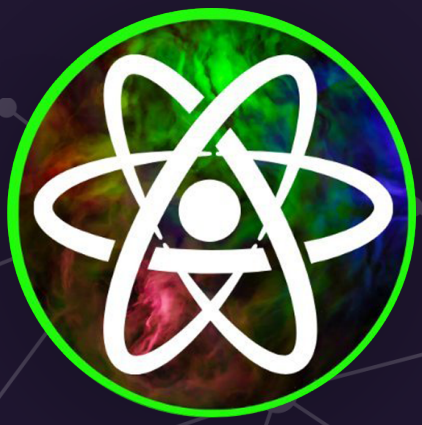
Element280

Security Review



Sep 20, 2024

Conducted by:
Blckhv, Security Researcher
Slavcheww, Security Researcher

Contents

1. About SBSecurity

SBSecurity is a duo of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

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 significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

Element 280 (AKA E280) is an NFT collection and token aimed to be a one stop shop for Titanx users. Element 280 (ELMNT) is an erc20 token bonded with all of the Titanx ecosystem tokens through LP's. By being bonded with all of the ecosystem, the market value of ELMT is a leverage play of all ecosystem tokens combined (including TITANX itself).

Overview

Project	Element280
Repository	Private
Commit Hash	Private
Timeline	September 16 - September 20, 2024

Scope

ElementE280.sol
ElementBuyBurn.sol
ElementHolderVault.sol
ElementNFT.sol
lib/*

Issues Found

Critical Risk	2
High Risk	2
Medium Risk	3
Low/Info Risk	8

5. Findings

5.1. Critical severity

5.1.1. Trading can't be enabled if either one of the liquidity pool already exists

Severity: Critical Risk

Description: `Element280` is non-transferrable until all the E280:ecoToken liquidity pools are deployed. Until then no one, except the `E280` contract itself can move tokens. But there is an issue with the function that is used to deploy the pools:

```
function deployLP(address target) external onlyOwner {
    require(lpPurchaseFinished, "Not all tokens have been purchased");
    uint8 allocation = tokenLpPercent[target];
    require(allocation > 0, "Incorrect target token");
    uint256 amount = lpPool * allocation / 100;
    _deployLiquidityPool(target, amount);
    unchecked {
        totalLPsCreated++;
    }
    if (totalLPsCreated == totalEcosystemTokens) _enableTrading();
}

function _deployLiquidityPool(address tokenAddress, uint256 e280Amount) internal {
    IUniswapV2Factory factory = IUniswapV2Factory(UNISWAP_V2_FACTORY);
    require(factory.getPair(address(this), tokenAddress) == address(0), "Pool already exists");
 ...MORE CODE
}
```

As we can see, if there is an existing pool the deployment will revert because protocol relies on being the first LP supplier. If a malicious user frontruns `E280::deployLP` by executing the UniswapV2Factory::createPair, for a given `target`, `deployLP` will reverting and `totalLPSCreated` will never be equal to the `totalEcosystemToken`, bricking the `Element280`.

Recommendation: Remove the check for the existing pair in `_deployLiquidityPool`, single-sided supply can't happen in an empty pool, so even if the pair creation is frontrunned `E280` will be the first supplier into the pool. Also, in `_deployLiquidityPool` make sure to limit the owner of adding supply only once.

Resolution: Fixed

5.1.2. `HolderVault` uses wrong tokens per cycle allocation

Severity: Critical Risk

Description: `getNextCyclePool` calculates the `Element` tokens that should be reserved for NFTs that haven't been claimed yet wrongly, leaving tokens locked in the `Vault`, without a way to be retrieved.

Instead of adding the `totalRewadsPaid` to the current `E280` balance and then subtracting it from the `totalRewardPool` so we can know how many tokens are not claimed from the past cycles and use only the difference for the new cycle. Currently, the code will subtract both from the `E280` balance, reserving more than needed, which will also make them locked in the `HolderVault`:

```
function updateCycle() external {
    require(block.timestamp > getNextCycleTime(), "Cooldown in progress");
    uint256 cyclePool = getNextCyclePool();
    require(cyclePool > minCyclePool, "Not enough E280 available");
    unchecked {
        currentCycle++;
        uint256 rewardPool = _processCyclePool(cyclePool);
        cycles[currentCycle] = Cycle(block.timestamp, rewardPool /
IElementNFT(E280_NFT).multiplierPool());
        totalRewardPool += rewardPool;
    }
    emit CycleUpdated();
}

function getNextCyclePool() public view returns (uint256) {
    return IERC20(E280).balanceOf(address(this)) - totalRewardPool - totalRewadsPaid;
}
```

Another issue with the current formula is that more tokens will be needed until a new cycle can be started.

