SpartaDEX Lockdrop Airdrop Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

SpartaDEX is a combination of real-time strategy game set in the realities of ancient Greece and a decentralized cryptocurrency exchange. They call it a **gamified DEX**. The main goal is to provide the exchange with user engagement known from video games, which builds loyalty and consistency in providing liquidity.

SpartaDEX Lockdrop is their strategy of launching the DEX with the goal to kick-start the TVL (liquidity) by incentivizing those who will transfer and lock their LP in SpartaDEX.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	SpartaDEX Lockdrop Airdrop
Repository	sdex-smart-contracts
Commit	47bf1157f87d...
Date	May 2023

Issues Found

Severity	Count
Critical Risk	4
High Risk	5
Medium Risk	3
Low Risk	6
Informational	0
Gas Optimizations	0
Total Issues	**18**

Summary of Findings

Title	Status
[C-1] A malicious user can drain all the funds from the TokenVesting contract	Resolved
[C-2] getLPTokenPrice can be manipulated through a flash loan	Resolved
[C-3] An adversary can lock user funds by spamming vesting schedules	Resolved
[C-4] Excessive user allocations can DoS the calculateTotalReward function and deprive the user from rewards	Resolved
[H-1] Unclaimed tokens are locked forever in the Airdrop.sol contract	Resolved
[H-2] addTargetLiquidity and removeSourceLiquidity has no slippage protection, which can result in in loss of funds	Resolved
[H-3] LockdropPhase2 lacks token recovery mechanism	Resolved
[H-4] ExchangeTokens lacks slippage control	Resolved
[H-5] AddLiquidity and removeLiquidity is called without expiration	Resolved
[M-1] Potential mismatch between claimable amounts and actual token balance	Resolved
[M-2] Potential manipulation of airdrop claimable amounts in setClaimableAmounts function	Resolved
[M-3] Duplicate locking expiration timestamps can dillute the rewards	Resolved

Summary of Findings

Title	Status
[L-1] Missing sanity checks for claimStartTimestamp in Airdrop contract	Resolved
[L-2] Possibility for division by zero in LockdropPhase1	Resolved
[L-3] LockdropPhase2 is prone to reentrancy	Resolved
[L-4] Violation of the Check-Effects-Interaction pattern in getRewardAndSendOnVesting	Resolved
[L-5] Potential for Cross-Chain Replay attacks due to hard-coded Domain Separator	Resolved
[L-6] Inconsistency in LiquidityProvided event parameters	Resolved

5 Findings

5.1 Critical Risk

[C-1] A malicious user can drain all the funds from the TokenVesting contract

Context: TokenVesting.sol

Impact: This vulnerability allows a malicious actor to drain the entire TokenVesting contract of its tokens prematurely. The issue arises from the lack of a proper check for whether the vesting schedule duration has elapsed.

Description: The claim function in the TokenVesting contract calculates the amount of claimable tokens based on the elapsed time since the start of the vesting schedule. The function assumes the vesting schedule is not over without validating this assumption, allowing users to claim more tokens than they should be entitled to at any point in time.

When the claim function is called, it calculates the vested amount based on the elapsed time and the total vesting duration. However, it does not check whether the elapsed time has exceeded the vesting duration. This oversight allows a user to claim tokens that should not yet be available for withdrawal, leading to the potential theft of all tokens in the contract.

The problematic code block is as follows:

```
// @audit-issue there's no check if the vesting schedule is over
uint256 elapsedTime = block.timestamp - vestingSchedule.startTime;
uint256 vestingDuration = vestingSchedule.endTime - vestingSchedule.startTime;
uint256 vestedAmount = (vestingSchedule.totalAmount * elapsedTime) / vestingDuration;

uint256 unclaimedAmount = vestedAmount - vestingSchedule.claimedAmount;
```

The provided proof-of-concept demonstrates how an attacker could exploit this vulnerability to claim all tokens in the contract:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "forge-std/console.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface ITokenVesting {
    error InvalidScheduleID();
    error VestingNotStarted();
    error AllTokensClaimed();
    error ZeroTokens();
    error ZeroDuration();
    error TransferFailed();
    error NothingToClaim();

    event VestingAdded(
        address indexed beneficiary,
        uint256 indexed allocationId,
        uint256 startTime,
        uint256 endTime,
        uint256 amount
    );

    event TokenWithdrawn(
        address indexed beneficiary,
        uint256 indexed allocationId,
        uint256 value
    );
```

```
    struct Vesting {
        uint256 startTime;
        uint256 endTime;
        uint256 totalAmount;
        uint256 claimedAmount;
    }

    function addVesting(
        address beneficiary,
        uint256 startTime,
        uint256 duration,
        uint256 amount
    ) external;

    function claim(uint256[] calldata scheduleIds) external;
}


contract TokenVesting is ITokenVesting {
    IERC20 public immutable token;
    mapping(address => Vesting[]) public vestingSchedules;

    constructor(IERC20 _token) {
        token = _token;
    }

    function addVesting(
        address beneficiary,
        uint256 startTime,
        uint256 duration,
        uint256 amount
    ) public {
        if (amount == 0) {
            revert ZeroTokens();
        }
        if (duration == 0) {
            revert ZeroDuration();
        }

        if (!token.transferFrom(msg.sender, address(this), amount)) {
            revert TransferFailed();
        }

        uint256 endTime = startTime + duration;

        vestingSchedules[beneficiary].push(
            Vesting({
                startTime: startTime,
                endTime: endTime,
                totalAmount: amount,
                claimedAmount: 0
            })
        );

        emit VestingAdded(
            beneficiary,
            vestingSchedules[beneficiary].length - 1,
            startTime,
            endTime,
            amount
        );
    }
```

