CD SECURITY

AUDIT REPORT

Ascendant

December 2024

Prepared by
MrPotatoMagic
tsvetanovv

Introduction

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

Ascendant is a decentralized ecosystem integrating token auctions, NFT minting, and a marketplace, designed with innovative tokenomics and liquidity management. Its key components include:

- **Ascendant Auction**: Conducts daily auctions for distributing Ascendant tokens in exchange for TitanX or ETH deposits.
- **Ascendant NFT Minting**: Enables users to mint NFTs by locking Ascendant tokens, offering tiered rewards and rarities.
- **Ascendant NFT Marketplace**: A decentralized platform for listing, buying, and selling NFTs with payments in ETH or TitanX.

Combining advanced financial mechanisms with user-friendly NFT and token services, Ascendant delivers a comprehensive and dynamic decentralized ecosystem.

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* - c137bf9a4f03bf537471e5bc97f60af5a1808e8b**

Scope

The following folders were in scope of the audit:

- `src/*`

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

- Critical & High: 0 issues
- Medium: 3 issues
- Low & Info: 13 issues

Findings Summary

ID	Title	Severity	Status
[M-01]	Insufficient Observations for TWAP Calculation	Medium	Acknowledged
[M-02]	sqrtPX96 can be manipulated to inflate or deflate pool launch price	Medium	Fixed
[M-03]	ETH item purchases can be DOSed	Medium	Acknowledged
[L-01]	Potential Grief Attack via `collectFees` with Insufficient Fees	Low	Acknowledged
[L-02]	Ascendant uses fixed supply of 1.155 billion instead of 1.055 billion	Low	Fixed
[L-03]	Missing baseURI and tokenURIs for different rarities	Low	Acknowledged
[L-04]	Consider auto-claiming rewards for NFT seller during buy operations	Low	Acknowledged
[L-05]	Default 5-Minute TWAP window is weak	Low	Fixed
[L-06]	It is possible that a small deposit does not receive a rewards	Low	Acknowledged
[L-07]	High default Slippage in `AscendantAuction.sol`	Low	Acknowledged
[I-01]	Leftover liquidity remains locked in `AscendantAuction.sol`	Informational	Acknowledged
[I-02]	Ensure `_newSlippage` is non-zero in `changeSlippageConfig()`	Informational	Fixed
[I-03]	Consider transferring NFT to marketplace first before claiming rewards	Informational	Acknowledged

ID	Title	Severity	Status
[I-04]	Consider pre-incrementing `currentListingId` to avoid non-existent 0 value	Informational	Acknowledged
[I-05]	Unused variables	Informational	Fixed
[I-06]	Inefficient Reward Distribution Model in `AscendantNFTMinting.sol`	Informational	Acknowledged

Detailed Findings

[M-01] Insufficient Observations for TWAP Calculation

Severity

Impact: Medium

Likelihood: Medium

Description

In `Ascendant.sol`, when the Uniswap V3 pool is created, the observation cardinality is set to `100`:

```
    function _createUniswapV3Pool(
        address _titanX,
        address _dragonX,
        address UNISWAP_V3_QUOTER,
        address UNISWAP_V3_POSITION_MANAGER
    ) internal returns (address _pool) {
        address _ascendant = address(this);

        IQuoter quoter = IQuoter(UNISWAP_V3_QUOTER);

        bytes memory path = abi.encodePacked(address(_titanX), POOL_FEE,
address(_dragonX));

        uint256 dragonXAmount = quoter.quoteExactInput(path,
INITIAL_TITAN_X_FOR_LIQ);

        uint256 ascendantAmount = INITIAL_ASCENDANT_FOR_LP;

        (address token0, address token1) = _ascendant < _dragonX ?
(_ascendant, _dragonX) : (_dragonX, _ascendant);

        (uint256 amount0, uint256 amount1) =
            token0 == _dragonX ? (dragonXAmount, ascendantAmount) :
(ascendantAmount, dragonXAmount);

        uint160 sqrtPX96 = uint160((sqrt((amount1 * 1e18) / amount0) * 2
```

```
** 96) / 1e9);

        INonfungiblePositionManager manager =
INonfungiblePositionManager(UNISWAP_V3_POSITION_MANAGER);

        _pool = manager.createAndInitializePoolIfNecessary(token0, token1,
POOL_FEE, sqrtPX96);

193:
IUniswapV3Pool(_pool).increaseObservationCardinalityNext(uint16(100));  <-
--
    }
```

Observations are snapshots of the state of the pool, specifically time-weighted average values such as prices and liquidity. These observations are used to calculate the TWAP.