Recommendation: Modify the `getNextCyclePool`, so only unclaimed cycle rewards are being reserved:

```
function getNextCyclePool() public view returns (uint256) {
-   return IERC20(E280).balanceOf(address(this)) - totalRewardPool - totalRewadsPaid;
+   return IERC20(E280).balanceOf(address(this)) + totalRewadsPaid - totalRewardPool;
}
```

Resolution: Fixed

5.2. High severity

5.2.1. `UniswapV3` partial swaps can cause wrong accounting

Severity: High Risk

Description: Using the `UniswapV3` pools directly instead of through the router can end up with a partial swap, i.e. not all input token amounts will be swapped. This happens because the code uses `sqrtPriceLimit` as a slippage and deadline check:

```
function _swapUniswapV3Pool(address outputToken, int256 amount, uint160 limit) private returns (uint256) {
    uint160 priceLimit = outputToken > TITANX
        ? uint160(uint256(limit) * (100 - slippage) / 100)
        : uint160(uint256(limit) * (100 + slippage) / 100);
    (int256 amount0, int256 amount1) = IUniswapV3Pool(tokens[outputToken].titanXPool).swap(
        address(this), outputToken > TITANX, amount, priceLimit, abi.encodePacked(outputToken)
    );
    uint256 swappedAmount = outputToken > TITANX
        ? uint256(amount1 >= 0 ? amount1 : -amount1)
        : uint256(amount0 >= 0 ? amount0 : -amount0);
    return swappedAmount;
}
```

`Element280` also has the same function. If swap is executed like that with a limit that is either stale or wrongly passed as an argument, there is a possibility not all the TitanX to be used and to leave some in the pool, because if we take a look in the UniswapV3Pool we will see that when the `sqrtPrice` of the pool is different from the one passed as an argument swap is fulfilled and executed:

```
while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96)
```

This documentation explains why if this approach is used as a slippage the swap won't fail but will be executed partially.

The impact of partial swap can be observed in `ElementBuyBurn::buyAndBurn` and `Element280::purchaseTokenForLp`. Both functions assume that the amount provided will be used at 100% and based on that allocations are decreased:

```
function purchaseTokenForLP(address target, uint160 priceLimit, uint16 slippage) external onlyOwner {
    require(lpPurchaseStarted && !lpPurchaseFinished, "LP phase not active");
    require(lpPurchases[target] < 10, "All purchases have been made for target token");
    uint8 allocation = tokenLpPercent[target];
    require(allocation > 0, "Incorrect target token");
    uint256 amount = lpPool * allocation / 1000;
    totalLpPoolUsed += amount;
    uint256 swappedAmount = _swapTitanXToToken(target, int256(amount), priceLimit, slippage);
    unchecked {
        tokenPool[target] += swappedAmount;
        lpPurchases[target]++;
        // account for rounding error
        if (totalLpPoolUsed >= lpPool - totalEcosystemTokens * 10) lpPurchaseFinished = true;
    }
}
```

