week_cursor == 0)
468         week_cursor = ((user_point.ts + WEEK - 1) / WEEK) * WEEK;
469       if (week_cursor >= last_token_time) return 0;
470       if (week_cursor < _start_time) week_cursor = _start_time;
471
472
473       IVotingEscrow.Point memory old_user_point;
474
475
476       for (uint i = 0; i < 50; i++) {
477         if (week_cursor >= _last_token_time) break;
478
479
480         if (week_cursor >= user_point.ts && user_epoch <= max_user_epoch) {
481           user_epoch += 1;
482           old_user_point = user_point;
483           if (user_epoch > max_user_epoch) {
484             user_point = IVotingEscrow.Point(0, 0, 0, 0);
485           } else {
486             user_point = IVotingEscrow(ve).user_point_history(
487               _tokenId,
488               user_epoch
489             );
490           }
491         } else {
492           // metabribe logic
493           uint weight = get_metabribe_weight(_tokenId, _gauge, week_cursor);
494           uint totalWeight = get_metabribe_total_weight(week_cursor);
495
496
497           rebase =
498             (((weight * 1000) / totalWeight) * tokens_per_week[week_cursor]) /
499             1000;
500
501
502           week_cursor += WEEK;
503         }
504       }
505
506
507       user_epoch = Math.min(max_user_epoch, user_epoch - 1);
508       user_epoch_of[_tokenId] = user_epoch;
509       time_cursor_of[_tokenId] = week_cursor;
510
511
512       // emit Claimed(_tokenId, to_distribute, user_epoch, max_user_epoch);
```

```
513
514
515       return rebase;
516     }
```

Listing 2.25: MetaBribe.sol

Impact Weight calculation in function `get_metabribe_weight()` is inaccurate.

Suggestion Correctly calculate `token_balance` in the function `check_total_bribes_value()`.

2.2.13 Incorrect Calculation of the Claimable Amount

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MetaBribe`, the function `_claim()` and `_claimable()` are used to calculate the amount of rewards for a specific `_tokenId` owned by the user over multiple past epochs. However, the current implementation only returns the amount of rewards in the last epoch.

```
430   function _claim(
431       uint _tokenId,
432       address ve,
433       uint _last_token_time,
434       address _gauge
435     ) internal returns (uint) {
436       uint user_epoch = 0;
437       uint rebase = 0;
438
439
440       uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
441       uint _start_time = start_time;
442
443
444       if (max_user_epoch == 0) return 0;
445
446
447       uint week_cursor = time_cursor_of[_tokenId];
448       if (week_cursor == 0) {
449         user_epoch = _find_timestamp_user_epoch(
450           ve,
451           _tokenId,
452           _start_time,
453           max_user_epoch
454         );
455       } else {
456         user_epoch = user_epoch_of[_tokenId];
457       }
458
459
460       if (user_epoch == 0) user_epoch = 1;
461
```

```
462
463         IVotingEscrow.Point memory user_point = IVotingEscrow(ve)
464           .user_point_history(_tokenId, user_epoch);
465
466
467         if (week_cursor == 0)
468           week_cursor = ((user_point.ts + WEEK - 1) / WEEK) * WEEK;
469         if (week_cursor >= last_token_time) return 0;
470         if (week_cursor < _start_time) week_cursor = _start_time;
471
472
473         IVotingEscrow.Point memory old_user_point;
474
475
476         for (uint i = 0; i < 50; i++) {
477           if (week_cursor >= _last_token_time) break;
478
479
480           if (week_cursor >= user_point.ts && user_epoch <= max_user_epoch) {
481             user_epoch += 1;
482             old_user_point = user_point;
483             if (user_epoch > max_user_epoch) {
484               user_point = IVotingEscrow.Point(0, 0, 0, 0);
485             } else {
486               user_point = IVotingEscrow(ve).user_point_history(
487                 _tokenId,
488                 user_epoch
489               );
490             }
491           } else {
492             // metabribe logic
493             uint weight = get_metabribe_weight(_tokenId, _gauge, week_cursor);
494             uint totalWeight = get_metabribe_total_weight(week_cursor);
495
496
497             rebase =
498               (((weight * 1000) / totalWeight) * tokens_per_week[week_cursor]) /
499               1000;
500
501
502             week_cursor += WEEK;
503           }
504         }
505
506
507         user_epoch = Math.min(max_user_epoch, user_epoch - 1);
508         user_epoch_of[_tokenId] = user_epoch;
509         time_cursor_of[_tokenId] = week_cursor;
510
511
512         // emit Claimed(_tokenId, to_distribute, user_epoch, max_user_epoch);
513
514
```

