Scale

Security review

Version 1.0

Reviewed by

nmirchev8
deth

Table of Contents

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	Scale
Repository	Private
Commit hash	6938547c66fac25444e96dc352a3618df952f29a
Resolution	30b5ccbe1435e651c75ed212fd0a58086eb7602e
Documentation	https://zibars-organization.gitbook.io/scale
Methods	Manual review

Scope

contracts/SCALE.sol
contracts/SHED.sol
contracts/SHEDMiner.sol
contracts/lib/*

Issues Found

Critical risk	0
High risk	2
Medium risk	1
Low risk	7
Informational	6

5 Findings

5.1 High risk

5.1.1 SCALE::_processIncentiveFee() - If receiver is excluded, he receives the entire amount

Severity: *High risk*

Context: SCALE.sol#L522-L523

Description: When someone calls `distributeReserve` we use `_processIncentiveFee` to give a small percentage (0.3%) of the SCALE that is left in `address(this)` to the caller `msg.sender`

```
function distributeReserve(uint256 minDragonXAmount, uint256 deadline) external {
        if (!tradingEnabled) revert TradingDisabled();
        //@info this will use up all the remaining SCALE that address(this) has
        uint256 balance = balanceOf(address(this));
        if (balance < minReserveDistribution) revert InsuffucientBalance();
        _processIncentiveFee(msg.sender, balance);
```

We use the entire `balance` of the contract when we call this function:

```
function _processIncentiveFee(address receiver, uint256 amount) internal {
        uint256 rValue = reflectionFromToken(amount);
        uint256 rIncentive = FullMath.mulDiv(rValue, incentiveFee, 10000);
        _rOwned[address(this)] -= rIncentive;
        _rOwned[receiver] += rIncentive;
        if (_isExcludedFromReflections[receiver]) _tOwned[receiver] += amount;
    }
```

We then turn the `balance` to `rValue` and then get the `rIncentive` and we add it to the `receiver`, but if he is excluded from reflection we also increase his `_tOwned`.

The problem is we increase it by `amount`, which is the entire `balanceOf(address(this))`, not the 0.3% fee.

Recommendation: Calculate `incentiveFee` based off `amount` and then add it to `_tOwned`.

Resolution: Fixed

5.1.2 `SCALE::getCurrentMinerParams()` - `share` calculation is incorrect

Severity: *High risk*

Context: SHED.sol#L278-L279

Description: `_availableMinerTypes.length - totalActiveMiners`; should be in paranthesis, otherwise the subtraction is executed after the division, which is not the expected calculation.

```
uint256 share = availableAmount / _availableMinerTypes.length - totalActiveMiners;
```

Recommendation: Introduce paranthesis for `(_availableMinerTypes.length - totalActiveMiners)`

Resolution: Fixed

5.2 Medium risk

5.2.1 User can self-sandwich `SCALE::distributeReserve()` to extract value

Severity: *Medium risk*

Context: SCALE.sol#L221-L222

Description: The function is public, anyone can call it with `minDragonXAmount = 0`, which is 0 slippage.

```
function distributeReserve(uint256 minDragonXAmount, uint256 deadline) external {
        if (!tradingEnabled) revert TradingDisabled();
        uint256 balance = balanceOf(address(this));
        if (balance < minReserveDistribution) revert InsuffucientBalance();
        _processIncentiveFee(msg.sender, balance);
        uint256 buyBurnShare = balanceOf(address(this)) / 2;
        _swapScaleToDragonX(buyBurnShare, minDragonXAmount, deadline);
        uint256 quarter = balanceOf(address(this)) / 2;
        uint256 rTransferAmount = reflectionFromToken(quarter);
        _rOwned[address(this)] -= rTransferAmount;
        _rOwned[marketingWallet] += rTransferAmount;
        _rBurn(address(this), quarter);
    }
```

Note there is no swap cap here, which means that if we have accrued enough `scale` tokens to be swapped, user may use flashloan to manipulate `scale/dragonX` pool, which will result in receiving 0 dragonX tokens from the swap and extracting those scale tokens, after the back-run swap of the exploiter. The following also result in violated tokenomics, because we won't be sending the predefined `dragonX` tokens to `bdxBuyAndBurn`

Recommendation: Consider implementing `swapCap`, which will regulate the swap amount. If the swapped amount is low enough, it may not be beneficial for the exploiter to execute the attack.