`100` observations provide good coverage for TWAP calculations, but maybe not enough if the pool is quite active or the lookback period is larger (20-30 minutes).

We can see that the Time-Weighted Average Price (TWAP) lookback period will be between 5 and 30 minutes:

```
    function changeSlippageConfig(address pool, uint224 _newSlippage,
uint32 _newLookBack)
        external
        notAmount0(_newLookBack)
        onlySlippageAdminOrOwner
    {
        require(_newLookBack >= 5 && _newLookBack <= 30,
SwapActions__InvalidLookBack());
        require(_newSlippage <= WAD, SwapActions__InvalidSlippage());

        emit SlippageConfigChanged(pool, _newSlippage, _newLookBack);

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

Suppose the TWAP lookback period is increased to a maximum of 30 minutes. In that case, the 100 observations may not be enough to provide accurate price data for highly active pools, as they might only cover the most recent 10-20 minutes.

Impact

- Insufficient observations could result in inaccurate TWAP calculations, affecting price-sensitive operations.
- If the lookback is set to 30 minutes, but 100 observations only cover the most recent 10-20 minutes due to frequent trades, the TWAP will fail to include the full 30-minute window.

Recommendations

Increasing the observation cardinality to 150-200.

[M-02] sqrtPX96 can be manipulated to inflate or deflate pool launch price

Severity

Impact: Medium

Likelihood: High

Description

The `_createUniswapV3Pool()` function uses the `quoteExactInput()` function to retrieve the dragonXAmount. This is used alongside the ascendantAmount to determine the sqrtPX96 to be used as the initialization price for the dragonX-ascendant pool. The issue with this is that the dragonXAmount returned from the `quoteExactInput()` call is manipulable since it retrieves the amount from a simulated swap (Quoter.sol). Due to this, an attacker can inflate or deflate the sqrtPriceX96 initialization value to be way higher or lower than expected. This makes the launch un-guarded since anyone can control the launch price.

```
uint256 dragonXAmount = quoter.quoteExactInput(path,
INITIAL_TITAN_X_FOR_LIQ);

uint256 ascendantAmount = INITIAL_ASCENDANT_FOR_LP;

(address token0, address token1) = _ascendant < _dragonX ? (_ascendant,
_dragonX) : (_dragonX, _ascendant);

(uint256 amount0, uint256 amount1) =
            token0 == _dragonX ? (dragonXAmount, ascendantAmount) :
(ascendantAmount, dragonXAmount);

uint160 sqrtPX96 = uint160((sqrt((amount1 * 1e18) / amount0) * 2 ** 96) /
1e9);
```

Recommendations

Consider using the TWAP price using OracleLib.sol.

[M-03] ETH item purchases can be DOSed

Severity

Impact: Medium

Likelihood: Medium

Description

Function `buyItemWithETH()` uses the spot price directly (adhering to a certain deviation). If the deviation exceeds the set percentage, we revert. The issue is that an attacker can intentionally manipulate the spot price, forcing the buyer's call to revert due to the deviation. Other than being a DOS issue, an attacker could potentially create a market around this by providing services to specific users to receive listed rare/legendary NFTs as a guarantee.

```
uint256 spotPrice = getSpotPrice();

checkIsDeviationOutOfBounds(spotPrice);
```

Recommendations

Consider using the TWAP price directly instead of the existing spot price and twap price deviation system.

[L-01] Potential Grief Attack via `collectFees` with Insufficient Fees

Description

The `collectFees()` function in the `AscendantAuction.sol` contract is responsible for retrieving accumulated fees from a Uniswap V3 liquidity position.