```
    function claim(uint256[] calldata scheduleIds) public {
        uint256 claimableAmount = 0;
        Vesting[] storage schedules = vestingSchedules[msg.sender];
        uint256 scheduleIdsLength = schedules.length;

        // @audit-info the function will revert if there is a
        // "invalid" schedule, while all others might be fine
        for (uint256 i = 0; i < scheduleIdsLength; i++) {
            uint256 id = scheduleIds[i];

            if (id >= schedules.length) {
                revert InvalidScheduleID();
            }

            Vesting storage vestingSchedule = schedules[id];

            if (block.timestamp < vestingSchedule.startTime) {
                revert VestingNotStarted();
            }

            if (vestingSchedule.totalAmount <= vestingSchedule.claimedAmount) {
                revert AllTokensClaimed();
            }

            // @audit-issue there's no check if the vesting schedule is over
            uint256 elapsedTime = block.timestamp - vestingSchedule.startTime;
            uint256 vestingDuration = vestingSchedule.endTime -
                vestingSchedule.startTime;
            uint256 vestedAmount = (vestingSchedule.totalAmount * elapsedTime) /
                vestingDuration;

            console.log(vestedAmount);
            uint256 unclaimedAmount = vestedAmount -
                vestingSchedule.claimedAmount;

            if (unclaimedAmount > 0) {
                vestingSchedule.claimedAmount = vestedAmount;
                claimableAmount += unclaimedAmount;

                emit TokenWithdrawn(msg.sender, id, unclaimedAmount);
            }
        }

        if (claimableAmount == 0) {
            revert NothingToClaim();
        }
        if (!token.transfer(msg.sender, claimableAmount)) {
            revert TransferFailed();
        }
    }
}


contract ERC20MintableToken is ERC20 {
    constructor(
        string memory _symbol,
        string memory _name
    ) ERC20(_symbol, _name) {}

    /**
     * @dev Function to mint tokens
```

```
     * @param to The address that will receive the minted tokens.
     * @param value The amount of tokens to mint.
     * @return A boolean that indicates if the operation was successful.
     */

    function mint(address to, uint256 value) public returns (bool) {
        _mint(to, value);
        return true;
    }
}


contract InflatedRewardsTest is Test {
    ERC20MintableToken token;
    TokenVesting vesting;
    address amelie = makeAddr("amelie");
    address noemi = makeAddr("noemie");

    function setUp() public {
        vm.createSelectFork("https://arb-mainnet.g.alchemy.com/v2/UVXidxBjyLOdMXEJxYtCMqqEkHATR2gQ");
        token = new ERC20MintableToken("Test", "TEST");
        vesting = new TokenVesting(token);
    }

    function testInflatedRewards() external {
        // Noemi vests 100k tokens
        vm.startPrank(noemi);
        token.mint(noemi, 100_000e18);
        token.approve(address(vesting), 100_000e18);
        vesting.addVesting(noemi, block.timestamp, 90 days, 100_000e18);
        vm.stopPrank();

        assertEq(token.balanceOf(address(vesting)), 100_000e18);

        vm.warp(block.timestamp + 1 days);

        // Amelie vests 10k tokens
        vm.startPrank(amelie);
        token.mint(amelie, 10_000e18);
        token.approve(address(vesting), 10_000e18);
        vesting.addVesting(amelie, block.timestamp, 1, 10_000e18);

        assertEq(token.balanceOf(address(vesting)), 110_000e18);

        vm.warp(block.timestamp + 11 seconds);

        uint256[] memory scheduleIds = new uint256[](1);
        scheduleIds[0] = 0;

        // amelie steals all the tokens in the contract
        vesting.claim(scheduleIds);

        assertEq(token.balanceOf(amelie), 110_000e18);

        assertEq(token.balanceOf(address(vesting)), 0);
    }
}
```

Recommendation

To fix this vulnerability, a check should be added to ensure the elapsed time does not exceed the vesting du-

ration before calculating the vested amount. This can be done by adding a conditional statement to verify if `block.timestamp` is greater than `vestingSchedule.endTime`.

Here is an example of how this can be implemented:

```
uint256 elapsedTime;
if (block.timestamp < vestingSchedule.endTime) {
    elapsedTime = block.timestamp - vestingSchedule.startTime;
} else {
    elapsedTime = vestingSchedule.endTime - vestingSchedule.startTime;
}

uint256 vestingDuration = vestingSchedule.endTime - vestingSchedule.startTime;
uint256 vestedAmount = (vestingSchedule.totalAmount * elapsedTime) / vestingDuration;
```

[C-2] getLPTokenPrice can be manipulated through a flash loan

Context: LockdropPhase1.sol

Impact

This vulnerability allows for manipulation of the Total Value Locked (TVL) and Liquidity Provider (LP) token price. If exploited, it could lead to incorrect pricing of LP tokens and potentially significant financial loss to users and the platform itself. It could also damage the platform's reputation and user trust.