In `Element280`, `totalLpPoolUsed` will become an inflated value.

```
function buyAndBurn(address tokenAddress, uint160 priceLimitToken, uint160 priceLimitE280) external {
    if (tokenAddress == TITANX) return _handleTitanXBuyAndBurn(tokenAddress, priceLimitE280);
    EcosystemToken storage token = tokens[tokenAddress];
    require(block.timestamp > token.lastTimestamp + token.interval, "Cooldown in progress");
    IERC20 tokenContract = IERC20(tokenAddress);
    uint256 tokenBalance = tokenContract.balanceOf(address(this));
    (uint256 totalAllocation, bool isNative) = _getNextSwapValue(token, tokenBalance);
    require(totalAllocation > 0, "No allocation available");
    uint256 tokensToDistribute;
    if (isNative) {
        tokensToDistribute = _processIncentiveFee(tokenContract, totalAllocation);
    } else {
        tokenBalance += _swapTitanXToToken(tokenAddress, int256(totalAllocation), priceLimitToken);
        uint256 newTotalAllocation = tokenBalance > token.capPerSwapEco ? token.capPerSwapEco :
tokenBalance;
        tokensToDistribute = _processIncentiveFee(tokenContract, newTotalAllocation);
    }
    (uint256 tokenBurnFee, uint256 tokensToSwap) = _handleTokenDisperse(tokenContract,
tokensToDistribute);
    _handleTokenBurn(tokenAddress, tokenBurnFee);
    _handleE280Swap(tokenAddress, tokensToSwap, priceLimitE280);
    uint256 totalBurned = _handleE280Burn();
    unchecked {
        if (!isNative) {
            totalTitanXUsed += totalAllocation;
            token.titanXAllocation -= totalAllocation;
        }
        token.totalE280Burned += totalBurned;
    }
    token.lastTimestamp = block.timestamp;
    emit BuyBurn(tokenAddress);
}
```

In `ElementBuyBurn`, `totalTitanXUsed` and `titanXAllocation` will be decreased with `totalAllocation`, even though less than was used for the swap. This will leave the remaining `TitanX` unused.

Recommendation: Use the `SwapRouter::exactInputSingle` with `params.sqrtPriceLimitX96 = 0` instead, which always takes the entire amount in tokens, allowing a swap to happen into the entire sqrt range. Also, this function will allow you to estimate the slippage more easily by passing only the desired tokens that you want to receive an effective deadline.

Resolution: Fixed

5.2.2. Malicious user can extract tokens from `buyAndBurn`

Severity: High Risk

Description: User can execute `buyAndBurn` with `Blaze` and pass `priceLimit = 0` which will completely disable the slippage and allow rcn to sandwich himself forcing the titanX → token to be executed in unfavorable conditions for the protocol while the executor will take the profit. The attack will look like this:

1. The user begins with TitanX → Blaze swap, so the price of the TitanX decreases, while Blaze increases

2. `ElementBuyBurn::buyAndBurn` with tokenAddress = `Blaze`, `priceLimitToken` = 0

```
function _swapUniswapV2Pool(address outputToken, int256 amount, uint160 limit) private returns (uint256)
{
    uint256 minAmountOut;
    unchecked {
        minAmountOut = uint256(limit) * (100 - slippage) / 100; // 0 * (100 - 20) / 100 = 0
    }
    IERC20(TITANX).safeIncreaseAllowance(UNISWAP_V2_ROUTER, uint256(amount));

    address[] memory path = new address[](2);
    path[0] = TITANX;
    path[1] = outputToken;

    uint256[] memory amounts = IUniswapV2Router02(UNISWAP_V2_ROUTER).swapExactTokensForTokens(
        uint256(amount), minAmountOut, path, address(this), block.timestamp
    );

    return amounts[1];
}
```

3. Attack ends up with Blaze → TitanX swap, to finish the sandwich attack and realize the profit from the inflated Blaze price.

This can be repeatedly executed for each `buyAndBurn` call, when `isNative = false` and a UniswapV2 swap is needed.

The same sandwich attack can be executed for `E280` pools where the magnitude of the loss will be bigger due to having less liquidity in the pools.

Recommendation: Prevent users from passing the 0 limit for the `UniswapV2` swaps. For `V3` performing this attack will be harder since `TWAP` is used which is harder to manipulate.

Resolution: Fixed

5.3. Medium severity

5.3.1. `E280` will be locked in `HolderVault` if some of the NFTs do not claim their rewards

Severity: Medium Risk

Description: If the Element NFTs are redeemed without harvesting their rewards, tokens will be locked. Nothing can be done to claim these tokens and they will also be excluded from the next cycle's rewards.

Recommendation: It was discussed with the team that calling `Vault::claimRewards` in `ElementNFT::redeemNFTs` can be gas intensive, the solution that we agreed to be applied is to warn the users on the frontend for unclaimed rewards.

