Catalyst Security Review

Pashov Audit Group

Conducted by: T1MOH, __141345__

December 18th 2023 - December 23rd 2023

Contents

1. About Pashov Audit Group 2
2. Disclaimer 2
3. Introduction 2
4. About Catalyst 2
5. Risk Classification 3
   5.1. Impact 3
   5.2. Likelihood 3
   5.3. Action required for severity levels 4
6. Security Assessment Summary 4
7. Executive Summary 5
8. Findings 7
   8.1. Critical Findings 7
      [C-01] Attacker can drain all ETH from IPSeed due to re-configuring tokenId 7
      [C-02] Attacker can drain all ETH from IPSeed due to malicious IIPSeedCurve implementation 8
   8.2. Medium Findings 10
      [M-01] burn() function doesn't have slippage control 10
   8.3. Low Findings 11
      [L-01] User can accidentally burn his tokens instead of selling 11
      [L-02] Malicious user can submit json injection in uri() 11
      [L-03] Add sanity checks when set storage variables 11
      [L-04] SQRT Overflow 12
      [L-05] MINIMUM_TRADE_SIZE value leak 13

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Introduction

A time-boxed security review of the **desci-ecosystem** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Catalyst

The protocol allows users to create a "project" by launching a new ERC1155 token. Different "projects" can be purchased and sold on a price bonding curve, which is uniquely configurable per each project. Each curve trade incurs trading fees.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **- e7980268004251020b47ba450c3c684dc0f38247**

fixes review commit hash **- 0dc2ee33e1e60ba812aa9e3a6da874728c374e4b**

Scope

The following smart contracts were in scope of the audit:

- `IPSeed`
- `curves/AlgebraicSigmoidCurve`

7. Executive Summary

Over the course of the security review, T1MOH, __141345__ engaged with Molecule to review Catalyst. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	Catalyst
Repository	https://github.com/moleculeprotocol/desci-ecosystem
Date	December 18th 2023 - December 23rd 2023
Protocol Type	ERC1155 bonding curve market

Findings Count

Severity	**Amount**
Critical	2
Medium	1
Low	5
Total Findings	**8**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Attacker can drain all ETH from IPSeed due to re-configuring tokenId	Critical	Resolved
[C-02]	Attacker can drain all ETH from IPSeed due to malicious IIPSeedCurve implementation	Critical	Resolved
[M-01]	burn() function doesn't have slippage control	Medium	Resolved
[L-01]	User can accidentally burn his tokens instead of selling	Low	Resolved
[L-02]	Malicious user can submit json injection in uri()	Low	Resolved
[L-03]	Add sanity checks when set storage variables	Low	Resolved
[L-04]	SQRT Overflow	Low	Resolved
[L-05]	MINIMUM_TRADE_SIZE value leak	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Attacker can drain all ETH from IPSeed due to re-configuring `tokenId`

Severity

Impact: High. Attacker can drain all ETH from IPSeed.

Likelihood: High. Nothing prevents from exploiting.

Description

Metadata of `tokenId` can be partially configured. While `projectId` is empty string, the creator can change the token parameters anytime:

```
function spawn(
    uint256 tokenId,
    string calldata name,
    string calldata symbol,
    string calldata projectId,
    IIPSeedCurve curve,
    bytes32 curveParameters,
    address sourcer
  ) public {
    if (tokenId != computeTokenId(_msgSender(), projectId)) {
      revert InvalidTokenId();
    }

    // ERC1155's `exists` function checks for totalSupply > 0, which is not what
    // we want here
    if (bytes(tokenMeta[tokenId].projectId).length > 0) {
      revert TokenAlreadyExists();
    }

    Metadata memory newMetadata =
      Metadata
        (sourcer, sourcer, name, symbol, projectId, curve, curveParameters);
    tokenMeta[tokenId] = newMetadata;

    emit Spawned(tokenId, sourcer, newMetadata);
  }
```

This behavior introduces following attack:

1. Attacker configures only Curve address for certain `tokenId`.
2. Attacker mints arbitrary amount of that `tokenId` because Curve parameters are 0, hence price is 0.
3. Attacker configures again that `tokenId`, but now with real values.
4. Attacker sells all minted tokens, draining whole balance of IPSeed.