```
515       return rebase;
516     }
```

Listing 2.26: MetaBribe.sol

```
385   function get_metabribe_weight(
386       uint _tokenId,
387       address _gauge,
388       uint week_cursor
389     ) public view returns (uint) {
390       uint user_bribes_value = check_user_bribes_value(_tokenId, week_cursor);
391       uint total_bribes_value = check_total_bribes_value();
392
393
394       address pool = voter.poolForGauge(_gauge);
395       uint pool_votes = voter.weights(pool);
396       uint partner_votes = check_partner_votes();
397       uint votesByNFTAndPool = voter.votesByNFTAndPool(_tokenId, pool);
398       if (
399         total_bribes_value > 0 &&
400         pool_votes > 0 &&
401         partner_votes > 0 &&
402         votesByNFTAndPool > 0
403       ) {
404         uint weight = ((2 * user_bribes_value * 1000) / total_bribes_value) +
405           ((pool_votes * 1000) / partner_votes);
406         return weight;
407       } else return 0;
408     }
```

Listing 2.27: MetaBribe.sol

```
504   function _claimable(
505       uint _tokenId,
506       address ve,
507       uint _last_token_time,
508       address _gauge
509     ) internal view returns (uint) {
510       uint user_epoch = 0;
511       // uint to_distribute = 0;
512       uint rebase = 0;
513
514
515       uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
516       uint _start_time = start_time;
517
518
519       if (max_user_epoch == 0) return 0;
520
521
522       uint week_cursor = time_cursor_of[_tokenId];
523       if (week_cursor == 0) {
524         user_epoch = _find_timestamp_user_epoch(
```

```
525           ve,
526           _tokenId,
527           _start_time,
528           max_user_epoch
529         );
530       } else {
531         user_epoch = user_epoch_of[_tokenId];
532       }
533
534
535       if (user_epoch == 0) user_epoch = 1;
536
537
538       IVotingEscrow.Point memory user_point = IVotingEscrow(ve)
539         .user_point_history(_tokenId, user_epoch);
540
541
542       if (week_cursor == 0)
543         week_cursor = ((user_point.ts + WEEK - 1) / WEEK) * WEEK;
544       if (week_cursor >= last_token_time) return 0;
545       if (week_cursor < _start_time) week_cursor = _start_time;
546
547
548       IVotingEscrow.Point memory old_user_point;
549
550
551       for (uint i = 0; i < 50; i++) {
552         if (week_cursor >= _last_token_time) break;
553
554
555         if (week_cursor >= user_point.ts && user_epoch <= max_user_epoch) {
556           user_epoch += 1;
557           old_user_point = user_point;
558           if (user_epoch > max_user_epoch) {
559             user_point = IVotingEscrow.Point(0, 0, 0, 0);
560           } else {
561             user_point = IVotingEscrow(ve).user_point_history(
562               _tokenId,
563               user_epoch
564             );
565           }
566         } else {
567           // metabribe logic
568           uint weight = get_metabribe_weight(_tokenId, _gauge, week_cursor);
569           uint totalWeight = get_metabribe_total_weight(week_cursor);
570
571
572           rebase =
573             (((weight * 1000) / totalWeight) * tokens_per_week[week_cursor]) /
574             1000;
575
576
577           week_cursor += WEEK;
```

```
578         }
579       }
580       return rebase;
581     }
```

Listing 2.28: MetaBribe.sol

Impact Users will receive less rewards than expected.

Suggestion Accumulate the amount of `rebase` in each iteration.

2.2.14 Delayed Epoch Rewards

Severity Medium

Status Acknowledged

Introduced by `Version 1`

Description In the contract `MetaBribe`, the function `_claim()` and `_claimable()` calculate the amount of rewards the user should receive over past epochs. The variable `week_cursor` rounds up to the next epoch following the epoch in which the user has claimed rewards, while `_last_token_time` rounds down to the latest epoch in which the reward can be claimed. Meanwhile, the check (i.e., `if (week_cursor >= _last_token_time) break;`) in the loop indicates that when `week_cursor` equals to `_last_token_time`, the reward calculation will stop. In this case, the reward for the epoch closest to `_last_token_time` (i.e., `tokens_per_week[week_cursor]`) cannot be claimed and users have to wait one more epoch to claim the reward.