Resolution: Acknowledged

5.3.2. Pool swaps use ineffective deadline checks

Severity: Medium Risk

Description: `block.timestamp` is used as a deadline check in the following functions:

- _swapUniswapV2Pool
- _swapWithDragonX
- _handleE280Swap

Using that value will disable the Uniswap checks for transaction staleness and execute the transaction, no matter how long it has passed since it was initiated. Furthermore, slippage will become ineffective because the price can increase while the transaction is being queued, which will still allow to sandwich of the transaction. For example, a swap is executed at a price 1 with a slippage check at 0.99, while tx is in the mempool price moves to 1.05, and the slippage becomes stale as well, allowing everyone to decrease the price to 0.99 and force the `Element280` and `ElementBuyBurn` contracts into bad swap.

Recommendation: Allow the callers to provide the deadline param as a function argument, also wherever Uniswap pools are used, instead of swap routers, make sure to replace them, because routers have built-in checks for the deadline, while pools only rely on the `sqrtPriceLimit` provided to protect both from staleness and slippage.

Resolution: Fixed

5.3.3. Slippage below 1% cannot be specified

Severity: Medium Risk

Description: Minimal slippage for Uniswap allowed currently is 1% because 100% are represented as 100, and most of the swaps will be sandwiched resulting in less tokens received.

The following functions in `Element280` and `ElementBuyBurn` are using that approach:

- _swapUniswapV2Pool
- _swapWithDragonX
- _swapUniswapV3Pool

```
function _swapUniswapV2Pool(address outputToken, int256 amount, uint160 limit) private returns (uint256) {
    uint256 minAmountOut;
    unchecked {
        minAmountOut = uint256(limit) * (100 - slippage) / 100;
}
```

As we can see only values equal or greater than 1 can be passed.

There is a 'hacky' way the slippage to be below 1% but we should adjust the `limit` that we provide as an argument as well, but based on the `H-01` the swap will end up not using all the `amount` passed and will execute partial swap.

Recommendation: Allow passing the `amountOut` desired as an argument, without calculating the slippage in on-chain in the functions, it will be easier for the callers to decide the swap params.

Also, consider using the 10000 BPS approach everywhere, allowing for more granular approach and percentages below 1.

Resolution: Fixed

5.4. Low/Info severity

5.4.1. Slippage protection variables are misleading

Severity: Low Risk

Description: Functions that execute swaps from pools and routers use the same slippage arguments, but they have different meaning in both contracts.

In `_swapUniswapV3Pool`, pool is used and there `limit` will mean what is the desired `sqrtPrice` that the swap will happen with, while in `_swapUniswapV2Pool` and `_swapWithDragonX` `limit` will mean how minimum much tokens out we expect to receive after the swap. All the functions are called from single place `_swapTitanXToToken` and it's easy for the caller to get confused and provide wrong value which can end up executing swap with unfavourable conditions.

Recommendation: Use `SwapRouter`, which will allow the caller to specify how much tokens he wants to receive after the swap, there will be no need to pass `sqrtPriceLimit` and `amountOut` to single argument.

Resolution: Fixed

5.4.2. Swap is executed no matter token balance is enough for `buyAndBurn`

Severity: Low Risk

Description: Off-by-one in the `_getNextSwapValue` will make `isNative = false` and will use the titanXAllocation to perform swap, while in reality contract has sufficient balance for the `buyAndBurn` operation:

```
function _getNextSwapValue(EcosystemToken memory token, uint256 tokenBalance)
    internal
    pure
    returns (uint256 tokensToSwap, bool isNative)
{
    require(token.capPerSwapTitanX > 0, "Token is disabled");
    isNative = token.capPerSwapEco < tokenBalance;
    tokensToSwap = isNative
        ? token.capPerSwapEco
        : token.titanXAllocation > token.capPerSwapTitanX ? token.capPerSwapTitanX :
token.titanXAllocation;
}

function buyAndBurn(address tokenAddress, uint160 priceLimitToken, uint160 priceLimitE280) external {
...
  uint256 tokenBalance = tokenContract.balanceOf(address(this));
  (uint256 totalAllocation, bool isNative) = _getNextSwapValue(token, tokenBalance);
...
  if (isNative) {
      tokensToDistribute = _processIncentiveFee(tokenContract, totalAllocation);
  } else {
      tokenBalance += _swapTitanXToToken(tokenAddress, int256(totalAllocation), priceLimitToken);
      uint256 newTotalAllocation = tokenBalance > token.capPerSwapEco ? token.capPerSwapEco :
tokenBalance;
      tokensToDistribute = _processIncentiveFee(tokenContract, newTotalAllocation);
  }
```

For example, if the `capPerSwapEco = 100` and `tokenBalance = 100`, it should be sufficient for the b&b operation without using the allocated `TitanX`.