Here is link to PoC:
https://gist.github.com/T1MOH593/4c28ede6cdc6d183927bb7e14352ea73

Recommendations

Disallow re-configuring of `tokenId`, for example require to pass non-empty `string projectId`

[C-02] Attacker can drain all ETH from IPSeed due to malicious `IIPSeedCurve` implementation

Severity

Impact: High. Attacker can drain all ETH.

Likelihood: High. Nothing prevents from exploiting.

Description

Currently user can specify arbitrary implementation of `IIPSeedCurve`:

```
function spawn(
    uint256 tokenId,
    string calldata name,
    string calldata symbol,
    string calldata projectId,
    IIPSeedCurve curve,
    bytes32 curveParameters,
    address sourcer
  ) public {
    ...

    Metadata memory newMetadata =
@>    Metadata
  (sourcer, sourcer, name, symbol, projectId, curve, curveParameters);
    tokenMeta[tokenId] = newMetadata;

    emit Spawned(tokenId, sourcer, newMetadata);
  }
```

However Curve implementation can be malicious: for example return 0 price on buy and arbitrary price on sell. On burning and minting IPSeed quotes it from Curve implementation:

```
function getBuyPrice(uint256 tokenId, uint256 want)
    public
    view
    returns
      (uint256 gross, uint256 net, uint256 protocolFee, uint256 sourcerFee)
  {
    net = tokenMeta[tokenId].priceCurve.getBuyPrice(
      totalSupply(tokenId), want, tokenMeta[tokenId].curveParameters
    );
    (protocolFee, sourcerFee) = computeFees(net);
    gross = net + protocolFee + sourcerFee;
  }

  function getSellPrice(uint256 tokenId, uint256 sell)
    public
    view
    returns
      (uint256 gross, uint256 net, uint256 protocolFee, uint256 sourcerFee)
  {
    gross = tokenMeta[tokenId].priceCurve.getSellPrice(
      totalSupply(tokenId), sell, tokenMeta[tokenId].curveParameters
    );
    (protocolFee, sourcerFee) = computeFees(gross);
    net = gross - protocolFee - sourcerFee;
  }
```

Malicious Curve implementation can incorrectly price tokens and therefore drain ETH on selling.

Recommendations

Allow using only whitelisted implementation of `IIPSeedCurve`

8.2. Medium Findings

[M-01] `burn()` function doesn't have slippage control

Severity

Impact: Medium. User receives less collateral than expects.

Likelihood: Medium. Price must go down after submitting transaction to mempool.

Description

Currently there is no mechanism for user to specify accepted price on selling tokens.

Therefore following scenario is possible:

1. User submits transaction to sell
2. Price goes down As a result, user receives less collateral for sell than expects. And there is no mechanism to set expected amount.

Recommendations

Introduce argument like `minOutputAmount`:

```
- function burn(address account, uint256 tokenId, uint256 amount)
+ function burn
+ (address account, uint256 tokenId, uint256 amount, uint256 minOutputAmount)
    public
    virtual
    override
    nonReentrant
  {
    ...

    //when selling, gross < net
    (uint256 gross, uint256 net, uint256 protocolFee, uint256 sourcerFee) =
      getSellPrice(tokenId, amount);
+   require(net >= minOutputAmount);
    ...
  }
```

8.3. Low Findings

[L-01] User can accidentally burn his tokens instead of selling

`IPSeed.sol` inherits `ERC1155BurnableUpgradeable.sol`. `burn()` is overriden to execute sell, however `burnBatch()` is not - it still burns tokens:

```
function burnBatch(
      addressaccount,
      uint256[]memoryids,
      uint256[]memoryvalues
    ) public virtual {
        if (account != _msgSender() && !isApprovedForAll(account, _msgSender
          ())) {
            revert ERC1155MissingApprovalForAll(_msgSender(), account);
        }

        _burnBatch(account, ids, values);
    }
```

[L-02] Malicious user can submit json injection in `uri()`

Function `uri()` currently builds JSON from tokenId Metadata. However Attacker can submit malicious Metadata with JSON injection on your website where you render Metadata.

Consider implementing preventative measures on your Frontend.

