



# Element280

## Security review

Version 1.0

Reviewed by  
**nmirchev8**  
**deth**

## Table of Contents

<b>1</b>	<b>About Egis Security</b>	<b>3</b>
<b>2</b>	<b>Disclaimer</b>	<b>3</b>
<b>3</b>	<b>Risk classification</b>	<b>3</b>
3.1	Impact . . . . .	3
3.2	Likelihood . . . . .	3
3.3	Actions required by severity level . . . . .	3
<b>4</b>	<b>Executive summary</b>	<b>4</b>
<b>5</b>	<b>Findings</b>	<b>5</b>
5.1	Critical risk . . . . .	5
5.1.1	Element280::_deployLiquidityPool() - Contract can be DoSed if someone donates to the pair . . . . .	5
5.2	High risk . . . . .	6
5.2.1	No slippage protection when providing liquidity in ElementLiquidityManager can result in sandwich attacks . . . . .	6
5.3	Medium risk . . . . .	7
5.3.1	No slippage protection ElementBuyBurn could result in sandwich attacks . . . . .	7
5.3.2	Stepwise jump farming in ElementHolderVault . . . . .	8
5.4	Low risk . . . . .	9
5.4.1	ElementBuyBurn::isRebalanceAvailable may return wrong value . . . . .	9
5.4.2	Same token ids may be used in ElementHolderVault::getRewards . . . . .	9
5.4.3	Consider emitting events on important state changes . . . . .	9
5.5	Informational . . . . .	10
5.5.1	DevDistribute::minAmount cannot be updated and it is set to only 100 . . . . .	10
5.5.2	First check in onlyNftContract is redundant . . . . .	10
5.5.3	If DevDistribute::changeWallet is called with existing dev wallet, some tokens will be left in the contract after distribution. . . . .	10
5.5.4	ElementNFT::calculateAllocation can be called with burned nfts, or duplicate ids . . . . .	11
5.5.5	Set lpPurchases[TITANX] to 5 in _registerLPPool . . . . .	11
5.5.6	Rounding down in calculating tokensPerMultiplier results in small amounts of locked tokens . . . . .	11
5.5.7	Claim rewards design may be used for honey-pot exploits . . . . .	12
5.5.8	In ElementBuyBurn::buyAndBurn we will use at most 80% of capPerSwapEco when swapping the token . . . . .	12
5.5.9	require(HOLDER_VAULT != address(0)) is redundant and it is guaranteed if above line has passed . . . . .	12
5.5.10	Some ecosystem tokens in constants.sol has address(0) . . . . .	13
5.5.11	Rebalance interval check in isRebalanceAvailable is slightly off . . . . .	13
5.5.12	Typo in _userPurchases mapping . . . . .	13
5.5.13	Wrong natspec comment . . . . .	14

## 1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding \$170,000. They have identified over 150 high and medium-severity vulnerabilities in both public contests and private audits.

## 2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

## 3 Risk classification

Severity	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** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

### 3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

### 3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

## 4 Executive summary

### Overview

Project Name	Element280
Repository	<a href="https://github.com/DudeGuy420/Element280-Contracts">https://github.com/DudeGuy420/Element280-Contracts</a>
Commit hash	9e248629d57db7099e71f0b69f001b4535821121
Resolution	e496519562057f4157bf1c04350b87f24d447153
Documentation	<a href="https://docs.helios-hlx.win/element280">https://docs.helios-hlx.win/element280</a>
Methods	Manual review

### Scope

contracts/lib/constants.sol
contracts/DevDistribute.sol
contracts/Element280.sol
contracts/ElementBuyBurn.sol
contracts/ElementHolderVault.sol
contracts/ElementLiquidityManager.sol
contracts/ElementNFT.sol

### Issues Found

Critical risk	1
High risk	1
Medium risk	2
Low risk	3
Informational	13

## 5 Findings

### 5.1 Critical risk

#### 5.1.1 Element280::\_deployLiquidityPool() - Contract can be DoSed if someone donates to the pair

**Severity:** *Critical risk*

**Context:** Element280.sol#L382-L391

**Description:** When we try to deploy a LP, we call Uni's addLiquidity, which will try to create a pair if it doesn't exist, but anyone can create a pair beforehand, which wouldn't be a problem in most cases except one.