```
430   function _claim(
431       uint _tokenId,
432       address ve,
433       uint _last_token_time,
434       address _gauge
435     ) internal returns (uint) {
436       uint user_epoch = 0;
437       uint rebase = 0;
438
439
440       uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
441       uint _start_time = start_time;
442
443
444       if (max_user_epoch == 0) return 0;
445
446
447       uint week_cursor = time_cursor_of[_tokenId];
448       if (week_cursor == 0) {
449         user_epoch = _find_timestamp_user_epoch(
450           ve,
451           _tokenId,
452           _start_time,
453           max_user_epoch
454         );
455       } else {
```

```
456         user_epoch = user_epoch_of[_tokenId];
457       }
458
459
460       if (user_epoch == 0) user_epoch = 1;
461
462
463       IVotingEscrow.Point memory user_point = IVotingEscrow(ve)
464         .user_point_history(_tokenId, user_epoch);
465
466
467       if (week_cursor == 0)
468         week_cursor = ((user_point.ts + WEEK - 1) / WEEK) * WEEK;
469       if (week_cursor >= last_token_time) return 0;
470       if (week_cursor < _start_time) week_cursor = _start_time;
471
472
473       IVotingEscrow.Point memory old_user_point;
474
475
476       for (uint i = 0; i < 50; i++) {
477         if (week_cursor >= _last_token_time) break;
478
479
480         if (week_cursor >= user_point.ts && user_epoch <= max_user_epoch) {
481           user_epoch += 1;
482           old_user_point = user_point;
483           if (user_epoch > max_user_epoch) {
484             user_point = IVotingEscrow.Point(0, 0, 0, 0);
485           } else {
486             user_point = IVotingEscrow(ve).user_point_history(
487               _tokenId,
488               user_epoch
489             );
490           }
491         } else {
492           // metabribe logic
493           uint weight = get_metabribe_weight(_tokenId, _gauge, week_cursor);
494           uint totalWeight = get_metabribe_total_weight(week_cursor);
495
496
497           rebase =
498             (((weight * 1000) / totalWeight) * tokens_per_week[week_cursor]) /
499             1000;
500
501
502           week_cursor += WEEK;
503         }
504       }
505
506
507       user_epoch = Math.min(max_user_epoch, user_epoch - 1);
508       user_epoch_of[_tokenId] = user_epoch;
```

```
509       time_cursor_of[_tokenId] = week_cursor;
510
511
512       // emit Claimed(_tokenId, to_distribute, user_epoch, max_user_epoch);
513
514
515       return rebase;
516     }
```

Listing 2.29: MetaBribe.sol

```
504   function _claimable(
505       uint _tokenId,
506       address ve,
507       uint _last_token_time,
508       address _gauge
509     ) internal view returns (uint) {
510       uint user_epoch = 0;
511       // uint to_distribute = 0;
512       uint rebase = 0;
513
514
515       uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
516       uint _start_time = start_time;
517
518
519       if (max_user_epoch == 0) return 0;
520
521
522       uint week_cursor = time_cursor_of[_tokenId];
523       if (week_cursor == 0) {
524         user_epoch = _find_timestamp_user_epoch(
525           ve,
526           _tokenId,
527           _start_time,
528           max_user_epoch
529         );
530       } else {
531         user_epoch = user_epoch_of[_tokenId];
532       }
533
534
535       if (user_epoch == 0) user_epoch = 1;
536
537
538       IVotingEscrow.Point memory user_point = IVotingEscrow(ve)
539         .user_point_history(_tokenId, user_epoch);
540
541
542       if (week_cursor == 0)
543         week_cursor = ((user_point.ts + WEEK - 1) / WEEK) * WEEK;
544       if (week_cursor >= last_token_time) return 0;
545       if (week_cursor < _start_time) week_cursor = _start_time;
```