Recommendation: Modify the check so no TitanX is used if the eco token balance is sufficient.

Resolution: Fixed

5.4.3. ERC20 safe operations are not used

Severity: Low Risk

Description: All contracts extend the `IERC20` with the `SafeERC20` library but do not use the `safeTransfer` and `safeTransferFrom` operations in none of the places. Although all the tokens that will be used are limited, safe operations will handle all the cases that can occur from EIP20 incompatible tokens.

Recommendation: Use the safe transfer operations in all the contracts.

Resolution: Fixed

5.4.4. Approved address receives the redeem fees, not the owner

Severity: Low Risk

Description: When ElementNFTs are redeemed `msg.sender` receives `97%` of the buy price in `E280` tokens, but this sender is not guaranteed to be the **owner**, and if someone approved by him executes `ElementNFT::redeemNFTs`, the executor will get the tokens instead:

```
function redeemNFTs(uint256[] calldata tokenIds) external {
    ...MORE CODE
    IElement280(E280).handleRedeem(totalAllocation, msg.sender);
}
```

Recommendation: Pass the owner of the tokens for the allocation instead of the `msg.sender` and also make sure the owner of all the tokenIds is the same person, otherwise, all the fees will go to the first one:

Resolution: Fixed

5.4.5. Missing duplicate ecosystem tokens check in constructors

Severity: Low Risk

Description: There is nothing that can prevent the protocol deployer from passing the same ecosystem token twice in the constructors of `Element280` and `ElementBuyBurn`. If duplicates happen, can lead to mistake and miscalculations in the contracts.

Recommendation: Add checks to prevent from adding duplicate ecosystem tokens.

Resolution: Fixed

5.4.6. First loop in _getNFTFirstCycle is unnecessary

Severity: Information Risk

Description: Cycles in `ElementHolderVault` always start from 1, in 0 cycles there will be no rewards and the code should not start the check for the first NFT cycle from 0:

```
function _getNFTFirstCycle(uint256 nftTimestamp) internal view returns (uint256) {
    for (uint256 i = 0; i <= currentCycle; i++) {
        if (cycles[i].timestamp > nftTimestamp) return i;
    }
    return currentCycle + 1;
}
```

Currently, for each token, there will 1 empty iteration.

Recommendation: Start the for cycle from 1, instead of 0.

Resolution: Fixed

5.4.7. Storage variables shadowing in ElementNFT

Severity: Information Risk

Description: Some of the storage variables of both `Ownable` and `ERC721A` are being shadowed in the inheriting `ElementNFT` contract. That includes:

- `owner` in `getBatchedTokensData`

- `totalSupply` in `tokenIdsOf` and `getTotalNftsPerTiers`

While this **can't lead** to overriding the values assigned to the storage variables it will confuse the users that decide to skim over the contracts.

Recommendation: If possible, try to use other variable names in the described functions.

Resolution: Fixed

5.4.8. `startLpPurchases` cannot be called at `presaleEnd`

Severity: Information Risk

Description: presaleEnd is the time at which the presale ends, this second and after that second is not in a `presale`.

```
function isPresaleActive() public view returns (bool) {
    return presaleEnd > block.timestamp;
}
```

The following operation is `startLpPurchases` but it cannot be called at time = presaleEnd.

```
function startLpPurchases() external onlyOwner {
    require(presaleEnd != 0, "Presale not started yet");
    require(!lpPurchaseStarted, "LP creation already started");
    uint256 availableBalance = IERC20(TITANX).balanceOf(address(this));
    require(availableBalance > 0, "No TitanX available");
    if (availableBalance < LP_POOL_SIZE) {
        require(block.timestamp > presaleEnd, "Presale not finished yet");
        _registerLPPool(availableBalance);
    } else {
        _registerLPPool(LP_POOL_SIZE);
    }
    lpPurchaseStarted = true;
}
```

Recommendation: In one of the functions, the check must also have =.

Resolution: Fixed