Resolution: Fixed

5.3 Low risk

5.3.1 Under some circumstances address that is being included will instantly increase it's balance

Severity: *Low risk*

Context: SCALE.sol#L360-L362

Description: `SCALE` system uses token reflection to insentivise holders. There is also functionality to add/remove addresses exposed to the reflections. Critical function for the contract is `_getCurrentSupply`, which accounts for two edge cases:

```
    function _getCurrentSupply() private view returns (uint256, uint256) {
        uint256 rSupply = _rTotal;
        uint256 tSupply = _tTotal;
        for (uint256 i = 0; i < _excluded.length; i++) {
            address account = _excluded[i];
            uint256 rValue = _rOwned[account];
            uint256 tValue = _tOwned[account];
            if (rValue > rSupply || tValue > tSupply) return (_rTotal, _tTotal);
            rSupply -= rValue;
            tSupply -= tValue;
        }
        if (rSupply < _rTotal / _tTotal) return (_rTotal, _tTotal);
        return (rSupply, tSupply);
    }
```

If either of `if` checks succeeds when owner is including account to reflections, the same account balance is instantly increased. This is due to decrease of `_rTotal` in `includeAccountToReflections`:

```
    function includeAccountToReflections(address account) public onlyOwner {
        if (!_isExcludedFromReflections[account]) revert ExcludedAddress();
        uint256 difference = _rOwned[account] - (_getRate() * _tOwned[account]);
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _rOwned[account] -= difference;
                _rTotal -= difference;
...
```

EXAMPLE

- Imagine we have `_rTotal = 1000` and `_tTotal = 10`
- Bob has balance of `_tOwned = 1` `_rOwned = 100`
- Some reflections are being recorded and `_rTotal` becomes 990
- There are many excluded addresses and `_getCurrentSupply` return the default `990, 10`, because `(rSupply < _rTotal / _tTotal)`
- Bob diff = 100 - 99 = 1
- `_rOwned[Bob] = 99`; `_rTotal = 989`
- `balanceOf[Bob]` now is `99 / 98,9 = 1,001011122346`, which is `0.001011122346` instant increase. **NOTE** that for simplicity the example is with very small amounts and that's why we work with decimals. In reality balance increase may be: 10000000000000 => 10010111223460

Resolution: Acknowledged

5.3.2 TitanX donation can brick `finalizePresale` and lock all titanX funds

Severity: *Low risk*

Context: `Lotus.sol#L100`

Description: `finalizePresale` is used to end the presale, distribute the TitanX and do some accounting.

```
function finalizePresale() external onlyOwner {
        if (presaleEnd == 0) revert PresaleInactive();
        if (isPresaleActive()) revert PresaleActive();
        if (shedContract == address(0)) revert ZeroAddress();
        if (presaleFinalized) revert Prohibited();

        _distributeTokens();

        // burn not minted
        uint256 tBurn = _tTotal - _totalMinted - scaleLpPool;
        uint256 rBurn = tBurn * _getRate();
        _rOwned[address(this)] -= rBurn;
        _rTotal -= rBurn;
        _tTotal = _totalMinted + scaleLpPool;

        presaleFinalized = true;
        emit Transfer(address(0), address(this), scaleLpPool);
    }
```

The line we'll focus on is:

```
 uint256 tBurn = _tTotal - _totalMinted - scaleLpPool;
```

The main logic here is that this won't underflow because `_totalMinted + scaleLpPool = _tTotal`, this is enforced with this check in `minWithETH` and `mintWithTitanX`.

```
        if ((_totalMinted + amount) * 135 / 100 > _tTotal) revert MaxSupply(); //@ok
```

The values for `_tTotal = 100` will be: `_tTotal = 100` `_totalMinted = 74` `scaleLpPool = 35`

This is because `_totalMinted` is limited, if we calculate the above statement we get:

`74 * 135 / 100 = 99.9 (~100)`, so that's our `_totalMinted` max.

