CD SECURITY

AUDIT REPORT

Sweepr
June 2024

Prepared by
yotov721
Arnie
MaslarovK
radev_eth

Introduction

A time-boxed security review of the **Sweepr** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Sweepr**

Sweepr is an NFT liquidity flywheel designed to deliver value to creators and communities through provisioned liquidity and user incentivisation.

At the core of the protocol is the SWEEPR token, whose central function is designed to simultaneously fund the Sweepr treasury and filter incentives the users through token tax derived from DEX trading volume. In order to wholly reap the benefits, SWEEPR holders are encouraged to provide SWEEPR/ETH liquidity in which they can then stake to receive xSWEEPR.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

review commit hash **- cb71661a3164e5a0d48f9678f4075c9846e1aee7**

Scope

The following smart contracts were in scope of the audit:

- `token/SweeprToken.sol`
- `staking/Stake.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 2 issues
- Medium: 2 issues
- Low: 8 issues
- Informational: 5 issues

Findings Summary

ID	Title	Severity
[H-01]	Several state variables are not decreased when needed	High
[H-02]	Transfers will revert on contracts that do not contain a fallback	High
[M-01]	No slippage parameter or deadline protection set on swaps	Medium
[M-02]	New `_uniswapV2Router` lacks `setExcludeFromFees;`	Medium
[L-01]	Use `call` instead of `transfer` to send ETH	Low
[L-02]	Approve token amount to the uniswap router before the transfer	Low
[L-03]	Multiple centralization attack vectors are present in the protocol	Low
[L-04]	Use UniswapFactory to verify a contract is an uniswap pair	Low
[L-05]	No storage gap left in `Stake.sol` and `SweeprToken.sol`	Low
[L-06]	Off-by-one issues at several functions	Low
[L-07]	`onlyOwner` functions not accessible if `owner` renounces ownership	Low
[L-08]	Insufficient input validation across several functions	Low
[I-01]	No need to use unchecked increments in `for` loops after solidity 0.8.22	Informational
[I-02]	A Year is not always 365 days	Informational
[I-03]	Some variables can be made immutable	Informational
[I-04]	setUniswapV2Router function does not emit an event	Informational
[I-05]	Lock `pragma` to specific version	Informational

Detailed Findings

[H-01] Several state variables are not decreased when needed

Severity

Impact: High

Likelihood: Medium

Description

There are several functions that should update the ETH state variables but do not:

1. In the `Stake::emergencyWithdrawToken`, `totalLpAmount should be decreased if the lpToken is the one being withdrawn:

```
function emergencyWithdrawToken(
        address _token
    ) external onlyOwner whenPaused {
        IERC20(_token).transfer(
            owner(),
            //@audit totalLpAmount is not decreased
            IERC20(_token).balanceOf(address(this))
        );
    }
```

2. In the `Stake::emergencyWithdrawETH`, `_revenueShareIndex` and `totalEthRewards` are not decreased:

```
function emergencyWithdrawETH() external onlyOwner whenPaused {
        //@audit _revenueShareIndex and totalEthRewards are not decreased
        payable(msg.sender).transfer(address(this).balance);
    }
```

Recommendations

Decrease these values when needed to ensure proper behavior.

[H-02] Transfers will revert on contracts that do not contain a fallback

Severity

Impact: High

Likelihood: Medium

Description

When transferring tokens, we must check if the from or to are uniswap pairs, this can be observed in the function `isUniswapPair`.

```
    function isUniswapPair(address _target) public view returns (bool) {
        if (_target.code.length == 0) {
            return false;
        }

        IUniswapV2Pair pairContract = IUniswapV2Pair(_target);

        address token0;
        address token1;

        try pairContract.token0() returns (address _token0) {
            token0 = _token0;
        } catch (bytes memory) {
            return false;
        }
```

The revert occurs in the first try block, if the target address is a contract and does not have a fallback function and is also not a uniswap pair contract, the function will revert. This happens because when attempting to call `token0` function on a contract that does not include it, it will cause the fallback to be executed, in case the contract does not have a fallback the execution will revert.

This will cause a long term dos for also the stake contract which relies on the sweepr token.

