e

CD SECURITY

AUDIT REPORT

Turbo

November 2024

Prepared by
tsvetanovv
GT_GSEC
minhquanym

Introduction

A time-boxed security review of the **Turbo** 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 **Turbo**

The Turbo Pillar is a decentralized auction system built around the TURBO and HYPER tokens, featuring Uniswap V3 for liquidity management and token burning mechanisms. It enables users to deposit tokens, participate in auctions, and receive proportional rewards. The protocol integrates token minting, supply control through burning, and automated reward distribution mechanisms.

Key functionalities include managing liquidity pools for TURBO and HYPER, fetching price data via Uniswap's TWAP, and ensuring efficient token distribution and burning. The system operates autonomously after initialization, with robust access controls and time-based constraints to ensure proper functionality and fairness.

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* - 6bc83028ca758dcad159603644835f2caffbec83**

Scope

The following smart contracts were in scope of the audit:

- `src/actions/*`
- `src/const/*`
- `src/interfaces/*`
- `src/Auction.sol`
- `src/Turbo.sol`
- `src/TurboBnB.sol`
- `src/TurboTreasury.sol`
- `src/VoltBurn.sol`

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

- Critical & High: 1 issues
- Medium: 0 issues
- Low & Info: 14 issues

Findings Summary

ID	Title	Severity
[H-01]	Incorrect amount of TURBO sent to treasury in `swapHyperToTurboAndBurn()`	High
[L-01]	Slippage security control does not take into account Uniswap's fee	Low
[L-02]	Incorrect check in `_addLiquidityToPool()` may cause TURBO tokens to become stuck	Low
[L-03]	VoltBurn: Default `swapCap` can be inefficient	Low
[I-01]	Slippage parameter has a misleading name	Informational
[I-02]	Missing input validation for `twapLookback`	Informational
[I-03]	Insufficient input validation for slippage	Informational
[I-04]	Default state variable visibility	Informational
[I-05]	Lack of two-step ownership transfer	Informational
[I-06]	The `_onlyEOA` modifier is overcomplicated	Informational
[I-07]	Inconsistent application of the SafeERC20 library	Informational
[I-08]	Duplicate condition within assertion	Informational
[I-09]	Redundant inheritance of `SwapActions` contract	Informational
[I-10]	Unused variable `volt` in `Auction` contract	Informational

ID	Title	Severity
[I-11]	Incorrect Natspec documentation	Informational

Findings Summary

Detailed Findings

[H-01] Incorrect amount of TURBO sent to treasury in `swapHyperToTurboAndBurn()`

Severity

Impact: High

Likelihood: High

Description

In the function `swapHyperToTurboAndBurn()`, which swaps HYPER tokens for TURBO tokens and allocates portions of the TURBO tokens to different addresses (e.g., treasury, liquidity bonding address), there is an issue with the amount of TURBO being sent to the treasury.

The amount being transferred to the treasury is specified as a fixed percentage (`0.5e18`). However, the implementation fails to multiply this percentage by the actual `turboAmount`, leading to an incorrect transfer value. As a result, the treasury receives an almost negligible amount of TURBO (just 0.5 TURBO), and the rest of the tokens intended for the treasury are incorrectly burned.

```
///@note - Allocations
turbo.transfer(LIQUIDITY_BONDING_ADDR, wmul(turboAmount,
uint256(0.08e18)));
turbo.transfer(address(turbo.treasury()), uint256(0.5e18)); // @audit
incorrect amount, forgot to multiply with `turboAmount`
burnTurbo();
```

Recommendations

Multiply the percentage (`0.5e18`) by the `turboAmount` to ensure the correct amount is sent to the treasury.

[L-01] Slippage security control does not take into account Uniswap's fee

Severity

Impact: Medium

Likelihood: High

Description

Within the `swapExactInput` function the expected amount of the `twapAmount` is firstly calculated by means of the `getTwapAmount` function before performing actual swap. Then, the `twapAmount` variable is used to determine the `minAmount` variable based on the `slippage` parameter. Eventually, this value is used as `amountOutMinimum` in Uniswap's structure `ExactInputParams`. However, such an approach does not take into account the fact that the actual swap will consume the fee applied. Presently, the pool fee is set to 1%. Thus, the actual amount out might be lower than calculated within the `minAmount` parameter. Thus, in the event of the high `slippage` application, the swap can revert due to an overestimated value set in the `amountOutMinimum`. Eventually, this weakness along with certain configurations prevents legitimate users from calling e.g. the `buyNSendToVoltTreasury` function.

```
//UNIV3
uint24 constant POOL_FEE = 10_000; //1%
```

```
    function swapExactInput(
        address tokenIn,
        address tokenOut,
        uint256 tokenInAmount,
        uint256 minAmountOut,
        uint32 deadline
    ) internal returns (uint256 amountReceived) {
...
        (uint256 twapAmount, uint224 slippage) = getTwapAmount(tokenIn,
tokenOut, tokenInAmount);

        uint256 minAmount = minAmountOut == 0 ? wmul(twapAmount, slippage)
: minAmountOut;

        ISwapRouter.ExactInputParams memory params =
ISwapRouter.ExactInputParams({
            path: path,
            recipient: address(this),
            deadline: deadline,
            amountIn: tokenInAmount,
            amountOutMinimum: minAmount
        });

        return ISwapRouter(uniswapV3Router).exactInput(params);
```

