

## REPORT 61F78CF61A6439001AE31664

Created Mon Jan 31 2022 07:17:10 GMT+0000 (Coordinated Universal Time)

Number of analyses 1

User 61f52e351fd393a0c51a34fe

# **REPORT SUMMARY**

Analyses ID Main source file Detected vulnerabilities

9b22fc97-9708-4dea-a076-7c2467d1ad7a

gauges.sol

3

Started Mon Jan 31 2022 07:17:21 GMT+0000 (Coordinated Universal Time)

Finished Mon Jan 31 2022 08:02:30 GMT+0000 (Coordinated Universal Time)

Mode Deep

Client Tool Remythx

Main Source File Gauges. Sol

### **DETECTED VULNERABILITIES**

(HIGH (MEDIUM (LOW

0 1 2

#### **ISSUES**

### MEDIUM Multiple calls are executed in the same transaction.

SWC-113

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source file gauges.sol Locations

```
require(rewardRate[token] > 0);

uint balance = erc20 token balanceOf address this ;

require(rewardRate[token] <= balance / DURATION, "Provided reward too high");

periodFinish[token] = block.timestamp + DURATION;
```

LOW Read of persistent state following external call.

SWC-107

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy look can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file gauges.sol Locations

```
rewardRate[token] = (amount + _left) / DURATION;

}

frequire(rewardRate:token > 0);

uint balance = erc20(token).balanceOf(address(this));

require(rewardRate[token] <= balance / DURATION, "Provided reward too high");
```

LOW Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file gauges.sol

Locations

```
require(token.code.length > 0);

(bool success, bytes memory data) =

token.call abi encodeWithSelector erc20 transferFrom selector from to value);

require(success &6 (data.length == 0 || abi.decode(data, (bool))));

}
```

```
Source file gauges.sol
```