Recommendations

Because a revert in a try block still reverts the entire transaction, find another way to see if an address is in fact a uniswap pair. A recommendation that can fix this may be to use `staticall` Something like

```
bytes memory data = abi.encodeWithSignature("token0()");
(bool success, bytes memory result) = pairContract.staticcall(data);
if (!success) {
    return false;
} else {
    if (result.length == 32) {
          // Check if result is address
          token0 = abi.decode(result, (address));
          if (token0 == address(0)) {
              return false;
          }
      } else {
          return false;
      }
```

```
    }

    // Analogical for token 1

    // Verify it is an uniswap pair via the uniswap factory
    return uniswapFactory.getPair(token0, token1) != address(0);
```

[M-01] No slippage parameter or deadline protection set on swaps

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol uses UniswapV2 to swap native tokens for ETH and vice-versa.

```
    uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
        swapAmountIn,
        0, // amountOutMin
        path,
        address(this),
        block.timestamp // deadline
    );
```

```
    uniswapV2Router.swapExactETHForTokensSupportingFeeOnTransferTokens{
value: swapAmountIn }(
        0, // amountOutMin
        path,
        address(this),
        block.timestamp // deadline
    );
```

There are two problems. The deadline for transaction execution time is hardcoded to `block.timestamp`. The minimum amount of tokens/ETH received back is 0 or in other words any amount.

Taking a look in the `uniswapV2Router` code we can see the following validations:

```
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
```

When the deadline is hardcoded as `block.timestamp` the transaction would not revert as the required statement is always fulfilled. If the fee the user provided is low it would not be interest to node validators to include the transaction and it could stay in the MEM pool for longer periods - days and even longer.

The transaction could be executed in worse market condition and the user would get a worse price.

```
 require(amountOut >= amountOutMin, 'UniswapV2Router:
INSUFFICIENT_OUTPUT_AMOUNT');
```

When `amountOutMin` is zero the require statement is always fulfilled. Any MEV bot could see this transaction in the MEM pool, aggressively sandwich attack it, and steal the majority of the funds.

Recommendations

Have the user input a slippage parameter to ensure that the amount of token they receive back from Uniswap is in line with what they expect. Use a user supplied deadline instead of `block.timestamp`

[M-02] New `_uniswapV2Router` lacks `setExcludeFromFees`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `SweeprToken.sol` contract, the `_uniswapV2Router` is excluded from fees during the `initialize()` function. However, when the `setUniswapV2Router()` function is called to update the Uniswap V2 Router address, the new router address is not excluded from fees. This oversight can lead to unexpected fee charges on transactions involving the new router, which can disrupt the token's integration with Uniswap.

`SweeprToken.sol#initialize()` function:

```
function initialize(
    uint256 _initialSupply,
    address _uniswapV2Router,
    address _treasuryWallet,
    uint256 _tradingTaxPercentage,
    uint256 _stakeContractPercentage,
    uint256 _treasuryPercentage
) public initializer {
    // ... other initializations ...
    uniswapV2Router = IUniswapV2Router02(_uniswapV2Router);
    setExcludeFromFees(_uniswapV2Router, true);
```

```
    // ... more code ...
}
```

`SweeprToken.sol#setUniswapV2Router()` function:

```
function setUniswapV2Router(address _uniswapV2Router) external onlyOwner {
    uniswapV2Router = IUniswapV2Router02(_uniswapV2Router);
    // Missing exclusion for the new router
}
```

Impact

- When a new Uniswap V2 Router is set, it is not excluded from fees, leading to additional fee charges on every transaction routed through it. This can affect liquidity provision, swapping, and other automated market-making functionalities.
- The additional fees can disrupt automated trading strategies, liquidity provision, and arbitrage activities, potentially leading to a less efficient market and reduced liquidity for the token.

Recommendations

Ensure that the new Uniswap V2 Router is also excluded from fees when it is set. Update the `setUniswapV2Router()` function to include the fee exclusion logic:

```
function setUniswapV2Router(address _uniswapV2Router) external onlyOwner {
    uniswapV2Router = IUniswapV2Router02(_uniswapV2Router);
    setExcludeFromFees(_uniswapV2Router, true); // Exclude the new router
from fees
}
```

Also can be added logic that call the `setExcludeFromFees()` function with `false` for the old `uniswapV2Router`

[L-01] Use `call` instead of `transfer` to send ETH

The protocol uses `transfer()` to send ETH on two occasions.

One is when it sends the collected tax to the treasury wallet in `SweeprToken`

```
if (treasuryWallet != address(0) && treasuryAmount != 0) {
  payable(treasuryWallet).transfer(treasuryAmount);
}
```

The other is when users claim revenue share rewards in the `Stake` contract

```
payable(msg.sender).transfer(revenueShare);
```

The problem with using `transfer()` to send ETH is that it forwards only 2300 gas and if the receiver is a smart contract that has `receive` or `fallback` function the 2300 gas could not be enough. Such contracts could be multi-sig wallets.