If someone creates a pair between E280 and Flux for example, then transfers a bit of Flux directly to the pair then calls sync on the pair, this will increase the reserves of Flux, but not for E280, so the pair will now have `reservesE280 == 0` and `reservesFlux != 0`.

This is very problematic, as when addLiquidity is called, it calls the internal function `_addLiquidity`, which has several cases in it. `UniswapRouter#_addLiquidity` function Because the reserves are the pair are `reserveA == 0(E280)` and `reserveB != 0(Flux)`, we'll go into the else statement. Then we call `UniswapV2Library.quote`.

The tx will revert on the second **require**, as `reserveA == 0(E280)`.

Because of this attack, `_enableTrading` will be impossible to call, thus it will make the token completely untransferable.

**Recommendation:** Fetch the corresponding univ2 pool and check if the other token `balance > 0`. If so, calculate e280 token amount that should be send to the pair to make the ratio correct. Send it to the pair and call `sync` on the pool. Then call `addLiquidity`, but be sure to calculate `amountAmin` and `amountBmin` correspondingly.

**Resolution:** Fixed

## 5.2 High risk

### 5.2.1 No slippage protection when providing liquidity in `ElementLiquidityManager` can result in sandwich attacks

**Severity:** *High risk*

**Context:** `ElementLiquidityManager.sol`#L63-L64

**Description:** Under certain circumstances actors with large capital can sandwich a call to `ElementLiquidityManager#addLiquidity` to extract value from the manipulated changes in the reserves. NOTE that the following issue also exists in `removeLiquidity`

**Recommendation:** Consider adding `minAmountA` & `minAmountB` params and pass them to `addLiquidity` function.

**Resolution:** Acknowledged

## 5.3 Medium risk

### 5.3.1 No slippage protection EElementBuyBurn could result in sandwich attacks

**Severity:** *Medium risk*

**Context:** ElementBuyBurn.sol#L401

**Description:** When swapping weth -> titanX in rebalance there is hardcoded `amountMin = 0`, which means that if there is ether to be swapped and exploiter has enough liquidity, he can manipulate the pool, call the func and extract value from the slippage.

On `buyAndBurn` user provides `minTokenAmount` and `minE280Amount`. Those can be set to 0/1 and user can extract value if the swapped amount is large enough, or reserves in the pool are low. This may also brick protocol tokenomics.

**Recommendation:** Consider using a twap to dynamically calculate the slippage tolerance.

**Resolution:** Acknowledged

### 5.3.2 Stepwise jump farming in ElementHolderVault

**Severity:** *Medium risk*

**Context:** ElementHolderVault.sol#L85

**Description:** Mint nft right before cycle update and then redeem it after 48 hours. Do the same for the next cycle update, etc. The user claims rewards accrued for 7 days, when he has held the nft for only 2 days.

**Recommendation:** Consider forcing user to be able to redeem nft after at least 7 days

**Resolution:** Acknowledged



## 5.4 Low risk

### 5.4.1 ElementBuyBurn::isRebalanceAvailable may return wrong value

**Severity:** *Low risk*

**Context:** ElementBuyBurn.sol#L135-L136

**Description:** When we call `rebalance` we do eth swaps and then we check if the `unaccounted titan`  $> 0$ . But when we fetch `unaccountedTitan` we check if the value is  $> \text{minTitanX}$ :

```
uint256 titanBalance = IERC20(TITANX).balanceOf(address(this));
unchecked {
    // titanX that we have swapped // titanX that
    // have already been allocated for tokens
    unaccountedTitan = titanBalance + totalTitanXUsed - totalTitanXAllocated
    ;
}
if (unaccountedTitan < minTitanX) return 0;
return unaccountedTitan;
```

So there may be the situation, when `isRebalanceAvailable` will return true, because `address(this).balance`  $> 0$ , but a call to `rebalance` would revert, because of the `minTitanX` check.

