MythX

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

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

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 SWC-113

Multiple calls are executed in the same transaction.

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

```
540 | }
541 | require(rewardRate[token] > 0);
542 | uint balance = erc20(token).balanceOf(address(this));
543 | require(rewardRate[token] <= balance / DURATION, "Provided reward too high");
544 | periodFinish[token] = block.timestamp + DURATION;
```

LOW SWC-107

Read of persistent state following external call.

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 lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`gauges.sol`

Locations

```
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");
```

LOW

SWC-123

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).

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 && (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
contract Gauge {

address public immutable stake; // the LP token that needs to be staked for rewards
address public immutable _ve; // the ve token used for gauges
address public immutable bribe;
address public immutable voter;

uint public derivedSupply;
mapping(address => uint) public derivedBalances;

uint internal constant DURATION = 7 days; // rewards are released over 7 days
uint internal constant PRECISION = 10 ** 18;

// default snx staking contract implementation
mapping(address => uint) public rewardRate;
mapping(address => uint) public periodFinish;
mapping(address => uint) public lastUpdateTime;
mapping(address => uint) public rewardPerTokenStored;

mapping(address => mapping(address => uint)) public lastEarn;
mapping(address => mapping(address => uint)) public userRewardPerTokenStored;
mapping(address => mapping(address => uint)) public userRewards;

mapping(address => uint) public tokenIds;

uint public totalSupply;
mapping(address => uint) public balanceOf;

address[] public rewards;
mapping(address => bool) public isReward;

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);
event ClaimFees(address indexed from, uint claimed0, uint claimed1);
event ClaimRewards(address indexed from, address indexed reward, uint amount);

function claimFees() external returns (uint claimed0, uint claimed1) {
(claimed0, claimed1) = IBaseV1Core(stake).claimFees();
(address _token0, address _token1) = IBaseV1Core(stake).tokens();
_safeApprove(_token0, bribe, claimed0);
_safeApprove(_token1, bribe, claimed1);
IBribe(bribe).notifyRewardAmount(_token0, claimed0);
IBribe(bribe).notifyRewardAmount(_token1, claimed1);

emit ClaimFees(msg.sender, claimed0, claimed1);
}


/// @notice A checkpoint for marking balance
struct Checkpoint {
uint timestamp;
uint balanceOf;
}

/// @notice A checkpoint for marking reward rate
struct RewardPerTokenCheckpoint {
uint timestamp;
uint rewardPerToken;
}

/// @notice A checkpoint for marking supply
struct SupplyCheckpoint {
uint timestamp;
uint supply;
}

/// @notice A record of balance checkpoints for each account, by index
mapping (address => mapping (uint => Checkpoint)) public checkpoints;

/// @notice The number of checkpoints for each account
mapping (address => uint) public numCheckpoints;

/// @notice A record of balance checkpoints for each token, by index
mapping (uint => SupplyCheckpoint) public supplyCheckpoints;

/// @notice The number of checkpoints
uint public supplyNumCheckpoints;

/// @notice A record of balance checkpoints for each token, by index
mapping (address => mapping (uint => RewardPerTokenCheckpoint)) public rewardPerTokenCheckpoints;

/// @notice The number of checkpoints for each token
mapping (address => uint) public rewardPerTokenNumCheckpoints;

// simple re-entrancy check
uint internal _unlocked = 1;
modifier lock() {
require(_unlocked == 1);
_unlocked = 2;
_;
_unlocked = 1;
}

constructor(address _stake, address _bribe, address __ve, address _voter) {
stake = _stake;
bribe = _bribe;
_ve = __ve;
voter = _voter;
}

/**
* @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.
* @param account The address of the account to check
* @param timestamp The timestamp to get the balance at
* @return The balance the account had as of the given block
*/
function getPriorBalanceIndex(address account, uint timestamp) public view returns (uint) {
uint nCheckpoints = numCheckpoints[account];
if (nCheckpoints == 0) {
return 0;
}

// First check most recent balance
if (checkpoints[account][nCheckpoints - 1].timestamp <= timestamp) {
return (nCheckpoints - 1);
}

// Next check implicit zero balance
if (checkpoints[account][0].timestamp > timestamp) {
return 0;
}

uint lower = 0;
uint upper = nCheckpoints - 1;
while (upper > lower) {
uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
Checkpoint memory cp = checkpoints[account][center];
if (cp.timestamp == timestamp) {
return center;
} else if (cp.timestamp < timestamp) {
lower = center;
} else {
upper = center - 1;
}
}
return lower;
}

function getPriorSupplyIndex(uint timestamp) public view returns (uint) {
uint nCheckpoints = supplyNumCheckpoints;
if (nCheckpoints == 0) {
return 0;
}