Locations

```
// Gauges are used to incentivize pools, they emit reward tokens over 7 days for staked LP tokens
47
     contract Gauge {
49
     address public immutable stake; // the LP token that needs to be staked for rewards
50
     address public immutable _ve; // the ve token used for gauges
51
52
     address public immutable bribe;
     address public immutable voter;
53
54
     uint public derivedSupply;
     mapping(address => uint) public derivedBalances;
56
57
     uint internal constant DURATION = 7 days; // rewards are released over 7 days
58
     uint internal constant PRECISION = 10 ** 18:
59
60
     // default snx staking contract implementation
61
     mapping(address => uint) public rewardRate;
62
     mapping(address => uint) public periodFinish;
63
     mapping(address => uint) public rewardPerTokenStored;
65
66
     mapping(address => mapping(address => uint)) public lastEarn;
67
     mapping(address => mapping(address => uint)) public userRewardPerTokenStored;
     mapping(address => mapping(address => uint)) public userRewards;
69
70
     mapping(address => uint) public tokenIds;
71
72
     uint public totalSupply;
73
     mapping(address => uint) public balanceOf;
74
     address[] public rewards;
76
     mapping(address => bool) public isReward;
77
78
    event Deposit(address indexed from, uint tokenId, uint amount);
event Withdraw(address indexed from, uint tokenId, uint amount)
event NotifyReward(address indexed from, address indexed reward, uint amount);
79
80
81
     event ClaimFees(address indexed from, uint claimed0, uint claimed1);
                 nRewards(address indexed from, address indexed reward, uint amount);
83
84
     function claimFees() external returns (uint claimed0, uint claimed1) {
85
     claimed0 claimed1 = IBaseV1Core(stake).claimFees();
address _token0 address _token1 = IBaseV1Core(stake) tokens();
86
87
        afeApprove(_token0, bribe, claimed0);
88
            pprove(_token1, bribe, claimed1);
     IBribe(bribe).notifyRewardAmount(_token0, claimed0);
```

```
Bribe(bribe).notifyRewardAmount(_token1, claimed1);
92
93
     emit ClaimFees(msg.sender, claimed0, claimed1);
94
95
96
     /// @notice A checkpoint for marking balance
97
     struct Checkpoint {
98
99
100
     uint balanceOf;
101
102
     /// Onotice A checkpoint for marking reward rate
struct RewardPerTokenCheckpoint {
103
105
     uint timestamp;
106
     uint rewardPerToken;
107
108
     /// @notice A checkpoint for marking supply
struct SupplyCheckpoint {
110
111
     uint timestamp;
112
     uint supply;
113
114
     /// @notice A record of balance checkpoints for each account, by index
115
     mapping (address => mapping (uint => Checkpoint)) public checkpoints;
116
118
     /// @notice The number of checkpoints for each account
     mapping (address => uint) public numCheckpoints;
119
120
121
     /// @notice A record of balance checkpoints for each token, by index
     mapping (uint => SupplyCheckpoint) public supplyCheckpoints;
123
124
     /// @notice The number of checkpoints
125
     uint public supplyNumCheckpoints;
126
127
     /// @notice A record of balance checkpoints for each token, by index
128
     mapping (address => mapping (uint => RewardPerTokenCheckpoint)) public rewardPerTokenCheckpoints;
129
130
     /// @notice The number of checkpoints for each token
     mapping (address => uint) public rewardPerTokenNumCheckpoints;
131
     // simple re-entrancy check
133
134
     uint internal _unlocked = 1;
135
136
     require(_unlocked == 1);
137
     _unlocked = 2;
138
139
     _unlocked = 1;
140
141
142
     constructor(address _stake, address _bribe, address __ve, address _voter) {
143
     stake = _stake;
144
     bribe = _bribe;
146
     voter = _voter;
147
148
149
     * @notice Determine the prior balance for an account as of a block number

* @dev Block number must be a finalized block or else this function will revert to prevent misinformation.
150
151
     * @param account The address of the account to check
152
     * @param timestamp The timestamp to get the balance at
```

```
ereturn The balance the account had as of the given block
 155
 156
             function getPriorBalanceIndex(address account, uint timestamp) public view returns (uint) {
 157
             uint nCheckpoints = numCheckpoints[account];