**Recommendation:** Don't revert `rebalance` call, if `unaccounted titan`  $= 0$ .

**Resolution:** Fixed

### 5.4.2 Same token ids may be used in ElementHolderVault::getRewards

**Severity:** *Low risk*

**Context:** ElementHolderVault.sol#L129

**Description:** If the view function `getRewards` is used by an outside integrator, it may manipulate the returned value by providing same `tokenId`, which will lead to manipulated `totalReward` value.

**Recommendation:** Consider reverting if there are duplicate `tokenIds` in the `tokenIds` array.

**Resolution:** Fixed

### 5.4.3 Consider emitting events on important state changes

**Severity:** *Low risk*

**Context:** Everywhere

**Description:** Consider emitting events on important state changes in the following functions:

- In `Element280.sol`:
  - `_enableTrading`
  - `setProtocolAddresses`
  - `setWhitelistTo`
  - `setWhitelistFrom`
- In `ElementHolderVault.sol`:

- `setMinCyclePool`
  - `setTreasury`
- In `ElementBuyBurn.sol`
  - `setTreasury`
  - `setIncentiveFee`
  - `setEthCapPerSwap`
  - `setRebalanceInterval`
  - `setTokenInterval`
  - `setTokenCapPerSwap`
- In `ElementNFT`:
  - `setContractURI`
  - `setBaseURI`

**Recommendation:** Emit event on important state changes.

**Resolution:** Partially Fixed

## 5.5 Informational

### 5.5.1 `DevDistribute::minAmount` cannot be updated and it is set to only 100

**Severity:** *Informational*

**Context:** `DevDistribute.sol`#L12-L13

**Description:**

`DevDistribute::minAmount` cannot be updated and it is set to only 100. Consider making it constant, or implementing a setter.

**Resolution:** Acknowledged

### 5.5.2 First check in `onlyNftContract` is redundant

**Severity:** *Informational*

**Context:** `Element280.sol`#L93-L94

**Description:**

First check is redundant. `msg.sender` cannot be `address(0)` anyways.

**Resolution:** Fixed

### 5.5.3 If `DevDistribute::changeWallet` is called with existing dev wallet, some tokens will be left in the contract after distribution.

**Severity:** *Informational*

**Context:** `DevDistribute.sol`#L39-L43

**Description:**

```
function changeWallet(address newWallet) external {
    require(isTeamWallet(msg.sender), "Unauthorized");
    _teamWallets.remove(msg.sender);
    _teamWallets.add(newWallet);
}
```

`_teamWallets` is `EnumerableSet`, which means that if we try to add an element, which is already existent, we skip the operation. As a result we will have `_teamWallets.length = 2`, but `distributeToken` always divide the balance by 3:

```
uint256 share = availableBalance / 3;
for (uint256 j = 0; j < _teamWallets.length(); j++) {
    token.safeTransfer(_teamWallets.at(j), share);
}
```

Consider checking if the `newWallet` is already in the set.

**Resolution:** Fixed

#### 5.5.4 ElementNFT::calculateAllocation can be called with burned nfts, or duplicate ids

**Severity:** *Informational*

**Context:** ElementNFT.sol#L174

**Description:**

Consider not counting the allocation of nfts, that have already been burned. Also consider checking for duplicate ids, which would otherwise return manipulated data.

**Resolution:** Fixed.

#### 5.5.5 Set lpPurchases[TITANX] to 5 in \_registerLPPool

**Severity:** *Informational*

**Context:** Element280.sol#L367-L368

**Description:**

Since we now only have 5 purchases for LP tokens, inside `_registerLPPool` which is a function used only for TitanX we still set `lpPurchases[TITANX] = 10`, while now there are only 5 purchases. To be completely accurate, `lpPurchases[TITANX]` should be set to 5, so it mirrors the logic that applies for all other tokens.

**Resolution:** Fixed.

#### 5.5.6 Rounding down in calculating tokensPerMultiplier results in small amounts of locked tokens

**Severity:** *Informational*

**Context:** ElementHolderVault.sol#L84-L86

**Description:**

Here is an example: `multiplierPool = 1100; rewardPool = 12341234 => tokensPerMultiplier = 12341234 / 1100 = 11219.30 => 11219 11219 * 1100 = 12340900 (claimable amount) => 12341234 - 12340900 = 334` is accounted as claimed, but it won't and will continue increasing for the next cycles.