```
function _getLPTokenPrice(
    uint256 _tokenIndex
) internal view returns (uint256) {
    IUniswapV2Pair lpToken = lockingTokens[_tokenIndex].token;
    (uint112 _reserve0, uint112 _reserve1, ) = lpToken.getReserves();
    (uint112 reserveA, uint112 reserveB) = lpToken.token0() == tokenAAddress
        ? (_reserve0, _reserve1)
        : (_reserve1, _reserve0);
    uint256 totalValueA = uint256(reserveA) * tokenAPrice;
    uint256 totalValueB = uint256(reserveB) * tokenBPrice;
    uint256 totalValueLP = totalValueA + totalValueB;
    uint256 lpTokenSupply = lpToken.totalSupply();
    uint256 lpTokenPrice = totalValueLP / lpTokenSupply;
    return lpTokenPrice;
}
```

Description

The vulnerability lies within the getLPTokenPrice algorithm of Sparta's system. The TVL computation, `r0 * p0 + r1 * p1`, does not account for a constant `k` in its equation. Thus, the TVL doesn't necessarily stay constant as the equation suggests. When plotted, the TVL function has its global minimum at `r0_min = sqrt(k*p1/p0)` around 110M $.

The absence of explicit `k` in the equation and the ease of obtaining a multi-million dollar flash loan allows a user to freely "move" to a different point on the curve. This means a malicious user could exploit this by dumping a large amount of any token into the pool. Such an action would significantly increase the TVL and, subsequently, the price of an LP token (TVL divided by total LP supply).

https://rekt.news/warp-finance-rekt/

Recommendation

To mitigate this vulnerability, the following recommendations are proposed:

1. Update the getLPTokenPrice algorithm to derive ideal reserve values from K and the underlying prices
2. Implement safeguards against rapid and large changes in the TVL that could be indicative of manipulative actions.

3. Consider introducing limits or controls on the amount of tokens a user can add to the pool at a given time.

Additionaly, you can take inspiration from this UniswapV2Oracle by Alpha Homora.

[C-3] An adversary can lock user funds by spamming vesting schedules

Context: TokenVesting.sol

Impact

The way the `claim` function is currently implemented, it iterates over all vesting schedules of a user, regardless of the `scheduleIds` specified. This could potentially lead to a Denial of Service (DoS) attack if a malicious actor spams the `schedules` of an arbitrary user with a large number of vesting schedules. As a result, the targeted user's `claim` function call **will** run out of gas, effectively preventing them from claiming **any** of their vested tokens.

Description

The `addVesting` function is used to create vesting schedules for beneficiaries. This function is used by the `LockdropPhase1` contract when users receive rewards, with half of them sent to the vesting contract.

However, the `addVesting` function can also be called by anyone to generate third-party vesting schedules. This opens the door for an attacker to add numerous vesting schedules to a user's account, leading to gas inefficiencies when the user tries to claim their vested tokens.

In the `claim` function, the function loops over all vesting schedules of a user, even if the user only specified a subset of these schedules in the `scheduleIds` parameter. This could cause the function to run out of gas if the user has a large number of vesting schedules, particularly if these were added by a malicious actor.

```
function claim(uint256[] calldata scheduleIds) public {
        uint256 claimableAmount = 0;
        Vesting[] storage schedules = vestingSchedules[msg.sender];
        uint256 scheduleIdsLength = schedules.length;

        for (uint256 i = 0; i < scheduleIdsLength; i++) {
            uint256 id = scheduleIds[i];

            if (id >= schedules.length) {
                revert InvalidScheduleID();
            }

            Vesting storage vestingSchedule = schedules[id];
```

Recommendation

To mitigate this potential issue, the `claim` function should be refactored to loop over only the `scheduleIds` specified by the user, rather than all vesting schedules associated with the user. This can be achieved by changing the `scheduleIdsLength` variable to be the length of the `scheduleIds` array, rather than the length of the user's `schedules` array.

Here's how the changes would look like:

```
        Vesting[] storage schedules = vestingSchedules[msg.sender];
+       uint256 scheduleIdsLength = scheduleIds.length;

        for (uint256 i = 0; i < scheduleIdsLength; i++) {
            uint256 id = schedules[scheduleIds[i]];

            ...

            Vesting storage vestingSchedule = schedules[id];
```

[C-4] Excessive user allocations can DoS the calculateTotalReward function and deprive the user from rewards

Context: LockdropPhase1.sol

Impact: The presence of this vulnerability may discourage users from making numerous allocations, as it could potentially leave them unable to access their rewards. This issue stems from a scenario where a user has a significant number of allocations, leading to the exhaustion of gas in the `calculateTotalReward` function.

Description: The `lock` function allows a user to lock a specified amount of tokens until a certain timestamp, potentially with a boost calculated based on the token index and the value.

First, it updates the user allocation counter and then updates the user's allocation itself, including the value, boost, token, unlock timestamp index, and points,

https://github.com/SpartaDEX/sdex-smart-contracts/blob/27d41915ff88f798123c47a04af5f3bff3b5fe83/contracts/lockdrop/LockdropPhase1.sol#L112-L128

```
        totalPointsInRound[stampId] += points;
        }

        IUniswapV2Pair token = lockingTokens[_tokenIndex].token;
        if (!token.transferFrom(msg.sender, address(this), _value)) {
            revert TransferFailed();
        }

        uint256 nextWalletAllocations = ++userAllocationsCount[msg.sender];
        userAllocations[msg.sender][nextWalletAllocations] = UserAllocation({
            taken: false,
            value: _value,
            boost: boost,
            token: token,
            unlockTimestampIndex: _lockingExpirationTimestampIndex,
            points: points
        });
```

Then, the total reward for the user is calculated through the `calculateTotalReward` function. The `calculateTotalReward` function iterates over all the allocations for a given user. If a user has a high number of allocations, the loop operation can exceed the gas limit, causing the function to fail. This is a problem as the `calculateTotalReward` function is fundamental to the `getRewardAndSendOnVesting` and `allocateRewardOnLockdropPhase2` functions. Consequently, a user may be unable to claim their rewards due to this vulnerability.