```
546
547
548       IVotingEscrow.Point memory old_user_point;
549
550
551       for (uint i = 0; i < 50; i++) {
552         if (week_cursor >= _last_token_time) break;
553
554
555         if (week_cursor >= user_point.ts && user_epoch <= max_user_epoch) {
556           user_epoch += 1;
557           old_user_point = user_point;
558           if (user_epoch > max_user_epoch) {
559             user_point = IVotingEscrow.Point(0, 0, 0, 0);
560           } else {
561             user_point = IVotingEscrow(ve).user_point_history(
562               _tokenId,
563               user_epoch
564             );
565           }
566         } else {
567           // metabribe logic
568           uint weight = get_metabribe_weight(_tokenId, _gauge, week_cursor);
569           uint totalWeight = get_metabribe_total_weight(week_cursor);
570
571
572           rebase =
573             (((weight * 1000) / totalWeight) * tokens_per_week[week_cursor]) /
574             1000;
575
576
577           week_cursor += WEEK;
578         }
579       }
580       return rebase;
581     }
```

Listing 2.30: MetaBribe.sol

Impact Users will not be able to claim rewards from the previous epoch.

Suggestion Implement corresponding logic to ensure that users can claim rewards for the last epoch.

2.2.15 Locked Reward for the Expired NFT

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `claim()` and `claim_many()` of the contract `MetaBribe`, claimed rewards will be deposited into the corresponding `NFT` automatically. However, in the function `deposit_for()` of the contract `VotingEscrow`, tokens are not allowed to be deposited into the expired `NFT`. In this case, unless the user increases the unlock time of the expired `NFT`, or the user may not be able to claim the rewards.

```
581    function claim(uint _tokenId, address _gauge) external returns (uint) {
582        require(voter.isGauge(_gauge) == true);
583        require(isPartner(msg.sender) == true, "not a partner");
584        if (block.timestamp >= time_cursor) _checkpoint_total_supply();
585        uint _last_token_time = last_token_time;
586        _last_token_time = (_last_token_time / WEEK) * WEEK;
587        uint amount = _claim(_tokenId, voting_escrow, _last_token_time, _gauge);
588        if (amount != 0) {
589            IVotingEscrow(voting_escrow).deposit_for(_tokenId, amount);
590            token_last_balance -= amount;
591        }
592        return amount;
593    }
```

Listing 2.31: MetaBribe.sol

```
595    function claim_many(
596        uint[] memory _tokenIds,
597        address[] memory _gauges
598      ) external returns (bool) {
599        if (block.timestamp >= time_cursor) _checkpoint_total_supply();
600        uint _last_token_time = last_token_time;
601        _last_token_time = (_last_token_time / WEEK) * WEEK;
602        address _voting_escrow = voting_escrow;
603        uint total = 0;
604
605
606        for (uint i = 0; i < _tokenIds.length; i++) {
607          uint _tokenId = _tokenIds[i];
608          address _gauge = _gauges[i];
609          if (_tokenId == 0) break;
610          uint amount = _claim(_tokenId, _voting_escrow, _last_token_time, _gauge);
611          if (amount != 0) {
612            IVotingEscrow(_voting_escrow).deposit_for(_tokenId, amount);
613            total += amount;
614          }
615        }
616        if (total != 0) {
617          token_last_balance -= total;
618        }
619
620
621        return true;
622      }
```

Listing 2.32: MetaBribe.sol

Impact Users will not be able to claim rewards if their `NFTs` have expired.

Suggestion Distribute rewards directly to the user if their `NFTs` expire.

2.2.16 Incorrect Calculation of Reward in earned()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `ExternalBribe`, the function `earned()` calculates the balance and supply within each epoch to get the amount of rewards. However, in the current implementation, the time frame used for calculation is `_bribeStart(currsTs) + DURATION`, which actually includes the first second of the next epoch. In this case, deposits made by users in the first second of the new epoch will also be considered in the previous epoch, which is incorrect. The same problem exists in the contracts `WrappedExternalBribe` and `InternalBribe`.

```
246   function earned(address token, uint tokenId) public view returns (uint) {
247       if (numCheckpoints[tokenId] == 0) {
248         return 0;
249       }
250
251
252       uint reward = 0;
253       uint _ts = 0;
254       uint _bal = 0;
255       uint _supply = 1;
256       uint _index = 0;
257       uint _currTs = _bribeStart(lastEarn[token][tokenId]); // take epoch last claimed in as
              starting point
258
259
260       _index = getPriorBalanceIndex(tokenId, _currTs);
261       _ts = checkpoints[tokenId][_index].timestamp;
262       _bal = checkpoints[tokenId][_index].balanceOf;
263       // accounts for case where lastEarn is before first checkpoint
264       _currTs = Math.max(_currTs, _bribeStart(_ts));
265
266
267       // get epochs between current epoch and first checkpoint in same epoch as last claim
268       uint numEpochs = (_bribeStart(block.timestamp) - _currTs) / DURATION;
269
270
271       if (numEpochs > 0) {
272         for (uint256 i = 0; i < numEpochs; i++) {
273           // get index of last checkpoint in this epoch
274           _index = getPriorBalanceIndex(tokenId, _currTs + DURATION);
275           // get checkpoint in this epoch
276           _ts = checkpoints[tokenId][_index].timestamp;
277           _bal = checkpoints[tokenId][_index].balanceOf;
278           // get supply of last checkpoint in this epoch
279           _supply = supplyCheckpoints[getPriorSupplyIndex(_currTs + DURATION)]
280             .supply;
281           if (_supply > 0)
282             // prevent div by 0
283             reward += (_bal * tokenRewardsPerEpoch[token][_currTs]) / _supply;
```