The issue lies in how `scaleLpPool` is calculated:

```
uint256 availableTitanX = titanX.balanceOf(address(this)); //@ok
        titanLpPool = availableTitanX * LP_POOL_PERCENT / 100; //@ok
        scaleLpPool = titanLpPool / 10 ** 9;
```

It's calculated based of the TitanX balance of the contract, which can be easily manipulated by just donating TitanX.

If we do this, we'll technically have more SCALE "minted" than `_tTotal`, which will revert the tx here:

```
uint256 tBurn = _tTotal - _totalMinted - scaleLpPool;
```

Example: I'll exclude the decimals for a simplified example.

`_tTotal = 100 _totalMinted = 74 titanx.balanceOf(address(this))= 74`

Calculating for `scaleLpPool = 74 * 35 / 100 = 25`

`uint256 tBurn = 100 - 74 - 25 = 1`

Now if we donate 100 TitanX to the contract we get: Calculating for `scaleLpPool = 174 * 35 / 100 = 60` `uint256 tBurn = 100 - 74 - 60 = panic underflow`

If this happens the protocol is permanently bricked,`finalizePresale` cannot be called, the issue can't be fixed with `claimDust`, because it can't be called until `tradingEnabled = false` and `buyBurnPurchases != purchasesRequired`, which both of the above rely on `finalizePresale` to be called first.

However, the likelihood of such event happening is very low, because we should have used almost all titanX supply to buy Scale during the presale, which is worth more than $50M at this moment.

Using `balanceOf` is a bad practices as it can lead to these types of manipulation attacks.

We recommend using `_totalMinted` for the calculations, as it holds how much SCALE was minted and since it's minted 1:1 with TItanX it can be used in `_distributeTokens`. It needs to be scaled to 1e18 first (the decimals of TitanX) for the calculations to be correct.

Recommendation: Using `balanceOf` is a bad practices as it can lead to these types of manipulation attacks.

We recommend using `_totalMinted` for the calculations, as it holds how much SCALE was minted and since it's minted 1:1 with TItanX it can be used in `_distributeTokens`. It needs to be scaled to 1e18 first (the decimals of TitanX) for the calculations to be correct.

Resolution: Acknowledged

5.3.3 SCALE::_fixPool() - `_rTotal` can overflow

Severity: *Low risk*

Context: SCALE.sol#L597-L599

Description: If we have minted almost all of the possible SCALE, we will burn very small `rBurn` from `_rTotal`. The variable is already very close to `type(uint256).max`, so inside `_fixPool` if we increase the value now it is possible to overflow.

To give a very simplified example, we'll use the following:

1. All possible SCALE is minted and when we call `finalizePresale` we get `tBurn = 0`, so we don't have to burn anything.
2. `_rTotal = 1157920892373161954235709850086879078532699846656405640000000000000000000000000000` which is only `3945758400791312963993 5` away from `uint256.max`.
3. We try to add liquidity to a pool and we enter `_fixPool`, for simplicity let's assume `requiredScale = 1` and `_getRate = 115792089237316195423570985008687907853269984665640564`

4. For `rAmount` we get `1 * 115792089237316195423570985008687907853269984665640564`.

5. We then add it to `_rTotal` the tx reverts with an overflow.

This is very low likelihood, as a massive amount of SCALE has to be minted so we burn a very small amount from it in `finalizePresale` and then someone needs to donate a proportional amount of tokens directly to one of the pools in order for this to happen. This can be applied to all 5 pools so the attack surface area is a larger and if this happens then the pool cannot be created and trading cannot be enabled, which bricks the protocol.