Recommendations

It is recommended to adjust the `minAmount` before inserting it into the `ExactInputParams` structure to take into account Uniswap's fee application and prevent any instance of revert due to overestimated `minAmountOut` value.

[L-02] Incorrect check in `_addLiquidityToPool()` may cause TURBO tokens to become stuck

Description

In the function `_addLiquidityToPool()`, liquidity is added to the Uniswap V3 pool. Due to slippage, the actual amount of tokens added might not exactly match the intended input. If there is a discrepancy, the leftover tokens are transferred to the owner, as shown in the code snippet below:

```
if (amount0Added < amount0Sorted) {
    IERC20(sortedToken0).transfer(owner(), amount0Sorted - amount0Added);
}
if (amount1Added < amount0Sorted) { // @audit incorrect check
    IERC20(sortedToken1).transfer(owner(), amount1Sorted - amount1Added);
}
```

The first block correctly checks if there is any leftover `token0` and transfers it to the owner. The second block does the same for `token1`, but the check is incorrect. Instead of comparing `amount1Added` with `amount1Sorted`, it mistakenly compares `amount1Added` with `amount0Sorted`. As a result, the logic inside that if block may not be triggered.

In the case where `token1` is HYPER, these tokens can still be used or distributed in future transactions. However, if `token1` is TURBO, the tokens will be stuck in the contract permanently because the check is incorrect and the transfer logic will never execute.

Recommendations

Fix the check to `if (amount1Added < amount1Sorted)`

[L-03] VoltBurn: Default `swapCap` can be inefficient

Description

The `VoltBurn` contract defines the upper bound limit for the `Hyper` tokens that can be swapped within a single call to the `buyNSendToVoltTreasury` function by means of the `swapCap` parameter. By default, this parameter is set to `uint128.max`. However, this default value might be inefficient:

- Exchanging any amount, including a significantly high amount, may have a negative impact on Uniswap's pair exchange price.
- Exchanging any amount each time may have a negative impact on the decentralised aspect of the function and unfair distribution of the incentive proposed by the functionality.

- Additionally, it is noticed that in every unit test the `changeSwapCap` function was called to explicitly set a certain amount.

```
    constructor(address _hyper, address _volt, SwapActionParams memory _s)
SwapActions(_s) {
...
        state.swapCap = type(uint128).max;
...
    }
```

```
 function buyNSendToVoltTreasury(uint32 _deadline)
...
        uint256 balance = erc20Bal(hyper);

        if (balance > $.swapCap) balance = $.swapCap;
...
```

Recommendations

It is recommended to review the business requirements and consider enforcing certain value set within the constructor for the `swapCap` parameter.

[I-01] Slippage parameter has a misleading name

Description

Within the `swapExactInput` function the slippage is applied to decrease the expected minimum amount received after the swap. It is done by calculating the percentage from the `twapAmount`. However, the slippage, in the context of crypto trading and AMMs, refers to the difference between the expected price of an asset and the actual price at which the trade is executed. Although the slippage and minimum amount out serve the same purpose, these two mechanisms have distinct applications. Eventually, such application of the slippage can be misleading, especially for the `slippageAdmin`, who can set an extremely low value for `slippage`, e.g. 2% (0.02e18).

```
    function swapExactInput(
...
        (uint256 twapAmount, uint224 slippage) = getTwapAmount(tokenIn,
tokenOut, tokenInAmount);

        uint256 minAmount = minAmountOut == 0 ? wmul(twapAmount, slippage)
: minAmountOut;
```

Recommendations

It is recommended to review the business requirements and consider changing the name of this parameter to reflect its application.