```
284             _currTs += DURATION;
285           }
286         }
287
288
289         return reward;
290       }
```

Listing 2.33: ExternalBribe.sol

Impact Rewards are miscalculated.

Suggestion The time frame of an epoch should be changed to `_currTs + DURATION - 1`.

2.2.17 Timely Update Rewards in Function killGauge()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract of `Voter`, the function `killgauge()` sets the claimable rewards of the Gauge to zero directly without updating and distributing the rewards to users.

```
390     function killGauge(address _gauge) external {
391         require(msg.sender == emergencyCouncil, "not emergency council");
392               require(isAlive[_gauge], "gauge already dead");
393                isAlive[_gauge] = false;
394                claimable[_gauge] = 0;
395                emit GaugeKilled(_gauge);
396           }
```

Listing 2.34: Voter.sol

Impact Users receive fewer rewards than expected.

Suggestion Update the claimable rewards before killing the `Gauge`, which allows the user to claim their rewards even if the `Gauge` is not alive.

2.2.18 Potential DoS in check_total_bribes_value()

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the contract `MetaBribe`, the function `check_total_bribes_value()` calculates the values of all `bribes` by iterating through all the `NFTs` in the contract `VotingEscrow`. This could potentially lead to a Denial of Service, as a new `NFT` can be minted by locking only 1 wei in the contract `VotingEscrow`. It may cost a large amount of gas to iterate all of them.

```
330     function check_total_bribes_value(uint _ts) public view returns (uint) {
331         uint bribes_value = 0;
332         uint veNFTAmount = IVotingEscrow(voting_escrow).tokenId();
```

```
333       for (uint i = 0; i < veNFTAmount; i++) {
334         uint pools_len = voter.length();
335         for (uint j = 0; j < pools_len; j++) {
336           address _wxBribe = getWrappedExternalBribeByPool(j);
337           (, , uint[] memory values, , ) = IWrappedExternalBribe(_wxBribe)
338             .getMetaBribe(i, _ts);
339           for (uint k = 0; k < values.length; k++) {
340             if (values[k] > 0) {
341               bribes_value += values[k];
342             }
343           }
344         }
345       }
346       return bribes_value;
347     }
```

Listing 2.35: MetaBribe.sol

Impact Users can not claim rewards if the function `check_total_bribes_value()` reverts due to running out of gas.

Suggestion Optimize the gas usage for the function `check_total_bribes_value()`.

2.2.19 Incorrect Calculation of the Bribe Value

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the contract `MetaBribe`, the function `estimateValue()` calculates the bribe value using the function `current()`. However, while assessing the bribe value for `intermediaryAmount` using `transitCurrencies[i]` tokens, the incorrect parameter `tokenOut` was used. The correct parameter to pass should be `transitCurrencies[i]`.

```
726   function estimateValue(
727   address tokenIn,
728   uint amountIn,
729   address tokenOut
730 ) external view returns (uint) {
731   if (tokenIn == tokenOut || amountIn == 0) {
732     return amountIn;
733   }
734
735
736   uint bestAmountOut = 0;
737   uint bestLiquidity;
738   address pair;
739
740
741   // direct route
742   (pair, bestLiquidity) = getMostLiquidPair(tokenIn, tokenOut);
743   if (pair != address(0)) {
```