Recommendation: The best way to fix this would be to leave some % of tokens unmintable which can stay in reserve just in case they are needed for `_fixPool`

Resolution: Acknowledged

5.3.4 Consider emitting events on important state changes

Severity: *Low risk*

Context: Everywhere

Description: LOCATIONS:

- `SHED.sol`:
  - `distributeReserveFund`
  - `setAvailableMinerTypes`
  - `setClaimPercentages`
  - `setDeviation`
  - `activateSHED`
- `SCALE.sol`:
  - `setPurchasesRequired`
  - `setReflectionFee`
  - `setIncentiveFee`
  - `setMinReserveDistribution`
  - `includeAccountToReflections`
  - `excludeAccountFromReflections`

Recommendation: Emit event on important state changes.

Resolution: Fixed

5.3.5 Consider implementing limit to `_excluded` array

Severity: *Low risk*

Context: SCALE.sol#L347

Description: Because we are looping trough the array on each function of the contract -> inside `_getCurrentSupply`, if many addresses are excluded, there is a risk to hit OOG revert

Recommendation: Introduce a max limit of excluded addresses.

Resolution: Fixed

5.3.6 SHED - some minters may be unclaimable

Severity: *Low risk*

Context: SHED.sol#L128-L132

Description:

`Hydra` project has a limit of 1000 minters per address and that's why scale team has implemented functionality to deploy new `SHEDMiner` instances when that limit is reached:

```
    function createNewInstance() external {
        uint256 lastId = IHydra(HYDRA_ADDRESS).getUserLatestMintId(activeInstance);
        if (lastId < MAX_MINT_PER_WALLET) revert Prohibited();
        _deployInstance();
    }
```

That means that we will update the state `activeInstance` and we won't be able to claim the duplicate ids, if there are any left:

```
        address instance = activeInstance;
        if (numInstances > 1) {
            uint256 lastId = IHydra(HYDRA_ADDRESS).getUserLatestMintId(
                activeInstance);
            if (id > lastId) instance = instances[numInstances - 2];
        }
```

- Imagine we have `id = 50` from the last instance, which had `numOfDays = 88`
- For the next 88 days, `SHED` has been very active and has gained 1000 new minters
- Now we have a new instance with current `id = 50`
- Tokens corresponding to the `id = 50` from the last mint are unclaimable, because we won't enter the check:

```
            uint256 lastId = IHydra(HYDRA_ADDRESS).getUserLatestMintId(
                activeInstance);
            if (id > lastId) instance = instances[numInstances - 2];
```

Recommendation: Make `claimMiner` accept `instanceAddress`, when it is called.

Resolution: Fixed

5.3.7 SCALE::_addLiquidity() - Only SCALE/BDX pool is excluded

Severity: *Low risk*

Context: SCALE.sol#L347

Description: All pools should be excluded so that users can't `skim` from the pool when we `reflect` SCALE. Leaving this L, as they can be excluded manually, but still want to raise awareness to the fact

Resolution: Acknowledged

5.4 Informational

5.4.1 SHED: Consider setting `secondsAgo` to larger value than `1 * 60` initially

Severity: *Informational*

Context: SHED.sol#L34-L35

Resolution: Fixed

5.4.2 In `SCALE::distributeReserve` we don't check if `marketingWallet` is excluded from reflection

Severity: *Informational*

Context: SCALE.sol#L232-L235

Description:
In `SCALE::distributeReserve` we don't check if `marketingWallet` is excluded from reflection. This may be a problem if the address is excluded, because we won't update his address.

Resolution: Fixed

5.4.3 `_totalMinted` is redundant to be updated in `_fixPool`, because it is private var and is never used afterwards

Severity: *Informational*

Context: SCALE.sol#L599-L600

Resolution: Fixed

5.4.4 `SCALE::_addLiquidity` using `block.timestamp` as deadline is bad practice

Severity: *Informational*

Context: SCALE.sol#L574

Resolution: Acknowledged

5.4.5 5. `SCALE::_distributeTokens` - When we transfer TitanX to `dragonXVault` we don't call `dragonXVault.updateVault`

Severity: *Informational*

Context: SCALE.sol#L538-L539

Resolution: Fixed

5.4.6 Remove `if (totalActiveMiners >= availableMinerNum) revert NoMinersAvailable();` from `deployMiner`, because it is checked inside `getCurrentMinerParams`

Severity: *Informational*

Context: SHED.sol#L104-L105

Resolution: Fixed