 158
             if (nCheckpoints == 0) {
            return 0;
 160
 161
 162
 163
            if (checkpoints[account][nCheckpoints - 1] timestamp <= timestamp) {</pre>
 164
            return (nCheckpoints - 1);
 165
 166
 167
             // Next check implicit zero balance
            if (checkpoints[account][0].timestamp > timestamp) {
 169
 170
 171
 172
            uint lower = 0;
 173
            uint upper = nCheckpoints - 1;
 174
            while (upper > lower) {
 175
            uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
 176
             Checkpoint memory cp = checkpoints[account][center];
            if (cp.timestamp == timestamp) {
 178
            return center;
 179
            } else if (cp.timestamp < timestamp) {
 180
            lower = center;
 181
 182
            upper = center - 1;
 183
 184
 185
            return lower;
 187
 188
            function \ getPriorSupplyIndex(uint \ \textbf{timestamp}) \ public \ view \ returns \ (uint) \ function \ function \ getPriorSupplyIndex(uint \ \textbf{timestamp}) \ public \ view \ returns \ (uint) \ function \ functio
 189
            uint nCheckpoints = supplyNumCheckpoints:
 190
            if (nCheckpoints == 0) {
 191
            return 0;
 192
 193
 194
             // First check most recent balance
             if (supplyCheckpoints[nCheckpoints - 1].timestamp <= timestamp) {</pre>
 196
            return (nCheckpoints - 1);
 197
 198
 199
             // Next check implicit zero balance
 200
            if (supplyCheckpoints[0].timestamp > timestamp) {
 201
            return 0;
 202
 203
 204
            uint lower = 0;
 205
            uint upper = nCheckpoints - 1;
 206
            while (upper > lower) {
 207
            uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
 208
            SupplyCheckpoint memory cp = supplyCheckpoints[center];
            if (cp.timestamp == timestamp) {
210
            return center;
 211
            } else if (cp.timestamp < timestamp) {</pre>
            lower = center;
212
 213
            } else {
 214
            upper = center - 1;
 215
```

```
return lower;
218
220
     function getPriorRewardPerToken(address token, uint timestamp) public view returns (uint, uint) {
      uint nCheckpoints = rewardPerTokenNumCheckpoints[token];
     if (nCheckpoints == 0) {
     return (0,0);
224
225
226
227
     if (rewardPerTokenCheckpoints[token][nCheckpoints - 1] timestamp <= timestamp) {</pre>
228
     return rewardPerTokenCheckpoints token nCheckpoints - 1 rewardPerToken rewardPerTokenCheckpoints token nCheckpoints - 1 timestamp
229
230
231
      // Next check implicit zero balance
     if (rewardPerTokenCheckpoints[token][0].timestamp > timestamp) {
233
     return (0,0);
234
235
236
     uint lower = 0;
237
     uint upper = nCheckpoints - 1;
238
     while (upper > lower) {
239
     uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
240
     RewardPerTokenCheckpoint memory cp = rewardPerTokenCheckpoints[token][center];
241
     if (cp.timestamp == timestamp) {
242
     return (cp.rewardPerToken, cp.timestamp);
243
     } else if (cp.timestamp < timestamp) {</pre>
244
     lower = center:
245
     } else {
246
     upper = center - 1;
247
248
     return (rewardPerTokenCheckpoints[token][lower].rewardPerToken, rewardPerTokenCheckpoints[token][lower] timestamp);
250
252
     function _writeCheckpoint(address account, uint balance) internal {
253
         nt _timestamp = block timestamp;
254
     uint _nCheckPoints = numCheckpoints[account];
255
256
     if (_nCheckPoints > 0 88 checkpoints[account][_nCheckPoints - 1].timestamp == _timestamp) {
257
     checkpoints[account][_nCheckPoints - 1].balanceOf = balance;
258
259
     checkpoints[account][_nCheckPoints] = Checkpoint(_timestamp, balance);
260
     numCheckpoints[account] = _nCheckPoints + 1;
261
262
263
264
     function _writeRewardPerTokenCheckpoint(address token, uint reward, uint timestamp) internal {
265
     uint _nCheckPoints = rewardPerTokenNumCheckpoints[token];
266
267
     if (_nCheckPoints > 0 88 rewardPerTokenCheckpoints[token][_nCheckPoints - 1].timestamp == timestamp) {
268
     rewardPerTokenCheckpoints[token][_nCheckPoints - 1].rewardPerToken = reward
269
270
     rewardPerTokenCheckpoints[token][_nCheckPoints] = RewardPerTokenCheckpoint(timestamp reward);