```
function calculateTotalReward(
        address _wallet
    )
        public
        view
        atLeastTheLockdropState(LockdropState.REWARD_RATES_CALCULATED)
        returns (uint256)
    {
        uint256 reward = 0;

        // @audit-issue will DoS if too many allocations
        uint256 allocationsLength = userAllocationsCount[_wallet];
        for (
            uint256 allocationId = 1;
            allocationId <= allocationsLength;
            ++allocationId
        ) {
```

Recommendation: The `calculateTotalReward` function should be refactored to handle a large number of allocations without exceeding the gas limit. One approach could be to design the function so it can process allocations

in ranges. This would allow it to handle a large number of allocations over multiple transactions, rather than attempting to process all allocations in a single transaction.

5.2 High Risk

[H-1] Unclaimed tokens are locked forever in the Airdrop.sol contract

Context: Airdrop.sol

Unclaimable tokens are stuck in the Airdrop.sol contract.

Impact: As a result of this issue, tokens that are not claimed during the designated timeframe, or "dust" tokens left over due to small discrepancies in calculations, remain stuck in the Airdrop.sol contract indefinitely. This means these tokens are effectively lost, decreasing the total available supply and potentially distorting the token's market dynamics.

Description: Airdrop.sol facilitates an airdrop mechanism for distributing tokens. It includes functions for setting claimable amounts for users, allowing users to claim their tokens once the claim start timestamp has passed, and enabling users to lock tokens on the LockdropPhase2 contract

The problem is that there might be the scenario where some tokens are left unclaimed (for example, if no user claim rewards at all) OR there is some dust left. In this case, the contract owners might want to retrieve the unclaimed tokens.

Recommendation

Implement a token recovery function that can be executed only by `onlyAirdropManagerRole`, but only after the airdrop period ends.

This will allow addresses which have the onlyAirdropManagerRole to recover excess tokens from the contract.

```
function recoverERC20(uint256 tokenAmount) external onlyAirdropManagerRole {
    IERC20(token).safeTransferFrom(address(this), msg.sender, tokenAmount);
}
```

[H-2] addTargetLiquidity and removeSourceLiquidity has no slippage protection, which can result in in loss of funds

Context: LockdropPhase1.sol

Impact

The current implementation of the `removeSourceLiquidity()` function in the smart contract lacks slippage protection. This omission might expose users to potential price manipulation, where malicious actors could intentionally cause price slippage, leading to significant losses when users remove liquidity. As a result, users could lose a significant portion of their funds when they remove liquidity from the pool, which could erode trust in the smart contract.

Description

The `removeSourceLiquidity()` function allows users to remove liquidity that they've previously added to the pool. This function iterates through each locking token and retrieves the balance of the token for the smart contract. However, when removing liquidity from the pool, there's no provision for slippage protection.

In the existing implementation, the function uses the `removeLiquidity()` function of the router without specifying the minimum amounts of tokens that should be received in return (`amountAMin` and `amountBMin` are set to zero):

```
function removeSourceLiquidity()
    external
    onlyOnLockdropState(LockdropState.REWARD_RATES_CALCULATED)
    onlyLockdropPhase1Resolver
{
    if (sourceLiquiditiesRemoved_) {
        revert SourceLiquidityAlreadyRemoved();
    }
    uint256 lockingTokensLength = lockingTokens.length;
```

```
        for (
            uint32 lockingTokenId = 0;
            lockingTokenId < lockingTokensLength;
            ++lockingTokenId
        ) {
            IUniswapV2Pair token = lockingTokens[lockingTokenId].token;
            uint256 balance = token.balanceOf(address(this));
            IUniswapV2Router02 router = lockingTokens[lockingTokenId].router;
            if (!token.approve(address(router), balance)) {
                revert ApproveFailed();
            }

            (address token0, address token1) = _tokens();

            if (balance == 0) {
                continue;
            }


            // @audit-issue no slippage protection
            router.removeLiquidity(
                token0,
                token1,
                balance,
>>                  0,
>>                  0,
                address(this),
                block.timestamp
            );
        }

        sourceLiquiditiesRemoved_ = true;
    }
```

This implementation essentially means that the function would still execute successfully even if the actual amounts of tokens received are far less than expected due to price slippage, which could occur due to market volatility or malicious manipulation.

Unfortunately, the same issue applies to `addTargetLiquidity`.

```
    function addTargetLiquidity(
        IUniswapV2Router02 _router
    )
        external
        onlyLockdropPhase1Resolver
        onlyOnLockdropState(LockdropState.SOURCE_LIQUIDITY_EXCHANGED)
    {
        if (address(spartaDexRouter) != address(0)) {
            revert TargetLiquidityAlreadyProvided();
        }

        spartaDexRouter = _router;
        address spartaDexRouterAddress = address(spartaDexRouter);

        // @audit-issue no slippage protection
        (, , initialLpTokensBalance) = _router.addLiquidity(
            tokenAAddress,
            tokenBAddress,
            _allowMaxErc20(IERC20(tokenAAddress), spartaDexRouterAddress),
            _allowMaxErc20(IERC20(tokenBAddress), spartaDexRouterAddress),
>             1,
>             1,
            address(this),
            block.timestamp
        );
    }
```

Recommendation: To mitigate the risk of losing funds due to price slippage, it's recommended to add slippage protection to both functions. This protection can be added by specifying the minimum amounts of tokens (`amountAMin` and `amountBMin`) that should be received when removing liquidity and pass them to `router.removeLiquidity` and `router.addLiquidity` respectively.

[H-3] LockdropPhase2 lacks token recovery mechanism

Context: LockdropPhase2.sol

Impact

The lack of a proper token recovery mechanism in the `LockdropPhase2` contract for unclaimed tokens presents a significant financial risk. In the current setup, any unclaimed "sparta" and "stable" tokens become irretrievable,

Description

The `LockdropPhase2` enables users to lock up their tokens (specifically, "sparta" and "stable" tokens) for a certain period and then later claim rewards based on the amount of tokens locked. The contract includes functions for locking tokens, unlocking tokens, and calculating rewards based on the total locked amount of each token type

The issue is that if some of these tokens are left unclaimed by users, there is no way they can be recovered from the contract owners.

Recommendation Implement a token recovery function that can be executed only after a certain state and only by the owner of the contract.

This will allow the owners of the protocol to recover excess tokens from the contract.