Use `call()` to send ETH. Also make sure to add the `nonReentrant` modifier because `call` opens up the possibility of reentrancy.

[L-02] Approve token amount to the uniswap router before the transfer

When executing swaps or adding liquidity in the UniswapV2Router the allowance of the router has to be enough for it to transfer the tokens from the sender. The protocol uses the following function to approve tokens to the router

```
  function _approveToken(address _token) private {
      if (
          IERC20(_token).allowance(address(this),
address(uniswapV2Router)) < 1e24
      ) {
          IERC20(_token).approve(address(uniswapV2Router), _MAX_INT);
      }
  }
```

Having in mind that most ERC20's including the SWEEPR token have 18 decimals and the function does NOT increase the allowance to `_MAX_INT` (`type(uint256).max -1`) if the allowance is less than 1e24, this leaves a minimum of 1e6 (1e24-1e18) or 1_000_000 when the allowance would not be updated.

This means that if the allowance is close to the threshold, but NOT bellow it and a user tries to deposit/swap more than the current allowance his transaction would revert, due to insufficient allowance, causing inconvenience.

Recommendations

Approve the tokens the router requires before every call to the router instead of one big approval when the allowance falls bellow the threshold (1e24)

In `Stake` add the following changes:

```
-    function _approveToken(address _token) private {
-        if (
-            IERC20(_token).allowance(address(this),
address(uniswapV2Router)) < 1e24
```

```
-            ) {
-            IERC20(_token).approve(address(uniswapV2Router), _MAX_INT);
-        }
-    }

...

  function _zapInSweepr(
      uint256 _tokenAmountIn
  ) internal returns (uint256 lpAmountOut) {

...

-       _approveToken(path[0]);

        uint256 ethBalanceBefore = address(this).balance;


uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            swapAmountIn,
            0, // @audit-issue DONE M-1: MEV sandwich attack
https://solodit.xyz/issues/m-01-mev-can-sandwich-every-harvest-due-to-
missing-slippage-tolerance-value-pashov-none-yield-ninja-markdown_
            path,
            address(this),
            block.timestamp
        );

        uint256 ethBalance = address(this).balance - ethBalanceBefore;

+       IERC20(path[0]).approve(address(uniswapV2Router), _tokenAmountIn -
swapAmountIn);
        (, , lpAmountOut) = uniswapV2Router.addLiquidityETH{value:
ethBalance}(
             path[0],
            _tokenAmountIn - swapAmountIn,
            0,
            0,
            address(msg.sender),
            block.timestamp
        );
}

...

    function _zapInETH(
        uint256 _tokenAmountIn
    ) internal returns (uint256 lpAmountOut) {

...

        uint256 sweeprBalance = sweeprToken.balanceOf(address(this)) -
            sweeprBalanceBefore;
```

```
-         _approveToken(path[1]);
+        IERC20(path[1]).approve(address(uniswapV2Router), sweeprBalance);

        (, , lpAmountOut) = uniswapV2Router.addLiquidityETH{
            value: _tokenAmountIn - swapAmountIn
        }(path[1], sweeprBalance, 0, 0, address(msg.sender),
block.timestamp);
    }
```

[L-03] Multiple centralization attack vectors are present in the protocol

The `Stake` contract has functions to withdraw all of the contract's ETH or an ERC20 token that are only callable by the owner

```
function emergencyWithdrawETH() external onlyOwner whenPaused {
    payable(msg.sender).transfer(address(this).balance);
}

function emergencyWithdrawToken(
    address _token
) external onlyOwner whenPaused {
    IERC20(_token).transfer(
        owner(),
        IERC20(_token).balanceOf(address(this))
    );
}
```

In the `Stake` contract the `_claimSweeprReward` function is used to send SWEEPR tokens to the user as rewards earned. The SWEEPR tokens are deposited into the contract by the admin. Before the tokens are sent to the user, `_updateSweeprReward` is called.

`_updateSweeprReward` add to the user rewards earned by calling `_calculateSweeprReward` and then updates the `_userSweeprRewardIndex` (`userLastRewardTime`) which is used in `_calculateSweeprReward`.

The first check in `_calculateSweeprReward` will return 0 if the SWEEPR token amount in the contract is 0. However, going back to the call stack in the `_updateSweeprReward` the `_userSweeprRewardIndex` is always updated to `block.timestamp`.

This is ok by itself. However, if the contract has no tokens but the emissions period is running the user would lose his reward.

These can result in a rug from the protocol owner, but it requires the owner's wallet to be compromised or for him to be a malicious owner.