```
    function collectFees() external returns (uint256 amount0, uint256
amount1) {
        LP memory _lp = lp;

        INonfungiblePositionManager.CollectParams memory params =
INonfungiblePositionManager.CollectParams({
            tokenId: _lp.tokenId,
            recipient: address(this),
            amount0Max: type(uint128).max,
            amount1Max: type(uint128).max
        });
        (amount0, amount1) =
INonfungiblePositionManager(uniswapV3PositionManager).collect(params);

        (uint256 ascendantAmount, uint256 titanXAmount) =
            _lp.isAscendantToken0 ? (amount0, amount1) : (amount1,
amount0);

        titanX.transfer(LIQUIDITY_BONDING, titanXAmount);
```

```
        sendToGenesisWallets(ascendant, ascendantAmount);
    }
```

The collected Ascendant fees are split and distributed to the Genesis Wallets via the `sendToGenesisWallets()` function.

```
    function sendToGenesisWallets(IERC20 erc20Token, uint256 _amount)
private {
        uint256 genesisHalf = wmul(_amount, HALF);

        erc20Token.safeTransfer(GENESIS_WALLET_1, genesisHalf);
        erc20Token.safeTransfer(GENESIS_WALLET_2, genesisHalf);
    }
```

The splitting logic in `sendToGenesisWallets()` uses fixed-point arithmetic (`wmul`) to compute half of the `ascendantAmount`:

```
uint256 genesisHalf = wmul(_amount, HALF); // (_amount * 0.5e18) / 1e18
```

A malicious user could exploit the fact that `collectFees()` has no access control and call the function with a minimal amount of accumulated fees. For very small `_amount` values, genesisHalf results in zero, leading to no funds being sent to the Genesis Wallets.

Impact

For very small fee amounts, `genesisHalf` can round down to zero, resulting in no funds being transferred to the Genesis Wallets.

Recommendations

Require that collected fees exceed a predefined threshold before distribution.

[L-02] Ascendant uses fixed supply of 1.155 billion instead of 1.055 billion

Description

The Ascendant contract mentions that it has a fixed max supply of 1.055B tokens but in reality it uses 1.155B tokens.

```
 *      Features:
 *      - Fixed max supply of 1.055B tokens
```

If we sum up the below mints i.e. 50 million + 100 million + 1 billion + 5 million = 1 billion + 155 million = 1.155 billion tokens.

```
_mint(LIQUIDITY_BONDING, 50_000_000e18); // 50 million
 _mint(msg.sender, INITIAL_TO_ASCENDANT_PRIDE); // 100 million

function emitForAuction() external onlyAuction returns (uint256 emitted) {
        emitted = AUCTION_EMIT;

        _mint(address(auction), emitted); // 100 million every day for 10
days (1 billion in total)
    }

function emitForLp() external onlyAuction returns (uint256 emitted) {
        emitted = INITIAL_ASCENDANT_FOR_LP; // 5 million here

        _mint(address(auction), emitted);
    }
```

Recommendations

Consider updating the comment or the mint amounts appropriately as per what the expected behavior is.

[L-03] Missing baseURI and tokenURIs for different rarities

Description

The AscendantNFTMinting contract inherits the ERC721.sol contract. As we know the minting contract has three different tiers i.e. COMMON, RARE, LEGENDARY. But since the _baseURI() function is not overridden to store a value or there are no separate tokenURIs for each tier, the onchain representation of these NFTs would always be an empty URI.

```
enum Rarity {
    COMMON,
    RARE,
    LEGENDARY
}
```

Recommendations

While this could be handled on the frontend, it is recommended to have onchain URIs that point to the location where the NFT is stored.

[L-04] Consider auto-claiming rewards for NFT seller during buy operations

Description

When an NFT is listed using function listItem(), it claims any pending rewards for the msg.sender, who is the seller. We do not know how long it takes for a NFT to be purchased i.e. from the time at which it was listed to the time it was purchased. During this duration, it is possible for rewards to accumulate. Since the NFT is held in the contract, it does not seem to be clear to whom these rewards should belong i.e. to the seller, buyer, or the contract itself for some other purpose that could be supported.