[L-03] Add sanity checks when set storage variables

1. Validate that fee is in accepted range in functions `setProtocolFeeBps()` and `setSourcerFeeBps()`
2. Add validation of Metadata parameters in `spawn()`

[L-04] SQRT Overflow

Some edge case with extreme parameters. The conditions are:

- "b" parameter set to some really high value
- totalSupply is also high

In such situations, the sqrt function could overflow. The mint function will break due to getBuyPrice() DoS. Because in line 29, could be very large, causing sqrt input parameter become too large and overflow.

What might happen is, the project operates normally at first, but after the totalSupply grows large enough, the funding process will stop and DoS further mint().

```
File: packages\desci-contracts\src\curves\AlgebraicSigmoidCurve.sol
27: function collateral
  (UD60x18 x, UD60x18 a, UD60x18 b, UD60x18 c) internal pure returns (uint256) {
28: UD60x18 b2plusc = (b.mul(b)).add(c);
29: UD60x18 inner = sqrt(((x.mul(x)).add(b2plusc)).sub(ud(2e18).mul(b).mul(x)));
30: UD60x18 result = (x.add(inner)).sub(sqrt(b2plusc)).mul(a);
31:
32: return unwrap(result);
33: }
```

The following is the test, when "a" is 2, "b" is 1e28 and "c" is 1e8. The key is that "b" is set to a huge value, then in the formula, could go quite large.

```
function testExtremeCurve() public {

    AlgebraicSigmoidCurve curve = new AlgebraicSigmoidCurve();

    bytes32 curveParams = bytes32(abi.encodePacked(uint64(2), uint96
      (1e28), uint96(1e8)));

    console.log("extreme case: %s", curve.getBuyPrice
      (4e38, 1 ether, curveParams));
}
```

```
Running 1 test for test/Curves.t.sol:CurveTest
[FAIL. Reason: PRBMath_UD60x18_Sqrt_Overflow
  (152100000000000000000000000000000000000000001000000000000000000 [1.521e59])] testExtre
```

It could be that for some projects, the curve is expected to have "constant" steepness, and the upper limit as high as possible. So the "b" parameter would be set as high as possible here.

To resolve this, add checks for a, b, c for reasonable value range.

[L-05] MINIMUM_TRADE_SIZE value leak

When "b" and "c" are large, "a" is small, MINIMUM_TRADE_SIZE as the amount, the `getBuyPrice()` could return 0, so can mint for free.

```
File: packages\desci-contracts\src\curves\AlgebraicSigmoidCurve.sol
27:   function collateral
  (UD60x18 x, UD60x18 a, UD60x18 b, UD60x18 c) internal pure returns (uint256) {
28:     UD60x18 b2plusc = (b.mul(b)).add(c);
29:     UD60x18 inner = sqrt(((x.mul(x)).add(b2plusc)).sub(ud(2e18).mul(b).mul
  (x)));
30:     UD60x18 result = (x.add(inner)).sub(sqrt(b2plusc)).mul(a);
31:
32:     return unwrap(result);
33:   }
```

The following is the test.

```
function testMinimum() public {

    AlgebraicSigmoidCurve curve = new AlgebraicSigmoidCurve();

        // a = 200, b = 2e18, c = 2e18
    bytes32 curveParams = bytes32(abi.encodePacked(uint64(200), uint96
      (2e18), uint96(2e18)));

    console.log("first token price: %s", curve.getBuyPrice
      (0, 0.00001 ether, curveParams));
  }
```

```
[PASS] testMinimum() (gas: 349645)
Logs:
  first token price: 0
```

To resolve this, enforce the `getBuyPrice()` return positive result:

```
File: packages\desci-contracts\src\curves\AlgebraicSigmoidCurve.sol
66:   function getBuyPrice
  (uint256 supply, uint256 want, bytes32 curveParameters)
72:   {
73:     (UD60x18 a, UD60x18 b, UD60x18 c) = decodeParameters(curveParameters);
74:     uint256 startPrice = collateral(ud(supply), a, b, c);
75:     uint256 endPrice = collateral(ud(supply + want), a, b, c);
+       require(endPrice > startPrice);
76:     return endPrice - startPrice;
77:   }
```