```
    function recoverERC20(uint256 tokenAmount) external onlyAirdropManagerRole {
        IERC20(token).safeTransferFrom(address(this), msg.sender, tokenAmount);
    }
```

[H-4] ExchangeTokens lacks slippage control

Context: LockdropPhase2.sol

Impact: Without slippage protection, there's a significant risk of unfavorable trade outcomes. The lack of guarantee that `spartaTotalLocked` will equal the stable coins required can lead to substantial financial loss. This becomes particularly concerning during periods of high market volatility.

Description:

The `exchangeTokens` function in the LockdropPhase2 contract, is designed to add liquidity via the DEX router.

However, currently the call lacks slippage protection, which is critical to prevent excessive losses during trade execution. The function allows for liquidity provision with minimum amounts (`1` for both `sparta` and `stable` tokens), without specifying the minimum amounts of tokens that should be received in return (`amountAMin` and `amountBMin` are set to 1):

```
    function exchangeTokens(
        IUniswapV2Router02 router_
    )
        external
        onlyOnLockdropState(LockdropState.ALLOCATION_FINISHED)
        onlyLockdropPhase2Resolver
    {
        (, , initialLpTokensBalance) = router_.addLiquidity(
            address(sparta),
            address(stable),
            _allowErc20(sparta, address(router_), spartaTotalLocked),
            _allowMaxErc20(stable, address(router_)),
            // @audit-issue no slippage protection
>             1,
>             1,
            address(this),
            block.timestamp
        );

        spartaDexRouter = router_;
    }
```

Recommendation

To mitigate the risk of losing funds due to price slippage, it's recommended to add slippage protection to both functions. This protection can be added by specifying the minimum amounts of tokens (`amountAMin` and `amountBMin`) that should be received.

[H-5] AddLiquidity and removeLiquidity is called without expiration

Context: LockdropPhase1, LockdropPhase2

Impact: The current implementation can lead to significant risks including unfavorable trade outcomes and potential financial loss. Without a user-specified deadline, transactions can remain in the mempool for an extended period, resulting in execution at a potentially disadvantageous time. Moreover, by setting the deadline to `block.timestamp`, a validator can hold the transaction without any time constraints, further exposing users to the risk of price fluctuations.

Description

Advanced protocols like Automated Market Makers (AMMs) can allow users to specify a deadline parameter that enforces a time limit by which the transaction must be executed. Without a deadline parameter, the transaction may sit in the mempool and be executed at a much later time potentially resulting in a worse price for the user.

Protocols shouldn't set the deadline to block.timestamp as a validator can hold the transaction and the block it is eventually put into will be block.timestamp, so this offers no protection.

No expiration deadline may create a potential critical loss of funds vulnerability for any swap, especially if there is also no slippage parameter.

However, all the calls to addLiquidity and removeLiquidity right now are assigned `block.timestamp` as deadline

https://github.com/SpartaDEX/sdex-smart-contracts/blob/27d41915ff88f798123c47a04af5f3bff3b5fe83/contracts/lockdrop/LockdropPhase2.sol#L230-L249

```
    function exchangeTokens(
        IUniswapV2Router02 router_
    )
        external
        onlyOnLockdropState(LockdropState.ALLOCATION_FINISHED)
        onlyLockdropPhase2Resolver
    {
        (, , initialLpTokensBalance) = router_.addLiquidity(
            address(sparta),
            address(stable),
            _allowErc20(sparta, address(router_), spartaTotalLocked),
            _allowMaxErc20(stable, address(router_)),
            1,
            1,
            address(this),
>             block.timestamp
        );

        spartaDexRouter = router_;
    }
```

https://github.com/SpartaDEX/sdex-smart-contracts/blob/27d41915ff88f798123c47a04af5f3bff3b5fe83/contracts/lockdrop/LockdropPhase1.sol#L458-L467

```
        (, , initialLpTokensBalance) = _router.addLiquidity(
            tokenAAddress,
            tokenBAddress,
            _allowMaxErc20(IERC20(tokenAAddress), spartaDexRouterAddress),
            _allowMaxErc20(IERC20(tokenBAddress), spartaDexRouterAddress),
            1,
            1,
            address(this),
>             block.timestamp
        );
```

https://github.com/SpartaDEX/sdex-smart-contracts/blob/27d41915ff88f798123c47a04af5f3bff3b5fe83/contracts/lockdrop/LockdropPhase1.sol#L458-L467

```
            router.removeLiquidity(
                token0,
                token1,
                balance,
                0,
                0,
                address(this),
>                 block.timestamp
            );
```

Recommendation: To mitigate this issue, it's recommended to add deadline parameters to all functions interacting with AMMs and allow users to specify their preferred deadlines. These user-specified deadlines should then be passed on to the respective AMM function calls. This modification will give users more control over their transactions, reducing the associated risks. Here is a conceptual code suggestion:

```
function exchangeTokens(
    IUniswapV2Router02 router_,
    uint256 deadline  // added deadline argument
)
    external
    onlyOnLockdropState(LockdropState.ALLOCATION_FINISHED)
    onlyLockdropPhase2Resolver
{
    (, , initialLpTokensBalance) = router_.addLiquidity(
        address(sparta),
        address(stable),
        _allowErc20(sparta, address(router_), spartaTotalLocked),
        _allowMaxErc20(stable, address(router_)),
        1,
        1,
        address(this),
        deadline  // pass the deadline to the function call
    );

    spartaDexRouter = router_;
}
```

5.3 Medium Risk

[M-1] Potential mismatch between claimable amounts and actual token balance

Context: Airdrop

Impact: The `setClaimableAmounts` function could potentially allow the administrators to set claimable token amounts for users that exceed the actual token balance of the contract. This could lead to a situation where users are unable to claim their tokens despite being told they have a certain amount available. This discrepancy can undermine trust in the contract, the airdrop process, and the overall project.

Description: The `setClaimableAmounts` function sets the amount of tokens that specific users are eligible to claim from an airdrop. The function is only accessible by addresses with the `onlyAirdropManagerRole`. This function takes in two arrays, `users` and `amounts`, representing the addresses of the users and the respective amounts they can claim.

However, this function does not validate whether the contract's actual token balance is sufficient to cover the total claimable amounts set for all users. This could potentially allow the contract administrators to allocate more tokens than available in the contract, leading to a shortfall when users attempt to claim their tokens.