Recommendations

Use a TimeLock contract to be the protocol owner, so users can monitor the admins' protocol actions. Another option is to make the admin a governance-controlled address.

[L-04] Use UniswapFactory to verify a contract is an uniswap pair

The `SweeprToken` contract's `update` function used to update user balances has extra logic that takes fee's if the `from` or `to` address is an uniswap pair. The address is checked if it is a uni pair if it has a `token0` and `token1` selectors.

```
        try pairContract.token0() returns (address _token0) {
            token0 = _token0;
        } catch (bytes memory) {
            return false;
        }

        try pairContract.token1() returns (address _token1) {
            token1 = _token1;
        } catch (bytes memory) {
            return false;
        }

        return token0 != address(0) && token1 != address(0);
```

However, any contract can implement these functions.

The severity is low because it is very unlikely (but not impossible) that the `to` or `from` address will be a contract that implements `token0` and `token1`. In the worst case, the user would just pay tax without having to, which would result in a minor loss.

Recommendations

Use Uniswap Factory's getPair to verify the to/from is a uniswap pair contract

```
// define and initialize uniswapFactory here

  function isUniswapPair(address _target) public view returns (bool) {
...
      try pairContract.token0() returns (address _token0) {
          token0 = _token0;
      } catch (bytes memory) {
          return false;
      }

      try pairContract.token1() returns (address _token1) {
```

```
            token1 = _token1;
        } catch (bytes memory) {
            return false;
        }

+.    return uniswapFactory.getPair(token0, token1) != address(0);
-      return token0 != address(0) && token1 != address(0);
    }
```

[L-05] No storage gap left in `Stake.sol` and `SweeprToken.sol`

For upgradeable contracts, there must be storage gap to "allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments" (quote OpenZeppelin).

Consider adding an appropriate storage gap at the end of upgradeable contracts such as the one below.

```
uint256[50] private __gap;
```

[L-06] Off-by-one issues at several functions

There are several functions across the contract introducing off-by-one issues. This means that the validation should be either inclusive or exclusive of some parameter, but is not and can result in problems in rare cases.

1. In the `Stake::calculateRevenueShareEarned`:

```
function calculateRevenueShareEarned(
        address _user
    ) external view returns (uint256) {
        if (
            //@audit off-by-one, revenueShareGracePeriod should be
inclusive and still return 0 if block.timestamp -
userDepositTimestamp[_user] <=  revenueShareGracePeriod
            block.timestamp - userDepositTimestamp[_user] <
revenueShareGracePeriod
        ) {
            return 0;
        }

        return _userRevenueShareEarned[_user] +
_calculateRevenueShare(_user);
    }
```

2. In the `Stake::_calculateRevenueShare`:

```
function _calculateRevenueShare(
        address _user
    ) private view returns (uint256) {
        if (address(this).balance == 0) {
            return 0;
        }

        if (
            //@audit off-by-one, revenueShareGracePeriod should be
inclusive and still return 0 if block.timestamp -
userDepositTimestamp[_user] <=  revenueShareGracePeriod
            block.timestamp - userDepositTimestamp[_user] <
revenueShareGracePeriod
        ) {
            return 0;
        }

        uint256 lpShare = userDeposit[_user];
        if (lpShare == 0) {
            return 0;
        }

        return
            (lpShare * (_revenueShareIndex -
_userRevenueShareIndex[_user])) /
            _REWARD_MULTIPLIER;
    }
```

3. . In the `Stake::_calculateSweeprReward`:

```
function _calculateSweeprReward(
        address _user
    ) private view returns (uint256) {
        if (sweeprToken.balanceOf(address(this)) == 0) {
            return 0;
        }

        uint256 lpShare = userDeposit[_user];
        if (lpShare == 0) {
            return 0;
        }

        uint256 userLastRewardTime = _userSweeprRewardIndex[_user];

        if (userLastRewardTime < emissionStartTimestamp) {
            userLastRewardTime = emissionStartTimestamp;
        }
```

```
        //@audit off-by-one, here it should be `if (block.timestamp <
emissionStartTimestamp)` as the `emissionStartTimestamp` is when it should
start, not after it.
        if (block.timestamp <= emissionStartTimestamp) {
            return 0;
        }
```

These issues are not that serious but can get to one block time difference as the block.timestamp is the same for all the transactions in a block.Consider changing the checks as described above.