// First check most recent balance
if (supplyCheckpoints[nCheckpoints - 1].timestamp <= timestamp) {
return (nCheckpoints - 1);
}

// Next check implicit zero balance
if (supplyCheckpoints[0].timestamp > timestamp) {
return 0;
}

uint lower = 0;
uint upper = nCheckpoints - 1;
while (upper > lower) {
uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
SupplyCheckpoint memory cp = supplyCheckpoints[center];
if (cp.timestamp == timestamp) {
return center;
} else if (cp.timestamp < timestamp) {
lower = center;
} else {
upper = center - 1;
}
}
```

```
return lower;
}

function getPriorRewardPerToken(address token, uint timestamp) public view returns (uint, uint) {
uint nCheckpoints = rewardPerTokenNumCheckpoints[token];
if (nCheckpoints == 0) {
return (0,0);
}

// First check most recent balance
if (rewardPerTokenCheckpoints[token][nCheckpoints - 1].timestamp <= timestamp) {
return (rewardPerTokenCheckpoints[token][nCheckpoints - 1].rewardPerToken, rewardPerTokenCheckpoints[token][nCheckpoints - 1].timestamp);
}

// Next check implicit zero balance
if (rewardPerTokenCheckpoints[token][0].timestamp > timestamp) {
return (0,0);
}

uint lower = 0;
uint upper = nCheckpoints - 1;
while (upper > lower) {
uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
RewardPerTokenCheckpoint memory cp = rewardPerTokenCheckpoints[token][center];
if (cp.timestamp == timestamp) {
return (cp.rewardPerToken, cp.timestamp);
} else if (cp.timestamp < timestamp) {
lower = center;
} else {
upper = center - 1;
}
}
return (rewardPerTokenCheckpoints[token][lower].rewardPerToken, rewardPerTokenCheckpoints[token][lower].timestamp);
}

function _writeCheckpoint(address account, uint balance) internal {
uint _timestamp = block.timestamp;
uint _nCheckPoints = numCheckpoints[account];

if (_nCheckPoints > 0 && checkpoints[account][_nCheckPoints - 1].timestamp == _timestamp) {
checkpoints[account][_nCheckPoints - 1].balanceOf = balance;
} else {
checkpoints[account][_nCheckPoints] = Checkpoint(_timestamp, balance);
numCheckpoints[account] = _nCheckPoints + 1;
}
}

function _writeRewardPerTokenCheckpoint(address token, uint reward, uint timestamp) internal {
uint _nCheckPoints = rewardPerTokenNumCheckpoints[token];

if (_nCheckPoints > 0 && rewardPerTokenCheckpoints[token][_nCheckPoints - 1].timestamp == timestamp) {
rewardPerTokenCheckpoints[token][_nCheckPoints - 1].rewardPerToken = reward;
} else {
rewardPerTokenCheckpoints[token][_nCheckPoints] = RewardPerTokenCheckpoint(timestamp, reward);
rewardPerTokenNumCheckpoints[token] = _nCheckPoints + 1;
}
}

function _writeSupplyCheckpoint() internal {
uint _nCheckPoints = supplyNumCheckpoints;
uint _timestamp = block.timestamp;

if (_nCheckPoints > 0 && supplyCheckpoints[_nCheckPoints - 1].timestamp == _timestamp) {
```

```
supplyCheckpoints[_nCheckPoints - 1].supply = derivedSupply;
} else {
supplyCheckpoints[_nCheckPoints] = SupplyCheckpoint(_timestamp, derivedSupply);
supplyNumCheckpoints = _nCheckPoints + 1;
}
}

function rewardsListLength() external view returns (uint) {
return rewards.length;
}

// returns the last time the reward was modified or periodFinish if the reward has ended
function lastTimeRewardApplicable(address token) public view returns (uint) {
return Math.min(block.timestamp, periodFinish[token]);
}

function batchUserRewards(address token, address account, uint maxRuns) external {
(rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRewardPerToken(token);
(userRewards[token][account], lastEarn[token][account]) = _batchUserRewards(token, account, maxRuns);
}

function getReward(address account, address[] memory tokens) external lock {
require(msg.sender == account || msg.sender == voter);
for (uint i = 0; i < tokens.length; i++) {
(rewardPerTokenStored[tokens[i]], lastUpdateTime[tokens[i]]) = _updateRewardPerToken(tokens[i]);

uint _reward = earned(tokens[i], account);
userRewards[tokens[i]][account] = 0;
lastEarn[tokens[i]][account] = block.timestamp;
userRewardPerTokenStored[tokens[i]][account] = rewardPerTokenStored[tokens[i]];
if (_reward > 0) _safeTransfer(tokens[i], account, _reward);

emit ClaimRewards(msg.sender, tokens[i], _reward);
}

uint _derivedBalance = derivedBalances[account];
derivedSupply -= _derivedBalance;
_derivedBalance = derivedBalance(account);
derivedBalances[account] = _derivedBalance;
derivedSupply += _derivedBalance;