[I-02] Missing input validation for `twapLookback`

Description

The `changeSlippageConfig` function allows the privileged account to update the `slippage` and `twapLookback` variables for a particular Uniswap pool. However, the input validation for the `twapLookback` parameter appears to be insufficient, as it only checks whether the proposal value is not 0. To obtain a reliable price, the TWAP oracle requires checking values for a sufficiently long period. On the other hand, the considered period cannot be too long, as the provided price can be inaccurate as well. Currently, the implementation does not aid the slippage admin with setting this parameter within the possibly valid range of periods.

```
struct Slippage {
    uint224 slippage; //< Slippage in WAD (scaled by 1e18)
    uint32 twapLookback; //< TWAP lookback period in minutes (used as
seconds in code)
}
```

```
    function changeSlippageConfig(address pool, uint224 _newSlippage,
uint32 _newLookBack)
        external
        notAmount0(_newLookBack)
        onlySlippageAdminOrOwner
    {
        require(_newSlippage <= WAD, SwapActions__InvalidSlippage());

        emit SlippageConfigChanged(pool, _newSlippage, _newLookBack);

        slippageConfigs[pool] = Slippage({slippage: _newSlippage,
twapLookback: _newLookBack});
    }
```

Recommendations

It is recommended to review the business requirements and consider implementing input validation for the `twapLookback`, so it has minimum and maximum cap defined.

[I-03] Insufficient input validation for slippage

Description

The `changeSlippageConfig` function allows the privileged account to update the `slippage` and `twapLookback` variables for a particular Uniswap pool. However, the input validation for the `slippage` parameter can be considered insufficient. Presently, the function only checks whether the proposal value is lower than `WAD` (1e18). But, it does not enforce a minimum value that can be set. Thus, privileged accounts can set extremely low values, which will make the security control ineffective.

```
struct Slippage {
    uint224 slippage; //< Slippage in WAD (scaled by 1e18)
    uint32 twapLookback; //< TWAP lookback period in minutes (used as
seconds in code)
}
```

```
    function changeSlippageConfig(address pool, uint224 _newSlippage,
uint32 _newLookBack)
        external
        notAmount0(_newLookBack)
        onlySlippageAdminOrOwner
    {
        require(_newSlippage <= WAD, SwapActions__InvalidSlippage());

        emit SlippageConfigChanged(pool, _newSlippage, _newLookBack);

        slippageConfigs[pool] = Slippage({slippage: _newSlippage,
twapLookback: _newLookBack});
    }
```

Recommendations

It is recommended to review the business requirements and consider implementing input validation for the `insufficient`, so it has a minimum cap defined as well.

[I-04] Default state variable visibility

Description

Within the `VoltBurn` contract, two immutable variables are defined, however, `hyper` the first one lacks explicit visibility modifier defined. Thus, the default modifier is applied, which is `internal`.

```
contract VoltBurn is SwapActions {
    /* == IMMUTABLE == */

    ERC20Burnable immutable hyper;
    ERC20Burnable public immutable volt;
```

Similarly, within the `TurboAuction` contract following immutable variables: `turbo`, `volt`, `v3PositionManager` lack explicit visibility modifier defined.

```
contract TurboAuction is SwapActions {
    using SafeERC20 for *;

    //========IMMUTABLE=========//

    ITurbo immutable turbo;
    IHyper public immutable hyper;
    ERC20Burnable immutable volt;
    uint32 public immutable startTimestamp;
    address immutable v3PositionManager;
```

Recommendations

It is recommended to set the explicit visibility modifier for the aforementioned state variable. In this case, the `public` modifier can be considered sufficient and expected.

[I-05] Lack of two-step ownership transfer

Description

The `Turbo` and `SwapActions` contracts implement a single step of ownership transfer. In the event of a transfer to invalid address, the transfer is immediate and the authority is lost. Thus, access to all functionalities protected by the restricted modifier will be permanently lost.

```
contract SwapActions is Ownable, Errors {
```

Recommendations

It is recommended to implement two-step ownership transfer. In the first step a new proposal address should be provided. In the second step the proposal address should confirm the transfer. This can be achieved by importing the OpenZeppelin's Ownable2Step library.

[I-06] The `_onlyEOA` modifier is overcomplicated

Description

The `_onlyEOA` modifier implements a check to prevent a function from being called by the smart contract. Currently, it checks both whether the caller has a code and whether the `tx.origin` is equal to the `msg.sender`. However, the latter condition check only is sufficient to provide the expected functionality. Within the Ethereum blockchain, only the EOA can trigger transactions, thus it is not possible that the `tx.origin` global variable has a smart contract address value. On the other hand, the sole first condition

part can be insufficient, as a smart contract during the deployment, when the constructor is executed, has a code length set to 0.

```
    function _onlyEOA() internal view {
        require(msg.sender.code.length == 0 && tx.origin == msg.sender,
OnlyEOA());
    }
```

Recommendations

It is recommended to simplify the modifier to save some Gas during the deployment and execution:

```
    function _onlyEOA() internal view {
        require(tx.origin == msg.sender, OnlyEOA());
    }
```

[I-07] Inconsistent application of the SafeERC20 library

Description

OpenZepplin's `SafeERC20` library was proposed to handle transfers for compliant and non-compliant ERC20 tokens. The wrapper handles tokens that return a boolean or do not return a boolean and revert to indicate transfer failure. The `TurboBuyAndBurn`, `TurboTreasury`, `TurboAuction` makes use of the `SafeERC20` library whereas the `VoltBurn` does not. However, the application of the `safeTransfer` and `safeFromTransfer` functions is inconsistent across the protocol.