[L-07] `onlyOwner` functions not accessible if `owner` renounces ownership

In OpenZeppelin's `Ownable` contract, the `renounceOwnership` function can be used to relinquish ownership, leaving the contract without an owner. This can pose a risk if done unintentionally or without a clear design plan, as it removes the ability to perform any owner-only functions, potentially limiting essential contract functionalities.

[L-08] Insufficient input validation across several functions

There are several instances of these issues across the `Stake.sol` contract:

1. In the `Stake::Initialize`

```
emissionStartTimestamp = _emissionStartTimestamp;//@audit there isnt check
if the emissionStartTimestamp is greater than block.timestamp
```

2. In the `Stake::setEmissionStartTimestamp`

```
function setEmissionStartTimestamp(
        uint256 _emissionStartTimestamp
        //@audit add check that the new emissionStartTimestamp is
different from the previous one
    ) external onlyOwner {
        if (
            block.timestamp < emissionStartTimestamp &&
            block.timestamp < _emissionStartTimestamp
        ) {
            emissionStartTimestamp = _emissionStartTimestamp;

            emit EmissionStartTimestampUpdated(_emissionStartTimestamp);
        }
        //@audit consider throwing an error so the owner is not confused
```

```
that it was successful.
    }
```

3. In the `Stake::setMaxZapReverseRatio`

```
function setMaxZapReverseRatio(
        uint256 _maxZapInverseRatio
        //@audit add check that the new maxZapReverseRatio is different
from the previous one
    ) external onlyOwner {
        maxZapReverseRatio = _maxZapInverseRatio;

        emit MaxZapReverseRatioUpdated(_maxZapInverseRatio);
    }
```

4. In the `Stake::setRevenueShareGracePeriod`

```
 function setRevenueShareGracePeriod(
        uint256 _revenueShareGracePeriod
         //@audit add check that the new revenueShareGracePeriod is
different from the previous one
    ) external onlyOwner {
        revenueShareGracePeriod = _revenueShareGracePeriod;

        emit RevenueShareGracePeriodUpdated(_revenueShareGracePeriod);
    }
```

5. In the `SweeprToken:initialize`

```
//@audit the tax percentages arent validated like its done in
setTradingTaxPercentage() function
        tradingTaxPercentage = _tradingTaxPercentage;
        stakeContractPercentage = _stakeContractPercentage;
        treasuryPercentage = _treasuryPercentage;
```

6. In the `SweeprToken:setStakeContract`

```
//@note add check that the new stake contract is different from the
previous one
    function setStakeContract(address _stakeContract) external onlyOwner {
// ok
        stakeContract = _stakeContract; // ok
        emit StakeContractUpdated(_stakeContract); // ok
    }
```

7. In the `SweeprToken:setTreasuryWallet`

```
//@note add check that the new treasury wallet address is different from
the previous one
    function setTreasuryWallet(address _treasuryWallet) external onlyOwner
{ // ok
        treasuryWallet = _treasuryWallet; // ok
        emit TreasuryWalletUpdated(_treasuryWallet); // ok
    }
```

[I-01] No need to use unchecked increments in `for` loops after solidity 0.8.22

Usually the unchecked block in `for` loop inside the loop body is used to save some gas by wrapping the loop counter in an unchecked block.

```
for (uint i = 0; i < array.length;) {
   acc += array[i];
   unchecked { i++; } // <=
}
```

However after solidity 0.8.22 this is no longer required because it is now set by default.

The protocol uses solidity version `^0.8.23` so it is safe to increment the loop counter inside the `for` loop parenthesis.

This is present in `Stake::_calculateSweeprReward()`

[I-02] A Year is not always 365 days

In leap years, the number of days is 366 instead of 365. Consequently, any calculations that assume a year is always 365 days will return incorrect values during leap years.

Instance of the Issue:

```
📁 File: contracts/staking/Stake.sol

802:                duration: 365 days,
```

[I-03] Some variables can be made immutable

Some of the variables in the `Stake.sol` are not intended to be changed, so they can be set to immutable.

```
IERC20 public sweeprToken; //@audit can be immutable
IERC20 public xSweeprToken;//@audit can be immutable
IWETH public weth;//@audit can be immutable
IUniswapV2Pair public lpToken; //@audit can be immutable
```

[I-04] setUniswapV2Router function does not emit an event

The `setUniswapV2Router` function does not emit an event when called:

```
    function setUniswapV2Router(address _uniswapV2Router) external
onlyOwner {
        uniswapV2Router = IUniswapV2Router02(_uniswapV2Router);
    }
```

Emit an event when calling `setUniswapV2Router`.

[I-05] Lock `pragma` to specific version

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. The project is currently using a floatable `^0.8.23` version.