```
744       bestAmountOut = IPair(pair).current(tokenIn, amountIn);
745       bestLiquidity = bestLiquidity * bestLiquidity;
746     }
747
748
749     // routes with one hop via intermediate token
750     for (uint i = 0; i < transitCurrencies.length; i++) {
751
752
753       uint intermediaryAmount = 0;
754       (address pair0, uint liquidity0) = getMostLiquidPair(tokenIn, transitCurrencies[i]);
755       if (pair0 == address(0)) {
756         continue;
757       }
758       intermediaryAmount = IPair(pair0).current(tokenIn, amountIn);
759
760
761       (address pair1, uint liquidity1) = getMostLiquidPair(transitCurrencies[i], tokenOut);
762       if (pair1 == address(0)) {
763         continue;
764       }
765
766
767       if (liquidity0 * liquidity1 > bestLiquidity) {
768         bestLiquidity = liquidity0 * liquidity1;
769         bestAmountOut = IPair(pair1).current(tokenOut, intermediaryAmount);
770       }
771
772
773     }
774
775
776     return bestAmountOut;
777   }
```

Listing 2.36: MetaBribe.sol

Impact The user's bribe value is calculated incorrectly.

Suggestion Change `current(tokenOut, intermediaryAmount)` to `current(transitCurrencies[i], intermediaryAmount)`.

2.2.20 The Return Value of the Function estimateValue() can be Manipulated

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the contract `MetaBribe`, the function `estimateValue()` estimates the value of `tokenIn` by calculating the `tokenOut` amount based on the input of `tokenIn` and `amountIn`. The function introduces `transitCurrencies` on the whitelist to act as intermediate tokens similar to routes in a `DEX` protocol to calculate the `bestAmountOut`. Ultimately, which swap path to select is determined by comparing the product of

liquidity including the `tokenIn/tokenOut` pair and several intermediary pairs, where a higher product value represents deeper liquidity pools.

However, the liquidity can actually be manipulated by `flashloan`, malicious users may make the output results favor their own interests.

```
726   function estimateValue(
727   address tokenIn,
728   uint amountIn,
729   address tokenOut
730 ) external view returns (uint) {
731   if (tokenIn == tokenOut || amountIn == 0) {
732     return amountIn;
733   }
734
735
736   uint bestAmountOut = 0;
737   uint bestLiquidity;
738   address pair;
739
740
741   // direct route
742   (pair, bestLiquidity) = getMostLiquidPair(tokenIn, tokenOut);
743   if (pair != address(0)) {
744     bestAmountOut = IPair(pair).current(tokenIn, amountIn);
745     bestLiquidity = bestLiquidity * bestLiquidity;
746   }
747
748
749   // routes with one hop via intermediate token
750   for (uint i = 0; i < transitCurrencies.length; i++) {
751
752
753     uint intermediaryAmount = 0;
754     (address pair0, uint liquidity0) = getMostLiquidPair(tokenIn, transitCurrencies[i]);
755     if (pair0 == address(0)) {
756       continue;
757     }
758     intermediaryAmount = IPair(pair0).current(tokenIn, amountIn);
759
760
761     (address pair1, uint liquidity1) = getMostLiquidPair(transitCurrencies[i], tokenOut);
762     if (pair1 == address(0)) {
763       continue;
764     }
765
766
767     if (liquidity0 * liquidity1 > bestLiquidity) {
768       bestLiquidity = liquidity0 * liquidity1;
769       bestAmountOut = IPair(pair1).current(tokenOut, intermediaryAmount);
770     }
771
```

```
772
773     }
774
775
776     return bestAmountOut;
777   }
```

Listing 2.37: MetaBribe.sol

Impact This calculation mechanism allows users to leverage `flashloan` to make the results in a way that benefits themselves.

Suggestion Compare the `bestAmountOut` that can be obtained from each route, and select the maximum or minimum value, rather than comparing the pool liquidity.

2.2.21 Incorrect Calculation of the MetaBribe_Weight

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the contract `MetaBribe`, the function `get_metabribe_weight_info()` calculates both the user's bribe weight and the total bribe weight on a specified `tokenId` and `timestamp`. According to the design, the `user_weight` is calculated through a formula of ''`alpha*first_term + beta*second_term`'' (line 420). In the `first_term`, the denominator is the sum of the bribe value from all pools. However, during the calculation process, the current pool's bribe value was continuously added with each `partner_tokens_ids` iteration, which is incorrect.