     rewardPerTokenNumCheckpoints[token] = _nCheckPoints + 1;
273
275
276
     uint _nCheckPoints = supplyNumCheckpoints;
277
     uint _timestamp = block timestamp;
278
     if (_nCheckPoints > 0 88 supplyCheckpoints[_nCheckPoints - 1] timestamp == _timestamp) {
```

```
supplyCheckpoints[_nCheckPoints - 1].supply = derivedSupply:
281
282
          supplyCheckpoints[_nCheckPoints] = SupplyCheckpoint(_timestamp, derivedSupply);
283
          supplyNumCheckpoints = _nCheckPoints + 1;
284
285
286
287
          function rewardsListLength() external view returns (uint) {
288
         return rewards.length;
289
290
291
          // returns the last time the reward was modified or periodFinish if the reward has ended
292
          function lastTimeRewardApplicable(address token) public view returns (uint) {
293
          return Math.min(block.timestamp, periodFinish[token]);
294
295
         296
297
          [rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRewardPerToken(token);
298
          [userRewards[token][account], lastEarn[token][account]) = _batchUserRewards[token, account, maxRuns];
299
300
301
          function getReward(address account, address[] memory tokens) external lock {
302
          require(msg.sender == account || msg.sender == voter);
303
          for (uint i = 0; i < tokens.length; i++) {
          [rewardPerTokenStored[tokens[i]], \ lastUpdateTime[tokens[i]]) = \\ \_updateRewardPerToken(tokens[i]); \\ [substitution of the last updateTime[tokens[i]]) = \\ [substitution of the last updateTime[tokens[i]]) = \\ [substitution of tokens[i]], \\ [subst
305
306
          uint _reward = earned(tokens[i], account);
307
          userRewards[tokens[i]][account] = 0;
308
          lastEarn[tokens[i]][account] = block.timestamp;
309
          userRewardPerTokenStored[tokens[i]][account] = rewardPerTokenStored[tokens[i]];
310
          if (_reward > 0) _safeTransfer(tokens[i], account, _reward);
311
          emit ClaimRewards(msg.sender, tokens[i], _reward);
313
314
315
          uint _derivedBalance = derivedBalances[account];
316
          derivedSupply -= _derivedBalance;
          _derivedBalance = derivedBalance(account);
318
          derivedBalances[account] = _derivedBalance/
319
          derivedSupply += _derivedBalance;
320
321
          writeCheckpoint(account, derivedBalances(account]);
writeSupplyCheckpoint();
323
324
325
326
          function rewardPerToken(address token) public view returns (uint) {
327
          if (derivedSupply == 0) {
328
          return rewardPerTokenStored[token];
329
330
          return rewardPerTokenStored token + (lastTimeRewardApplicable(token - Math min lastUpdateTime token), periodFinish token)) * rewardRate token * PRECISION / derivedSupply
331
332
333
         function derivedBalance(address account) public view returns (uint) {
334
          uint _tokenId = tokenIds[account];
335
          uint _balance = balanceOf[account];
336
          uint _derived = _balance * 40 / 100;
337
          uint _adjusted = 0;
338
          if account == ve(_ve).ownerOf(_tokenId) 88 _supply > 0) |
adjusted = ve(_ve _balanceOfNFT(_tokenId);
339
340
341
         _adjusted = (totalSupply * _adjusted / _supply) * 60 / 100:
342
```

```
return Math.min((_derived + _adjusted), _balance);
344
345
346
     function _batchUserRewards(address token, address account, uint maxRuns) internal view returns (uint, uint) {
      uint _startTimestamp = lastEarn[token][account];
348
     if (numCheckpoints[account] == 0) {
     return (userRewards[token][account], _startTimestamp);
350
351
352
     uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
353
     uint _endIndex = Math.min(numCheckpoints[account]-1, maxRuns);
354
355
     uint reward = userRewards[token][account];
356
     for (uint i = _startIndex; i < _endIndex; i++)</pre>
357
     Checkpoint memory cp0 = checkpoints[account][i];
     Checkpoint memory cp1 = checkpoints[account][i+1];
     uint _rewardPerTokenStored0.) = getPriorRewardPerToken.token. cp0 timestamp;;
uint _rewardPerTokenStored1.) = getPriorRewardPerToken.token. cp1 timestamp;;
359
360
361
     reward += cp0.balanceOf * (_rewardPerTokenStored1 - _rewardPerTokenStored0) / PRECISION:
362
     _startTimestamp = cp1.timestamp;
363
364
365
     return (reward, _startTimestamp);