```
ascendantNFTMinting.claim(_tokenId, msg.sender);
```

Currently, by default the rewards go to the buyer along with the NFT. If this is expected to be the intended behavior, a potential scenario could arise where the seller cancels the listing by frontrunning the buy operation to claim the rewards. Following this, the seller can re-list the NFT again. In this case, the buyer's call fails with the `NftMarketplace__NotListed(_listingId)` error. When the buyer tries to buy the NFT again from the frontend, it will be purchased without any rewards.

Recommendations

Depending on the expected behavior on where these rewards should go, consider auto-claiming them during the buy operations and sending them to the seller.

[L-05] Default 5-Minute TWAP window is weak

Description

The `AscendantNFTMarketplace` contract relies on a default 5-minute TWAP (time-weighted average price) window for price calculations. This window is configured in the `secondsAgo` state variable as follows:

```
uint32 public secondsAgo = 5 * 60;
```

The TWAP is used in the `getTwapPrice()` function to calculate prices based on historical data from Uniswap V3.

```
    function getTwapPrice() public view returns (uint256 quote) {
        uint32 _secondsAgo = secondsAgo;
        uint32 oldestObservation =
OracleLibrary.getOldestObservationSecondsAgo(TITANX_WETH_POOL);
        if (oldestObservation < _secondsAgo) _secondsAgo =
oldestObservation;
```

```
        (int24 arithmeticMeanTick,) =
OracleLibrary.consult(TITANX_WETH_POOL, _secondsAgo);
        uint160 sqrtPriceX96 =
TickMath.getSqrtRatioAtTick(arithmeticMeanTick);

        quote = OracleLibrary.getQuoteForSqrtRatioX96(sqrtPriceX96, WAD,
address(titanX), weth9);
    }
```

While a shorter window allows for faster price updates, it is more susceptible to price manipulation and does not adequately smooth out sudden market fluctuations.

Impact

A shorter TWAP period (5 minutes) may not adequately smooth out sharp price changes caused by short-term trades or manipulation.

Recommendations

Set the default `secondsAgo` to a minimum of 15 minutes to improve price stability:

```
uint32 public secondsAgo = 15 * 60;
```

[L-06] It is possible that a small deposit does not receive a rewards

Description

The function `amountToClaim` in the `AscendantAuction` contract calculates the claimable rewards for a user based on their deposit relative to the total TitanX deposited for a specific day.

```
    function amountToClaim(address _user, uint32 _day) public view returns
(uint256 toClaim) {
        uint256 depositAmount = depositOf[_user][_day];
        DailyStatistic memory stats = dailyStats[_day];

        return (depositAmount * stats.ascendantEmitted) /
stats.titanXDeposited;
    }
```

However, in a edge cases, if a user deposits(`depositAmount`) a very small amount of TitanX, the below calculation can round down to zero due to integer division, causing the user to receive no rewards.:

```
return (depositAmount * stats.ascendantEmitted) / stats.titanXDeposited;
```

Impact

Users with very small deposits might not receive any rewards.

Recommendations

You can implement a minimum deposit threshold.

[L-07] High default Slippage in `AscendantAuction.sol`

Description

In `AscendantAuction.sol`, the `lpSlippage` variable is set to 20%:

```
uint128 lpSlippage = WAD - 0.2e18;
```

This value is relatively high and may lead to suboptimal liquidity additions, potential value loss, and inefficiencies in pool pricing.

With such a high tolerance, liquidity could be added at significantly less favorable rates, leading to potential value loss.

Recommendations

Set a more conservative default slippage tolerance (e.g., 1%-5%). This can still accommodate normal market fluctuations while protecting against significant value loss.

[I-01] Leftover liquidity remains locked in `AscendantAuction.sol`

Description

In function addLiquidityToAscendantDragonXPool(), we approve dragonX and ascendant tokens to the NonfungiblePositionManager contract and create a LP position by calling the function mint(). The issue is that any pending approval is not cleared and any unused tokens are not sent to the owner() address. Due to this, certain amount dragonX and ascendant tokens would be locked permanently in the AscendantAuction.sol contract.

```
(uint256 tokenId,,,) =
INonfungiblePositionManager(uniswapV3PositionManager).mint(params);
```

Recommendations

Retrieve the last two parameters from the mint() function call and withdraw any unused tokens. The example provided in the Uniswap docs can serve as a good reference (here).

[I-02] Ensure `_newSlippage` is non-zero in `changeSlippageConfig()`

Description

Function changeSlippageConfig() only checks the `_newLookBack` parameter to be non-zero but it does not ensure that `_newSlippage` is not 0 as well.