```
412     function get_metabribe_weight_info(
413     uint _tokenId,
414     uint _ts
415   ) public view returns (uint user_weight, uint total_weight) {
416
417
418     //
419     // see https://stratum-exchange.gitbook.io/stratum-exchange/meta-bribes
420     //
421     // user_weight = weight(_tokenId) = alpha*first_term + beta*second_term
422     //
423     // total_weight = sum of weight(partner veNFT) over all partner veNFTs
424     //
425     // first_term =
426     //  (sum over all pools: bribes value of pool from _tokenId)
427     //  / (sum over all pools: total bribes value of pool)
428     //
429     // second_term =
430     //   sum over all pools of: [
431     //      (bribes value for that pool from _tokenId) * (total votes for that pool)
432     //      / ( (total bribes for that pool) * (votingPower of _tokenId) )
433     //   ]
434     //
435
```

```
436
437   // in memory due to EVM stack size limits
438   WeightsFormulaTerms memory terms;
439
440
441   // memorize temporary lookup tables
442   uint[] memory partner_token_ids = get_partner_token_ids();
443   uint[] memory partner_voting_power = new uint[](partner_token_ids.length);
444   for (uint i = 0; i < partner_token_ids.length; i++) {
445     partner_voting_power[i] = ve_for_at(partner_token_ids[i], _ts);
446   }
447
448
449   // for each pool
450   for (uint i = 0; i < voter.length(); i++) {
451     address wxBribe = getWrappedExternalBribeByPool(i);
452     if (wxBribe == address(0)) {
453       continue; // pools without gauge can't have bribes
454     }
455
456
457     uint total_bribes_value_of_pool = IWrappedExternalBribe(wxBribe).getTotalBribesValue(_ts);
458     uint all_votes_for_pool = votesCheckpointPerEpoch[_ts].votesPerPoolIndex[i];
459
460
461     // for each partner
462     for (uint j = 0; j < partner_token_ids.length; j++) {
463
464
465       uint partner_bribes_value_of_pool = IWrappedExternalBribe(wxBribe).getPartnerBribesValue(
              _ts, partner_token_ids[j]);
466
467
468       // first term
469       terms.first_term_nominator_all += partner_bribes_value_of_pool;
470       if (partner_token_ids[j] == _tokenId) {
471         terms.first_term_nominator_tkn += partner_bribes_value_of_pool;
472       }
473       terms.first_term_denominator += total_bribes_value_of_pool;
474
475
476       // second term (only if bribes occurred, to prevent division by zero)
477       if (partner_bribes_value_of_pool > 0 && total_bribes_value_of_pool > 0 &&
              partner_voting_power[j] > 0) {
478
479
480         uint second_term_summand =
481           (partner_bribes_value_of_pool * all_votes_for_pool * 1e18)
482           / (total_bribes_value_of_pool * partner_voting_power[j]);
483
484
485         terms.second_term_all += second_term_summand;
486         if (partner_token_ids[j] == _tokenId) {
```

```
487             terms.second_term_tkn += second_term_summand;
488           }
489
490
491         }
492
493
494       }
495
496
497     }
498
499
500     user_weight =
501       (alpha * terms.first_term_nominator_tkn * 1e18)
502       / terms.first_term_denominator + beta * terms.second_term_tkn;
503     total_weight =
504       (alpha * terms.first_term_nominator_all * 1e18)
505       / terms.first_term_denominator + beta * terms.second_term_all;
506   }
```

Listing 2.38: MetaBribe.sol

Impact The function `get_metabribe_weight_info()` returns incorrect values for both `user_weight` and `total_weight`.

Suggestion The bribe value should be accumulated only when iterating through the pools, rather than being accumulated again when entering the iteration of `partner_tokens_ids`.

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 `setGovernor()` and `constructor()` of the contract `Voter`.

```
83    constructor(
84        address __ve,
85        address _factory,
86        address _gauges,
87        address _bribes
88      ) {
89        _ve = __ve;
90        factory = _factory;
91        base = IVotingEscrow(__ve).token();
92        gaugefactory = _gauges;
93        bribefactory = _bribes;
94        minter = msg.sender;
95        governor = msg.sender;
```

```
96        emergencyCouncil = msg.sender;
97      }
```

Listing 2.39: Voter.sol

```
125   function setGovernor(address _governor) public {
126       require(msg.sender == governor);
127       governor = _governor;
128     }
```

Listing 2.40: Voter.sol

Suggestion I 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()`.

```
557   function distro() external {
558       distribute(0, pools.length);
559     }
```

Listing 2.41: Voter.sol

```
561   function distribute() external {
562       distribute(0, pools.length);
563     }
```

Listing 2.42: Voter.sol

Suggestion I Remove the redundant function.

2.3.3 Lack of Check for _swapAddress

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `claimFeesFor3Pool()` of the contract `Gauge`, users can feed any addresses due to the lack of validation for the parameter `_swapAddress`.