_writeCheckpoint(account, derivedBalances[account]);
_writeSupplyCheckpoint();
}


function rewardPerToken(address token) public view returns (uint) {
if (derivedSupply == 0) {
return rewardPerTokenStored[token];
}
return rewardPerTokenStored[token] + ((lastTimeRewardApplicable(token) - Math.min(lastUpdateTime[token], periodFinish[token])) * rewardRate[token] * PRECISION / derivedSupply);
}

function derivedBalance(address account) public view returns (uint) {
uint _tokenId = tokenIds[account];
uint _balance = balanceOf[account];
uint _derived = _balance * 40 / 100;
uint _adjusted = 0;
uint _supply = erc20(_ve).totalSupply();
if (account == ve(_ve).ownerOf(_tokenId) && _supply > 0) {
_adjusted = ve(_ve).balanceOfNFT(_tokenId);
_adjusted = (totalSupply * _adjusted / _supply) * 60 / 100;
}
```

```
return Math.min((_derived + _adjusted), _balance);
}

function _batchUserRewards(address token, address account, uint maxRuns) internal view returns (uint, uint) {
uint _startTimestamp = lastEarn[token][account];
if (numCheckpoints[account] == 0) {
return (userRewards[token][account], _startTimestamp);
}

uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
uint _endIndex = Math.min(numCheckpoints[account]-1, maxRuns);

uint reward = userRewards[token][account];
for (uint i = _startIndex; i < _endIndex; i++) {
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);
reward += cp0.balanceOf * (_rewardPerTokenStored1 - _rewardPerTokenStored0) / PRECISION;
_startTimestamp = cp1.timestamp;
}

return (reward, _startTimestamp);
}

function batchRewardPerToken(address token, uint maxRuns) external {
(rewardPerTokenStored[token], lastUpdateTime[token]) = _batchRewardPerToken(token, maxRuns);
}

function _batchRewardPerToken(address token, uint maxRuns) internal returns (uint, uint) {
uint _startTimestamp = lastUpdateTime[token];
uint reward = rewardPerTokenStored[token];

if (supplyNumCheckpoints == 0) {
return (reward, _startTimestamp);
}

uint _startIndex = getPriorSupplyIndex(_startTimestamp);
uint _endIndex = Math.min(supplyNumCheckpoints-1, maxRuns);

for (uint i = _startIndex; i < _endIndex; i++) {
SupplyCheckpoint memory sp0 = supplyCheckpoints[i];
if (sp0.supply > 0) {
SupplyCheckpoint memory sp1 = supplyCheckpoints[i+1];
(uint _reward, uint _endTime) = _calcRewardPerToken(token, sp1.timestamp, sp0.timestamp, sp0.supply, _startTimestamp);
reward += _reward;
_writeRewardPerTokenCheckpoint(token, reward, _endTime);
_startTimestamp = _endTime;
}
}

return (reward, _startTimestamp);
}

function _calcRewardPerToken(address token, uint timestamp1, uint timestamp0, uint supply, uint startTimestamp) internal view returns (uint, uint) {
uint endTime = Math.max(timestamp1, startTimestamp);
return (((Math.min(endTime, periodFinish[token]) - Math.min(Math.max(timestamp0, startTimestamp), periodFinish[token])) * rewardRate[token] * PRECISION / supply), endTime);
}

function _updateRewardPerToken(address token) internal returns (uint, uint) {
uint _startTimestamp = lastUpdateTime[token];
uint reward = rewardPerTokenStored[token];

```

```
if (supplyNumCheckpoints == 0) {
return (reward, _startTimestamp);
}

uint _startIndex = getPriorSupplyIndex(_startTimestamp);
uint _endIndex = supplyNumCheckpoints-1;

if (_endIndex - _startIndex > 1) {
for (uint i = _startIndex; i < _endIndex-1; i++) {
SupplyCheckpoint memory sp0 = supplyCheckpoints[i];
if (sp0.supply > 0) {
SupplyCheckpoint memory sp1 = supplyCheckpoints[i+1];
(uint _reward, uint _endTime) = _calcRewardPerToken(token, sp1.timestamp, sp0.timestamp, sp0.supply, _startTimestamp);
reward += _reward;
_writeRewardPerTokenCheckpoint(token, reward, _endTime);
_startTimestamp = _endTime;
}
}
}