```
function setClaimableAmounts(
    address[] memory users,
    uint256[] memory amounts
) external override onlyAirdropManagerRole {
    if (users.length != amounts.length) {
        revert ArraysLengthNotSame();
    }

    uint256 length = users.length;
    for (uint256 i = 0; i < length; ++i) {
        claimableAmounts[users[i]] = amounts[i];
        emit WalletAdded(users[i], amounts[i]);
    }
}
```

Recommendation: It's recommended to add a check to ensure the total claimable amounts do not exceed the actual token balance of the contract. This can be done by calculating the sum of all claimable amounts and comparing it to the contract's token balance. If the total claimable amount exceeds the contract's balance, the function should revert the transaction.

```
        unchecked {
            sum += _claimableAmount[i];
        }
    }

    // sanity check that the current has been sufficiently allocated
    require(token.balanceOf(address(this)) >= sum, "TokenDistributor: not enough balance");
    totalClaimable = sum;
```

Alternatively, to ensure that the claimable amounts always correspond to the actual tokens available, you could modify the `setClaimableAmounts` function so that it pulls the corresponding amount of tokens from the sender (i.e., the address with `onlyAirdropManagerRole`) when setting the claimable amounts. This way, every time claimable amounts are set, the tokens are immediately transferred to the contract, guaranteeing that the contract has enough tokens for users to claim.

```
            unchecked {
                sum += _claimableAmount[i];
            }
        }
        // Transfer the total amount of tokens from the sender to the contract
        require(IERC20(token).transferFrom(msg.sender, address(this),totalAmount), "Token transfer failed");
        totalClaimable = sum;
```

[M-2] Potential manipulation of airdrop claimable amounts in setClaimableAmounts function

Context: Airdrop

Description: The `setClaimableAmounts` function sets the amount of tokens that specific users are eligible to claim from an airdrop. The function is only accessible by addresses with the `onlyAirdropManagerRole`. This function takes in two arrays, `users` and `amounts`, representing the addresses of the users and the respective amounts they can claim.

```
function setClaimableAmounts(
    address[] memory users,
    uint256[] memory amounts
) external override onlyAirdropManagerRole {
    if (users.length != amounts.length) {
        revert ArraysLengthNotSame();
    }

    uint256 length = users.length;
    for (uint256 i = 0; i < length; ++i) {
        claimableAmounts[users[i]] = amounts[i];
        emit WalletAdded(users[i], amounts[i]);
    }
}
```

However, the function as currently written can potentially be exploited. Specifically, the `onlyAirdropManagerRole` user has the ability to manipulate the claimable amounts even after they are first set. This could be used to arbitrarily change the amount of tokens a user can claim, which could lead to unfair token distribution or even potential loss of tokens for the user.

Another potential issue is that if the `users` array contains duplicate addresses, the claimable amounts for these addresses will be overwritten by the last corresponding amount in the `amounts` array. This means that a user might not receive the amount they were originally intended to, or they could potentially receive more than they should. This could lead to errors in token distribution and potential loss or over-distribution of tokens.

Recommendation: Use += instead of =

```
for (uint256 i = 0; i < length; ++i) {
+       claimableAmounts[users[i]] += amounts[i];
    emit WalletAdded(users[i], amounts[i]);
}
```

[M-3] Duplicate locking expiration timestamps can dillute the rewards

Context: LockdropPhase1.sol

Impact The current implementation of the `calculateRewardRates()` function in the LockdropPhase1.sol smart contract has a potential vulnerability where rewards might get diluted. This dilution could occur if there are any duplicate locking expiration timestamps, which could lead to a duration of zero during reward calculation. As a result, users may receive less reward than they are supposed to, impacting the fairness and trustworthiness of the reward distribution mechanism.

Description The `calculateRewardRates()` function calculates reward rates for different time ranges based on locking expiration timestamps and total points in each round. The function is designed to prevent multiple calls once the reward rates have been calculated. However, it doesn't account for a situation where there could be duplicate timestamps.

```
function calculateRewardRates()
    external
    override
    onlyOnLockdropState(LockdropState.TOKENS_ALLOCATION_FINISHED)
{
    if (rewardRateCalculated_) {
        revert RewardRatesAlreadyCalculated();
    }

    uint256 timeRangesLength = lockingExpirationTimestamps.length;
    // @audit-issue timeRangesLength could DoS, as it has no limit
    uint256 lastTimestamp = lockingEnd;

    for (
        uint32 timeRangeIndex = 0;
        timeRangeIndex < timeRangesLength;
        ++timeRangeIndex
    ) {
        if (totalPointsInRound[timeRangeIndex] != 0) {

            // @audit-issue if last lockingEnd == lockingExpirationTimestamps[timeRangeIndex]
            // duration will be 0, if its the first locking expiration timestamp
            // that will dillute the rewards
            uint256 duration = lockingExpirationTimestamps[timeRangeIndex] -
                lastTimestamp;
            rewardPerPointInTimeRange[timeRangeIndex] =
                (rewardRate * duration) /
                totalPointsInRound[timeRangeIndex];
        }

        lastTimestamp = lockingExpirationTimestamps[timeRangeIndex];
    }

    rewardRateCalculated_ = true;
}
```

Here, `lastTimestamp` is set to the current locking expiration timestamp for each iteration:

```
lastTimestamp = lockingExpirationTimestamps[timeRangeIndex];
```

If there happen to be duplicate timestamps, `duration` will equate to zero. Consequently, the calculated reward per point for that time range will also be zero.

Running the following test

https://github.com/SpartaDEX/sdex-smart-contracts/blob/27d41915ff88f798123c47a04af5f3bff3b5fe83/test/lockdrop/lockdrop.test.ts#L755-L768

```
    const allocation2 = await lockdropPhase1.userAllocations(
      lpProvider1.address,
      2
    );



    const expectedReward = await calculateExpectedRewardFromAllocations([
      allocation1,
      allocation2,
    ]);

    console.log(expectedReward);
```

With these rewards

```
expirationTimestamps = [
  lockdropPhase2LockingEnd,
  lockdropPhase2LockingEnd + 100,
  lockdropPhase2LockingEnd + 200,
  lockdropPhase2LockingEnd + 350,
  lockdropPhase2LockingEnd + 1000,
];
```

However, if the tests are executed with the rewards below

```
expirationTimestamps = [
  lockdropPhase2LockingEnd,
  lockdropPhase2LockingEnd,
  lockdropPhase2LockingEnd + 100,
  lockdropPhase2LockingEnd + 200,
  lockdropPhase2LockingEnd + 350,
  lockdropPhase2LockingEnd + 1000,
];
```

Recommendation: It is recommended to add a check during the constructor time to ensure that the `lockingExpirationTimestamps` array doesn't contain any duplicate values.

5.4 Low Risk

[L-1] Missing sanity checks for claimStartTimestamp in Airdrop contract

Context: Airdrop.sol

Impact

The absence of sanity checks for the `_claimStartTimestamp` parameter in the Airdrop contract can lead to potential vulnerabilities and unexpected behavior. It may allow the contract to be initialized with a `claim` start timestamp that is not in the future, which **could** undermine the intended functionality and fairness of the airdrop.

Description

In the `Airdrop` contract, the constructor accepts a `_claimStartTimestamp` parameter to set the start timestamp for claiming airdrop tokens. However, there are no explicit sanity checks to ensure that the provided `_claimStartTimestamp` value is in the future.

```
uint256 _claimStartTimestamp,
        ILockdropPhase2 _lockdrop,
        IAccessControl _acl
    ) {
        token = _token;
        claimStartTimestamp = _claimStartTimestamp;
```

Recommendation

To mitigate this vulnerability, it is crucial to implement sanity checks and ensure that the `_claimStartTimestamp` value provided in the constructor is in the future. For example, you can add a validation check within the constructor to ensure that `_claimStartTimestamp` is greater than the current `block timestamp`.

This can be done using a condition such as `require(_claimStartTimestamp > block.timestamp, "Claim start time must be in the future");`.

[L-2] Possibility for division by zero in LockdropPhase1

Context: LockdropPhase1.sol

Impact

The vulnerability in the `LockdropPhase1.sol` contract can potentially lead to a division by zero error. This error can cause the function execution to revert, resulting in unexpected behavior or denial of service for users interacting with the contract.

Description

The issue lies in the calculation of the rewardRate variable in the `LockdropPhase1` contract. The rewardRate is calculated by dividing the `_rewardParams.rewardAmount` by the time difference between the last element in the `lockingExpirationTimestamps` array and the `lockingEnd` value.

```
rewardRate =
            _rewardParams.rewardAmount /
            (lockingExpirationTimestamps[
                lockingExpirationTimestamps.length - 1
            ] - lockingEnd);
```

However, if the value of `lockingExpirationTimestamps[lockingExpirationTimestamps.length - 1]` is equal to `lockingEnd`, the division operation would result in a division by zero error. This occurs because the denominator in the division operation becomes zero, which is an invalid mathematical operation.

As a result, when this condition is met, the contract will throw an exception and revert the transaction, preventing it from proceeding further.

Recommendation

To address this, make sure to sanitize the `lockingExpirationTimestamps` for erronous values before calculating the reward rate.

[L-3] LockdropPhase2 is prone to reentrancy

Context: LockdropPhase2

Impact: The code block performs a token transfer before updating the internal state (`stableTotalLocked`, `walletStableLocked[msg.sender]`). This violates the check-effects-interaction pattern, which is a best practice in smart contract development. If the token used is an ERC20 compatible ERC777 or has similar re-entrancy capabilities, the receiver of the `transfer` call could re-enter the contract and interact with it while its internal state is not yet updated, leading to to unlimited withdrawals.

```
        if (!stable.transfer(msg.sender, _amount)) {
            revert TransferFailed();
        }

        stableTotalLocked -= _amount;
        walletStableLocked[msg.sender] -= _amount;
```

Recommendation: To respect the check-effects-interaction pattern, the internal state should be updated before interacting with external contracts (i.e., before calling the `transfer` function). The following code changes are recommended:

```
+       stableTotalLocked -= _amount;
+       walletStableLocked[msg.sender] -= _amount;
        if (!stable.transfer(msg.sender, _amount)) {
            revert TransferFailed();
        }

-       stableTotalLocked -= _amount;
-       walletStableLocked[msg.sender] -= _amount;
```

By making these changes, the contract first updates the internal state and then interacts with the external token contract, which will mitigate the potential vulnerability.

[L-4] Violation of the Check-Effects-Interaction pattern in getRewardAndSendOnVesting

Context: LockdropPhase1.sol

Impact

This vulnerability can lead to potential unlimited claims of the reward if the reward token adheres to the ERC777 standard, which is backward compatible with ERC20. Although it's currently minor as the reward token is supposedly the `sparta` token, it can be a major issue if any other tokens are used in the future.

The `getRewardAndSendOnVesting` function is designed to distribute rewards to a user in a two-fold manner: immediate transfer and vesting. The function first validates if the second phase of locking has ended, then calculates the total reward for the user, divides it in half, and sends one half immediately while the other half is designated for vesting.

The issue is that it does not follow the Check-Effects-Interaction (CEI) pattern. Specifically, it attempts to transfer tokens before updating the critical `userRewardWithdrawn` state variable. This sequence can potentially allow a reentrancy attack, exploiting the token's hooks to call `getRewardAndSendOnVesting` again before the state variable is updated.