```
    function changeSlippageConfig(address pool, uint224 _newSlippage,
uint32 _newLookBack)
        external
        notAmount0(_newLookBack)
        onlySlippageAdminOrOwner
    {
```

Recommendations

Implement the below fix:

```
    function changeSlippageConfig(address pool, uint224 _newSlippage,
uint32 _newLookBack)
        external
        notAmount0(_newLookBack)
+       notAmount0(_newSlippage)
        onlySlippageAdminOrOwner
    {
```

[I-03] Consider transferring NFT to marketplace first before claiming rewards

Description

In function `listItem()`, we should transfer the tokenId to the contract first before claiming the rewards to msg.sender. This is because the claim() call would require the NFT marketplace contract to be approved by the msg.sender. While this could be supported, it would be better to make the marketplace contract as the owner first by transferring the NFT and then claiming the rewards. This would remove the need for an unnecessary approval.

```
ascendantNFTMinting.claim(_tokenId, msg.sender);

 collection.transferFrom(msg.sender, address(this), _tokenId);
```

Recommendations Switch the order of the statements in the snippet shared above.

[I-04] Consider pre-incrementing `currentListingId` to avoid non-existent 0 value

Description

We should avoid using 0 as the first listingId value since the EnumerableSet activeListings would always return true through contains() for the value 0. While this does not seem to pose a risk in the protocol currently, it should be mitigated.

```
uint256 listingId = currentListingId++;

        listings[listingId] = Listing(_tokenId, _price, msg.sender);
        activeListings.add(listingId);
```

Recommendations

Use ++currentListingId instead.

[I-05]Unused variables

Description

We do not increment `totalAscendantClaimed` in the `claim()` function, `totalETHDeposited` in `depositETH()` and `totalAscendantMinted` in `_updateAuction()` in the `AscendantAuction.sol` contract. The `liquidityAdded` variably in the buy and burn contract is also unused.

Recommendations

Increment `totalAscendantClaimed` and `totalAscendantMinted` in the respective functions. Remove the `totalETHDeposited` variable since it seems to be of no use due to the use of totalTitanXDeposited in the depositETH() function. Remove `liquidityAdded`.

[I-06] Inefficient Reward Distribution Model in `AscendantNFTMinting.sol`

Description

The current reward distribution model in the `AscendantNFTMinting` contract allocates the majority of rewards (50%) to the shortest lock-up period pool (`POOLS.DAY8`).

```
    function _distribute(uint256 amount) internal {
        uint32 currentDay = _getCurrentDay();

        updateRewardsIfNecessary();

        if (currentDay == 1) {
            toDistribute[POOLS.DAY8] += amount;
        } else {
            toDistribute[POOLS.DAY8] += wmul(amount, DAY8POOL_DIST);
            toDistribute[POOLS.DAY28] += wmul(amount, DAY28POOL_DIST);
            toDistribute[POOLS.DAY90] += wmul(amount, DAY90POOL_DIST);
        }
    }
```

```
uint64 constant DAY8POOL_DIST = 0.5e18; // 50%
uint64 constant DAY28POOL_DIST = 0.25e18; // 25%
uint64 constant DAY90POOL_DIST = 0.25e18; // 25%
```

This design contradicts standard DeFi and NFT staking principles, where longer lock-up periods are typically incentivized with higher rewards. As a result, the protocol may inadvertently disincentivize long-term commitments and stability.

Example Scenario

Over time, the 8-day pool becomes disproportionately utilized, while the 28-day and 90-day pools see minimal participation.

Impact

- Users are more likely to stake NFTs for the shortest period (8 days) to maximize rewards, reducing participation in longer-term pools.
- The 28-day and 90-day pools may remain underfunded and unattractive to users, undermining the protocol's goal of encouraging long-term stability.
- By allocating the majority of rewards to short-term lock-ups, the protocol fails to adequately compensate users who commit their NFTs for longer durations.

Recommendations

Consider prioritizing the distribution system in favor of long-term locking.