366
367
368
     function batchRewardPerToken(address token, uint maxRuns) external {
369
     (rewardPerTokenStored[token], lastUpdateTime[token]) = _batchRewardPerToken(token, maxRuns);
370
372
     function _batchRewardPerToken(address token, uint maxRuns) internal returns (uint, uint) {
     uint _startTimestamp = lastUpdateTime[token];
374
      uint reward = rewardPerTokenStored[token];
376
     if (supplyNumCheckpoints == 0) {
     return (reward, _startTimestamp);
378
379
380
     uint _startIndex = getPriorSupplyIndex(_startTimestamp);
381
     uint _endIndex = Math.min(supplyNumCheckpoints-1, maxRuns);
382
383
     for (uint i = _startIndex; i < _endIndex; i++) {</pre>
384
     SupplyCheckpoint memory sp0 = supplyCheckpoints[i];
385
     if (sp0.supply > 0) {
386
     SupplyCheckpoint memory sp1 = supplyCheckpoints[i+1];
387
         nt _reward, uint _endTime) = _calcRewardPerToken(token, sp1 timestamp sp0 timestamp, sp0 supply, _startTimestamp);
388
389
      _writeRewardPerTokenCheckpoint(token, reward, _endTime);
390
     _startTimestamp = _endTime;
391
392
393
394
     return (reward, _startTimestamp);
395
396
397
     function_calcRewardPerToken(address token_ uint timestamp1, uint timestamp0, uint supply, uint startTimestamp internal view returns (uint, uint)
398
      uint endTime = Math.max(timestamp1, startTimestamp);
399
     return (((Math miniendTime, periodFinish token)) - Math min(Math max timestamp0 startTimestamp), periodFinish token)) * rewardRate token * PRECISION / supply, endTime
400
401
402
     function _updateRewardPerToken(address token) internal returns (uint, uint) {
403
     uint _startTimestamp = lastUpdateTime[token];
404
     uint reward = rewardPerTokenStored[token];
405
```

```
if (supplyNumCheckpoints == 0) {
407
      return (reward, _startTimestamp);
400
      uint _startIndex = getPriorSupplyIndex(_startTimestamp);
411
      uint _endIndex = supplyNumCheckpoints-1;
412
413
      if (_endIndex - _startIndex > 1) {
414
          (uint i = _startIndex; i < _endIndex-1; i++) {</pre>
415
      SupplyCheckpoint memory sp0 = supplyCheckpoints[i];
416
      if (sp0.supply > 0) {
417
      SupplyCheckpoint memory sp1 = supplyCheckpoints[i+1];
418
      (uint _reward, uint _endTime) = _calcRewardPerToken(token, sp1.timestamp, sp0.timestamp, sp0.supply, _startTimestamp);
        writeRewardPerTokenCheckpoint(token, reward, _endTime);
420
      _startTimestamp = _endTime;
422
423
424
425
426
      SupplyCheckpoint memory sp = supplyCheckpoints[_endIndex];
427
      if (sp.supply > 0) {
428
      (uint _reward,) = _calcRewardPerToken(token, lastTimeRewardApplicable(token), Math.max(sp timestamp, _startTimestamp), sp supply, _startTimestamp);
430
      _writeRewardPerTokenCheckpoint(token, reward, block timestamp);
431
      _startTimestamp = block_timestamp;
432
433
434
      return (reward, _startTimestamp);
435
436
437
      // earned is an estimation, it won't be exact till the supply > rewardPerToken calculations have run
438
      function earned(address token, address account) public view returns (uint) {
439
      uint _startTimestamp = lastEarn[token][account];
440
      if (numCheckpoints[account] == 0) {
441
      return userRewards[token][account];
442
443
444
      uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
445
      uint _endIndex = numCheckpoints[account]-1;
446
447
      uint reward = userRewards[token][account];
448
449
     if (_endIndex - _startIndex > 1) {
450
          (uint i = _startIndex; i < _endIndex-1; i++) {</pre>
451
      Checkpoint memory cp0 = checkpoints[account][i];
452
      Checkpoint memory cp1 = checkpoints[account][i+1];
453
      uint _rewardPerTokenStored0.) = getPriorRewardPerToken(token, cp0 timestamp);
uint _rewardPerTokenStored1.) = getPriorRewardPerToken(token, cp1 timestamp);
454
455
      reward += cp0 balanceOf * (_rewardPerTokenStored1 - _rewardPerTokenStored0) / PRECISION:
456
457
458
459
     Checkpoint memory cp = checkpoints[account][_endIndex];
460
      (uint _rewardPerTokenStored,) = getPriorRewardPerToken(token, cp timestamp);
461
      reward += cp balanceOf * (rewardPerToken(token) - Math max(_rewardPerTokenStored, userRewardPerTokenStored(token)[account])) / PRECISION.
463
      return reward;
464
465
466