```
        uint256 reward = calculateTotalReward(msg.sender);
        uint256 alreadyWithdrawn = userRewardWithdrawn[msg.sender];
        if (_rewardClaimed(alreadyWithdrawn, reward)) {
            revert MaxRewardExceeded();
        }
        uint256 toSendOnVesting = reward / 2;
        uint256 remainingReward = toSendOnVesting - alreadyWithdrawn;
        if (!rewardToken.transfer(msg.sender, remainingReward)) {
            revert TransferFailed();
        }
        userRewardWithdrawn[msg.sender] = reward;


        vesting.addVesting(
            msg.sender,
            lockingEnd,
            VESTING_DURATION,
            _allowErc20(rewardToken, address(vesting), toSendOnVesting)
        );
```

Recommendation:

The solution to this vulnerability is to follow the Check-Effects-Interaction pattern by updating the state (`userRewardWithdrawn[msg.sender] = reward`) before attempting to transfer the tokens.

```
        uint256 reward = calculateTotalReward(msg.sender);
        uint256 alreadyWithdrawn = userRewardWithdrawn[msg.sender];
        if (_rewardClaimed(alreadyWithdrawn, reward)) {
            revert MaxRewardExceeded();
        }
        uint256 toSendOnVesting = reward / 2;
        uint256 remainingReward = toSendOnVesting - alreadyWithdrawn;
-       userRewardWithdrawn[msg.sender] = reward;
        if (!rewardToken.transfer(msg.sender, remainingReward)) {
            revert TransferFailed();
        }
+        userRewardWithdrawn[msg.sender] = reward;

        vesting.addVesting(
            msg.sender,
            lockingEnd,
            VESTING_DURATION,
            _allowErc20(rewardToken, address(vesting), toSendOnVesting)
        );
```

[L-5] Potential for Cross-Chain Replay attacks due to hard-coded Domain Separator

Context: SpartaDexERC20.sol

Impact

This vulnerability could lead to unauthorized transactions on a new chain after a split, posing significant security risks to users. In the worst case, assets could be unintentionally transferred or manipulated on the unintended chain.

Description

The smart contract in question computes the `DOMAIN_SEPARATOR` during initialization, which is then stored and subsequently used for signature verification. Notably, this `DOMAIN_SEPARATOR` contains the chain ID, which remains constant post-initialization.

The vulnerability arises in the event of a chain split. In such a case, only one chain will retain the original chain ID, with the other adopting a new one. Due to the static nature of the `DOMAIN_SEPARATOR`, a signature will be erroneously considered valid on both chains post-split. This opens up the potential for cross-chain replay attacks, as transactions on one chain could be replayed on the other.

```
constructor() {
    DOMAIN_SEPARATOR = keccak256(
        abi.encode(
            keccak256(
                "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
            ),
            keccak256(bytes(name)),
            keccak256(bytes("1")),
            1729,
            address(this)
        )
    );
}
```

To mitigate this vulnerability, it is recommended that the chain ID be computed dynamically rather than being hard-coded into the `DOMAIN_SEPARATOR` during initialization. This can be done using the `chainid()` function within the EVM, as illustrated below:

Recommendation

```
function _chainId() internal view returns (uint256 chainId_) {
    assembly {
        chainId_ := chainid()
    }
}
```

Furthermore, you should check whether the current chain ID matches the original one before computing the domain separator:

```
if (block.chainid != ORIGINAL_CHAIN_ID) {

        bytes32 domain_separator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes("...")),
        ↪  block.chainid, address(this)));
    } else {
        digestHash = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR, structHash));
    }
```

[L-6] Inconsistency in LiquidityProvided event parameters

Context: LockdropPhase1.sol

Impact

The inconsistent usage of event parameters in the `LiquidityProvided` event can lead to confusion and incorrect data representation

Description

The code snippet presents an inconsistency in the usage of event parameters for the `LiquidityProvided` event. The event declaration in the `ILockdropPhase1` contract defines the last parameter as `duration`, indicating the duration of the liquidity provision. However, in the `LockdropPhase1` contract, when the event is emitted, the last parameter is used to represent the `_value`, which does not align with the expected parameter.

For this event, the last parameter is duration

```
event LiquidityProvided(
    address indexed by,
    IUniswapV2Pair indexed pair,
    uint32 indexed durationIndex,
    uint256 duration
);
```

However, here we receive value instead as a last parameter

https://github.com/SpartaDEX/sdex-smart-contracts/blob/27d41915ff88f798123c47a04af5f3bff3b5fe83/contracts/lockdrop/LockdropPhase1.sol#L120-L125

```
emit LiquidityProvided(
    msg.sender,
    token,
    _lockingExpirationTimestampIndex,
    _value
);
```

Recommendation

Consider the following recommendations:

1. Update the event declaration in the `ILockdropPhase1` contract to match the usage in the `LockdropPhase1` contract. If the last parameter represents the `_value`, revise the event declaration to reflect this.
2. Alternatively, if the intended usage is to represent the duration, modify the event emission in the `LockdropPhase1` contract to include the correct duration value instead of `_value`.