Consider increasing `totalRewardPool` to the value after the rounding. This way we will use tokens, which we haven't accounted for on the next cycle:

```
totalRewardPool += cycles[currentCycle].tokensPerMultiplier * IElementNFT(E280_NFT)
    .multiplierPool();
```

**Resolution:** Fixed

**5.5.7 Claim rewards design may be used for honey-pot exploits**

**Severity:** *Informational*

**Context:** `ElementHolderVault`

**Description:**

Imagine a scenario where Bob has listed his E280 Nft in Opensea and it is Tier 6. Let's assume Tier 6 NFT can be redeemed for \$100, but additionally holds \$50 in rewards from `ElementHolderVault`. Bob has listed it for \$125 and instantly a victim submit a purchase order (Instant money - you buy \$150 for \$125). Then Bob front-runs victim's transaction with `ElementHolderVault#claimRewards`, so the buyer receives \$100 worth NFT for \$125. Consider implementing `claimRewardRequest`, which makes you claim with a little cooldown. Or you can just document potential frauds from 3rd parties.

**Resolution:** Acknowledged

**5.5.8 In ElementBuyBurn : buyAndBurn we will use at most 80% of capPerSwapEco when swapping the token**

**Severity:** *Informational*

**Context:** `ElementBuyBurn.sol#L169-L175`

**Description:**

`capPerSwapEco` limits the pool swaps and prevents potential sandwich attacks or big slippage. In current implementation we first: - limit the token allocation to `capPerSwapEco` (if allocation is greater) - calculate and send incentive fee - calculate, send and burn `tokenDisperse`, which is 20% from the `capPerSwapEco - incentive fee` - and then swap `(capPerSwapEco - incentiveFee) * 0.8`

Which means that the value that we swap is much smaller than `capPerSwapEco`.

**Resolution:** Acknowledged

**5.5.9 require(HOLDER\_VAULT != address(0)) is redundant and it is guaranteed if above line has passed**

**Severity:** *Informational*

**Context:** Element280.sol#L178-L179

**Description:**

We set `E280NFT` and `HOLDER_VAULT` in `setProtocolAddresses` function, where we do `address(0)` check for each of the arguments. As a result if `E280NFT != address(0)` guarantees that `HOLDER_VAULT` is also `!= address(0)`

**Resolution:** Fixed

#### 5.5.10 Some ecosystem tokens in constants.sol has address(0)

**Severity:** *Informational*

**Context:** constants.sol#L20-L26

**Resolution:** Acknowledged

#### 5.5.11 Rebalance interval check in isRebalanceAvailable is slightly off

**Severity:** *Informational*

**Context:** ElementBuyBurn.sol#L242

**Description:**

In `rebalance` we can only call the func if `block.timestamp` is larger than `token.lastTimestamp + token.interval`

```
require(block.timestamp > token.lastTimestamp + token.interval, "Cooldown in progress");
```

In `isRebalanceAvailable` it can return true if `block.timestamp` is larger than or equal to `token.lastTimestamp + token.interval`

```
if (block.timestamp < lastRebalance + rebalanceInterval) return false;
```

**Resolution:** Fixed

#### 5.5.12 Typo in \_userPurchases mapping

**Severity:** *Informational*

**Context:** Element280.sol#L81

**Description:**

Element280's `_userPurchases` mapping is defined as: `mapping(address token => EnumerableSet.UintSet) private _userPurchases`; While in reality token are user addresses, so it should be like so: `mapping(address user => EnumerableSet.UintSet) private _userPurchases`;

**Resolution:** Fixed

**5.5.13 Wrong natspec comment****Severity:** *Informational***Context:** Element280.sol#L78-L79**Description:**

Consider updating the comment about `Element280::lpPurchases` to follow the updated version, which makes 5 purchases, instead of 10.

**Resolution:** Fixed