SupplyCheckpoint memory sp = supplyCheckpoints[_endIndex];
if (sp.supply > 0) {
(uint _reward,) = _calcRewardPerToken(token, lastTimeRewardApplicable(token), Math.max(sp.timestamp, _startTimestamp), sp.supply, _startTimestamp);
reward += _reward;
_writeRewardPerTokenCheckpoint(token, reward, block.timestamp);
_startTimestamp = block.timestamp;
}

return (reward, _startTimestamp);
}

// earned is an estimation, it won't be exact till the supply > rewardPerToken calculations have run
function earned(address token, address account) public view returns (uint) {
uint _startTimestamp = lastEarn[token][account];
if (numCheckpoints[account] == 0) {
return userRewards[token][account];
}

uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
uint _endIndex = numCheckpoints[account]-1;

uint reward = userRewards[token][account];

if (_endIndex - _startIndex > 1) {
for (uint i = _startIndex; i < _endIndex-1; i++) {
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);
reward += cp0.balanceOf * (_rewardPerTokenStored1 - _rewardPerTokenStored0) / PRECISION;
}
}

Checkpoint memory cp = checkpoints[account][_endIndex];
(uint _rewardPerTokenStored,) = getPriorRewardPerToken(token, cp.timestamp);
reward += cp.balanceOf * (rewardPerToken(token) - Math.max(_rewardPerTokenStored, userRewardPerTokenStored[token][account])) / PRECISION;

return reward;
}

function depositAll(uint tokenId) external {
deposit(erc20(stake).balanceOf(msg.sender), tokenId);
}
```

```
function deposit(uint amount, uint tokenId) public lock {
require(amount > 0);
if (tokenId > 0) {
require(ve(_ve).ownerOf(tokenId) == msg.sender);
tokenIds[msg.sender] = tokenId;
}

if (balanceOf[msg.sender] == 0) Voter(voter).attachTokenToGauge(tokenId, msg.sender, amount);

_safeTransferFrom(stake, msg.sender, address(this), amount);
totalSupply += amount;
balanceOf[msg.sender] += amount;

uint _derivedBalance = derivedBalances[msg.sender];
derivedSupply -= _derivedBalance;
_derivedBalance = derivedBalance(msg.sender);
derivedBalances[msg.sender] = _derivedBalance;
derivedSupply += _derivedBalance;

_writeCheckpoint(msg.sender, _derivedBalance);
_writeSupplyCheckpoint();

emit Deposit(msg.sender, tokenId, amount);
}

function withdrawAll() external {
withdraw(balanceOf[msg.sender]);
}

function withdraw(uint amount) public lock {
uint tokenId = tokenIds[msg.sender];

totalSupply -= amount;
balanceOf[msg.sender] -= amount;
_safeTransfer(stake, msg.sender, amount);

if (tokenId > 0) tokenIds[msg.sender] = 0;
if (balanceOf[msg.sender] == 0) Voter(voter).detachTokenFromGauge(tokenId, msg.sender, amount);

uint _derivedBalance = derivedBalances[msg.sender];
derivedSupply -= _derivedBalance;
_derivedBalance = derivedBalance(msg.sender);
derivedBalances[msg.sender] = _derivedBalance;
derivedSupply += _derivedBalance;

_writeCheckpoint(msg.sender, derivedBalances[msg.sender]);
_writeSupplyCheckpoint();

emit Withdraw(msg.sender, tokenId, amount);
}

function left(address token) external view returns (uint) {
if (block.timestamp >= periodFinish[token]) return 0;
uint _remaining = periodFinish[token] - block.timestamp;
return _remaining * rewardRate[token];
}

function notifyRewardAmount(address token, uint amount) external lock {
require(token != stake);
(rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRewardPerToken(token);

if (block.timestamp >= periodFinish[token]) {
```

```
_safeTransferFrom(token, msg.sender, address(this), amount);
rewardRate[token] = amount / DURATION;
} else {
uint _remaining = periodFinish[token] - block.timestamp;
uint _left = _remaining * rewardRate[token];
require(amount > _left);
_safeTransferFrom(token, msg.sender, address(this), amount);
rewardRate[token] = (amount + _left) / DURATION;
}
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;
if (!isReward[token]) {
isReward[token] = true;
rewards.push(token);
}

emit NotifyReward(msg.sender, token, amount);
}

function _safeTransfer(address token, address to, uint256 value) internal {
require(token.code.length > 0);
(bool success, bytes memory data) =
token.call(abi.encodeWithSelector(erc20.transfer.selector, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))));
}

function _safeTransferFrom(address token, address from, address to, uint256 value) internal {
require(token.code.length > 0);
(bool success, bytes memory data) =
token.call(abi.encodeWithSelector(erc20.transferFrom.selector, from, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))));
}

function _safeApprove(address token, address spender, uint256 value) internal {
require(token.code.length > 0);
(bool success, bytes memory data) =
token.call(abi.encodeWithSelector(erc20.approve.selector, spender, value));
require(success && (data.length == 0 || abi.decode(data, (bool))));
}
}

contract BaseV1GaugeFactory {
```