      function depositAll(uint tokenId) external {
467
          osit(erc20(stake).balanceOf(msg.sender), tokenId);
```

```
470
     function deposit(uint amount, uint tokenId) public lock {
471
     require(amount > 0);
472
     if (tokenId > 0) {
473
     require(ve(_ve).ownerOf(tokenId) == msg.sender);
474
     tokenIds[msg.sender] = tokenId;
475
476
477
     if (balanceOf[msg.sender] == 0) Voter(voter).attachTokenToGauge(tokenId, msg.sender, amount);
478
479
      _safeTransferFrom(stake, msg.sender, address(this), amount);
480
     totalSupply += amount;
481
     balanceOf[msg.sender] += amount;
482
483
     uint _derivedBalance = derivedBalances[msg.sender];
     derivedSupply -= _derivedBalance;
485
     _derivedBalance = derivedBalance(msg_sender);
486
     derivedBalances[msg sender] = _derivedBalance;
487
     derivedSupply += _derivedBalance;
488
489
     _writeCheckpoint(msg sender, _derivedBalance);
490
      _writeSupplyCheckpoint();
491
492
     emit Deposit(msg sender, tokenId, amount);
493
494
495
496
     withdraw(balanceOf[msg.sender]);
497
499
     function withdraw(uint amount) public lock {
500
     uint tokenId = tokenIds[msg sender];
501
502
     totalSupply -= amount;
503
     balanceOf[msg.sender] -= amount;
504
      _safeTransfer(stake, msg.sender, amount);
505
506
     if (tokenId > 0) tokenIds[msg.sender] = 0;
507
     if (balanceOf[msg_sender] == 0) Voter(voter) detachTokenFromGauge(tokenId, msg_sender, amount)
508
509
     uint _derivedBalance = derivedBalances[msg.sender];
510
     derivedSupply -= _derivedBalance;
     _derivedBalance = derivedBalance(msg sender);
     derivedBalances[msg_sender] = _derivedBalance.
     derivedSupply += _derivedBalance;
514
     _mriteCheckpoint(msg sender derivedBalances(msg sender]);
writeSupplyCheckpoint();
515
516
518
     emit Withdraw(msg.sender, tokenId, amount);
519
520
521
     function left(address token) external view returns (uint) {
522
     if (block.timestamp >= periodFinish[token]) return 0;
523
     uint _remaining = periodFinish[token] - block timestamp
524
     return _remaining * rewardRate[token];
525
526
527
     function notifyRewardAmount(address token, uint amount) external lock {
528
     require(token != stake);
     rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRewardPerToken(token);
529
530
531
     if (block timestamp >= periodFinish[token]) {
```

```
{\sf afeTransferFrom}({\sf token}, \ {\sf msg}, {\sf sender}, \ {\sf address}({\sf this}), \ {\sf amount});
533
      rewardRate[token] = amount / DURATION;
534
      } else {
535
      uint _remaining = periodFinish[token] - block timestamp.
536
      uint _left = _remaining * rewardRate[token];
537
      require(amount > _left);
_safeTransferFrom(token, msg sender, address(this), amount);
538
539
      rewardRate[token] = (amount + _left) / DURATION;
540
541
      require(rewardRate[token] > 0);
542
      uint balance = erc20(token).balanceOf(address(this));
543
      require(rewardRate[token] <= balance / DURATION, "Provided reward too high");</pre>
544
      periodFinish token = block timestamp + DURATION
546
      isReward[token] = true;
      rewards.push(token);
548
549
550
      emit NotifyReward(msg.sender, token, amount);
551
552
      function _safeTransfer(address token, address to, uint256 value) internal {
553
554
      require(token.code.length > 0);
555
      | bool success, bytes memory data) | = | token.call(abi encodeWithSelector erc20 transfer selector, to, value)).
557
      require(success && (data.length == 0 || abi.decode(data, (bool))));
558
559
560
      function _safeTransferFrom(address token, address from, address to, uint256 value) internal {
561
      require(token.code.length > 0);
      | bool success, bytes memory data | = | token.call(abi encodeWithSelector(erc20.transferFrom selector_from_to_value)).
562
563
564
      require(success 88 (data.length == 0 || abi.decode(data, (bool))));
565
566
567
      function _safeApprove(address token, address spender, uint256 value) internal (
568
      require(token.code.length > 0);
569
      (bool success, bytes memory data) =
      token.call(abi.encodeWithSelector(erc20.approve_selector, spender, value));
571
      require(success 88 (data.length == 0 || abi.decode(data, (bool))));
572
573
     contract BaseV1GaugeFactory {
```