E.g. within the `Auction.sol` the `transfer` usage can be observed within the `_distribute` function and `safeTransfer` in the `collectFees`.

```
    function _distribute(uint256 _amount) internal {
        uint256 hyperBalance = hyper.balanceOf(address(this));
        //@note - If there is no added liquidity, but the balance exceeds
the initial for liquidity, we should distribute the difference
        if (!lp.hasLP) {
            if (hyperBalance <= INITIAL_HYPER_FOR_LP) return;

            _amount = uint192(hyperBalance - INITIAL_HYPER_FOR_LP);
        }

        hyper.transfer(DEAD_ADDR, wmul(_amount, HYPER_BURN));

        hyper.transfer(LIQUIDITY_BONDING_ADDR, wmul(_amount, TO_LP));
        hyper.transfer(DEV_WALLET, wmul(_amount, TO_DEV_WALLET));
        hyper.transfer(GENESIS_WALLET, wmul(_amount, TO_GENESIS));
```

```
        hyper.transfer(address(turbo.voltBurn()), wmul(_amount,
TO_VOLT_BURN));

        hyper.approve(address(turbo.turboBnb()), wmul(_amount,
TO_TURBO_BNB));
        turbo.turboBnb().distributeHyperForBurning(wmul(_amount,
TO_TURBO_BNB));
    }
```

```
    function collectFees() external returns (uint256 _turboAmount, uint256
_hyperAmount) {
        // Collect fees from TURBO/HYPER pool
        (_turboAmount, _hyperAmount) = _collectFeesFromPool(lp);

        // Transfer collected TURBO tokens
        if (_turboAmount > 0) turbo.safeTransfer(LIQUIDITY_BONDING_ADDR,
_turboAmount);

        // Transfer collected HYPER tokens
        if (_hyperAmount > 0) hyper.safeTransfer(LIQUIDITY_BONDING_ADDR,
_hyperAmount);
    }
```

Within the protocol three ERC20 tokens are being used: Hyper, Volt, and Turbo. In every instance, each token is EIP-20 compliant. The finding is reported as a deviation from leading security practices.

Recommendations

It is recommended to unify the usage and application of the `SafeERC20` library.

[I-08] Duplicate condition within assertion

Description

Within the `addInitialLiquidity` function there is an assertion implement that checks the same condition twice. Thus, it consumes additional Gas needlessly.

```
    function addInitialLiquidity(uint32 _deadline) external onlyOwner
notExpired(_deadline) {
        require(!lp.hasLP && !lp.hasLP, LiquidityAlreadyAdded());
...
```

Recommendations

Replace the redundant check with a single evaluation:

```
require(!lp.hasLP, LiquidityAlreadyAdded());
```

[I-09] Redundant inheritance of SwapActions contract

Description

The `TurboAuction` inherits from the `SwapActions`, however, it does not make any use of the functionality implemented within the parent contract The only meaningful functionality used is `Ownable`. Thus, the smart contract size is needlessly increased. Additionally,an overestimated amount of Gas is consumed during the contract's deployment.

```
contract TurboAuction is SwapActions {
...
```

Recommendations

It is recommended to remove the redundant inheritance.

[I-10] Unused variable `volt` in `Auction` contract

Description

In the Auction contract, the immutable variable `volt` is declared and initialized in constructor but never used anywhere in the code.

```
constructor(
    uint32 _startTimestamp,
    address _turbo,
    address _hyper,
    address _volt,
    address _v3PositionManager,
    SwapActionParams memory _s
) notAddress0(_turbo) notAddress0(_hyper) SwapActions(_s) {
    // if ((_startTimestamp - 14 hours) % 1 days != 0) revert
MustStartAt2PMUTC();

    turbo = ITurbo(_turbo);
    volt = ERC20Burnable(_volt);
    // ...
}
```

Recommendations

If the `volt` variable is not required, remove it from the contract to simplify the code.

[I-11] Incorrect Natspec documentation

Description

In the contract `TurboBnB`, the Natspec documentation for the function `swapHyperToTurboAndBurn()` is misleading. The documentation states that the function swaps Hyper for VOLT, but the actual implementation of the function swaps Hyper for TURBO. The incorrect Natspec is shown below:

```
/**
 * @notice Swaps Hyper for VOLT and distributes the VOLT tokens
 * @param _deadline The deadline for which the passes should pass
 */
function swapHyperToTurboAndBurn(uint32 _deadline) external intervalUpdate
notExpired(_deadline) {
```

Recommendations

Update the Natspec documentation to accurately describe the function's behavior.