```
175   function claimFeesFor3Pool(
176       address _swapAddress
177     ) external lock returns (uint claimed0, uint claimed1, uint claimed2) {
178       return _claimFeesFor3Pool(_swapAddress);
179     }
```

Listing 2.43: Gauge.sol

Suggestion I Add the check for `_swapaddress` accordingly.

2.3.4 Lack of Check for _transferFrom

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `_transferFrom()` of the contract `VotingEscrow`, if the parameter `from` is equal to `to`, the transfer function is meaningless, and this situation should be prevented.

```
307   function _transferFrom(
308       address _from,
309       address _to,
310       uint _tokenId,
311       address _sender
312     ) internal {
313       require(
314         _restrictions_permit_transfer(_tokenId, _to),
315         "restrictions: receiver not lender"
316       );
317       require(attachments[_tokenId] == 0 && !voted[_tokenId], "attached");
318       // Check requirements
319       require(_isApprovedOrOwner(_sender, _tokenId));
320       // Clear approval. Throws if `_from` is not the current owner
321       _clearApproval(_from, _tokenId);
322       // Remove NFT. Throws if `_tokenId` is not a valid NFT
323       _removeTokenFrom(_from, _tokenId);
324       // auto re-delegate
325       _moveTokenDelegates(delegates(_from), delegates(_to), _tokenId);
326       // Add NFT
327       _addTokenTo(_to, _tokenId);
328       // Set the block of ownership transfer (for Flash NFT protection)
329       ownership_change[_tokenId] = block.number;
330       // Log the transfer
331       emit Transfer(_from, _to, _tokenId);
332     }
```

Listing 2.44: VotingEscrow.sol

Suggestion I Add the check to ensure that the parameter ''`from`'' is not equal to ''`to`''.

2.3.5 Incorrect Error Message

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `_claimFees()` of the contract `Gauge`, the `require` statement limits that the contract should not be the type of `3pool`. However, the error message is not correct. It should indicate that this function is invoked in the ''`3 pool`'' instead of ''`Not a 3 pool`''.

```
142   function _claimFees() internal returns (uint claimed0, uint claimed1) {
143       require(IVoter(voter).is3poolGauge(address(this)) == false, "Not a 3pool");
144       if (!isForPair) {
145         return (0, 0);
```

```
146       }
147       (claimed0, claimed1) = IPair(stake).claimFees();
148       if (claimed0 > 0 || claimed1 > 0) {
149         uint _fees0 = fees0 + claimed0;
150         uint _fees1 = fees1 + claimed1;
151         (address _token0, address _token1) = IPair(stake).tokens();
152         if (
153           _fees0 > IBribe(internal_bribe).left(_token0) && _fees0 / DURATION > 0
154         ) {
155           fees0 = 0;
156           _safeApprove(_token0, internal_bribe, _fees0);
157           IBribe(internal_bribe).notifyRewardAmount(_token0, _fees0);
158         } else {
159           fees0 = _fees0;
160         }
161         if (
162           _fees1 > IBribe(internal_bribe).left(_token1) && _fees1 / DURATION > 0
163         ) {
164           fees1 = 0;
165           _safeApprove(_token1, internal_bribe, _fees1);
166           IBribe(internal_bribe).notifyRewardAmount(_token1, _fees1);
167         } else {
168           fees1 = _fees1;
169         }
170
171
172         emit ClaimFees(msg.sender, claimed0, claimed1);
173       }
174     }
```

Listing 2.45: Gauge.sol

Suggestion Use the message `"This is 3 pool"`.

2.4 Notes

2.4.1 Potential Centralization Problem

Introduced by `version 1`

Description This project has potential centralization problems. The privileged role `minter` has the ability to mint large number of tokens(This risk arises if the privileged role `minter` isn't assigned to the contract `Minter`). The privileged role `gov` can change the whitelist token, which changes the reward token users receive in the contract `ExternalBribe` and `InternalBribe`. We suggest these roles should be in multi-signature.

2.4.2 Incompatible Tokens

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.

2.4.3 Timely Pull Back Borrowed NFT for Lenders

Introduced by `version 1`

Description For users who lend their `NFTs` to borrowers and enable the functionality of pulling back, they should be cautious of the expiration time. If they fail to pull back their `NFTs` after expiration via the function `pullBack()`, there is a risk of the borrower revoking the ability of lenders to pull back them.

2.4.4 Invocation of Function poke() in One Epoch

Introduced by `version 1`

Description Within each epoch, the `owner` or an authorized account of a `NFT` can repeatedly invoke the function `poke()` to refresh the `NFT's` voting power.