Catalyst Security Review

Pashov Audit Group

Conducted by: Said, ast3ros, ether_sky

April 8th 2024 - April 12th 2024

Contents

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 **moleculeprotocol/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 - ed869fb84c764232fceda77732def7db8d315f4d

fixes review commit hash - 57e66c94c3537d1673e359549e7fc477be48782e

Scope

The following smart contracts were in scope of the audit:

- `LinearCurve`
- `SafeIPSeedTrust`
- `IPSeed`
- `IPSeedTrust`

7. Executive Summary

Over the course of the security review, Said, ast3ros, ether_sky engaged with Catalyst to review Catalyst. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Catalyst
Repository	https://github.com/moleculeprotocol/desci-ecosystem
Date	April 8th 2024 - April 12th 2024
Protocol Type	ERC1155 bonding curve market

Findings Count

Severity	Amount
Critical	1
High	1
Medium	3
Low	2
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Withdrawing collateral and fees and bypassing trust safety mechanism	Critical	Resolved
[H-01]	Broken mint if market pre-mint less than p	High	Resolved
[M-01]	Revert if supply is less than premint parameter	Medium	Resolved
[M-02]	Donating without transferring contribution	Medium	Resolved
[M-03]	10% maximum pre-mint check can be bypassed	Medium	Resolved
[L-01]	Collateral not returned if there is a disagreement	Low	Acknowledged
[L-02]	Dust amount left after all refund is made	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Withdrawing collateral and fees and bypassing trust safety mechanism

Severity

Impact: High

Likelihood: High

Description

From the project specification:

```
sourcer: the user who creates the project, a medium-high trust party, as defined by on
    protocolTrustee: a protocol-controlled wallet, and high trust party
      (molecule), set on the configured IPSeedTrust contract.
```

The IPSeedTrust contract is designed to ensure that both the sourcer and the protocol trustee are required to sign off on token withdrawal transactions (e.g., `claimCollateral` and `projectSucceeded`) by verifying that the `protocolTrustee` is an owner and the threshold is set to 2.

```
function checkIfBeneficiaryIsATrustedSafe(address beneficiary) public view {
    if (protocolTrustee == address(0)) {
      return; //when no trustee is configured, we're not checking for Safe
      // accounts
    }
    IOwnerManager ownerManager = IOwnerManager(beneficiary);

    if (

        || ownerManager.getOwners().length != 2 || !ownerManager.isOwner
          (protocolTrustee)
    ) {
      revert BeneficiaryIsNotTrustful();
    }
  }
```

However, the current implementation only verifies the beneficiary's codehash against the Safe Proxy contract v1.3.0's codehash. It does not confirm whether the Safe singleton itself is the legitimate v1.3.0 contract. This oversight allows a malicious actor to deploy a Safe contract with a fraudulent singleton, circumventing the intended security checks. Consequently, the malicious actor can assume control over the beneficiary safe and execute token withdrawal transactions without protocol trustee approval.

For instance, a malicious singleton might simulate a `protocolTrustee` ownership status, misleading the `isOwner` check to always return true for the `protocolTrustee` under the guise of meeting the two-signatory requirement.

```
contract MaliciousGnosisSafe is GnosisSafe {
    address public fakeOwner;
    address public realOwner;

    function setReturnOwner(address _fakeOwner, address _realOwner) public {
        fakeOwner = _fakeOwner;
        realOwner = _realOwner;
    }

    function getOwners() public view override returns (address[] memory) {
        address[] memory array = new address[](2);
        array[0] = fakeOwner;
        array[1] = realOwner;
        return array;
    }

    function getThreshold() public view override returns (uint256) {
        return 2;
    }

    function isOwner(address owner) public view override returns (bool) {
        if (owner == fakeOwner) {
            return true;
        } else {
            return super.isOwner(owner);
        }
    }
}
```

Setup and POC:

- Put the POC file MaliciousGenosisSafe.t.sol in test/MaliciousGenosisSafe.t.sol https://gist.github.com/thangtranth/701abaf86864510967fbacc7dbe79eba

- Put the GnosisSafe.sol (MaliciousGnosisSafe singleton) in test/helpers/GnosisSafe.sol https://gist.github.com/thangtranth/4fba8229778254d59c18a2e740239ad8

- Put the ISafeProxyFactory.sol in test/helpers/ISafeProxyFactory.sol https://gist.github.com/thangtranth/20edd2cbbcb7afb73dd0a6c13a771448

Run `forge test -vvvvv --match-path test/MaliciousGenosisSafe.t.sol --match-test testPOC`

Recommendations

To mitigate this vulnerability, ensure the validation of the singleton's authenticity:

- Reading the slot 0 of the proxy and verify offchain, warning users if the singleton is not in the allowed list.
- In contract:

```
+  interface ISafe {
+     function getStorageAt(uint256, uint256) external view returns
+ (bytes memory);
+  }

+  address SAFE_SINGLETON_CONTRACT = 0xd9Db270c1B5E3Bd161E8c8503c55cEABeE709552;

  function checkIfBeneficiaryIsATrustedSafe(address beneficiary) public view {
    if (protocolTrustee == address(0)) {
      return; //when no trustee is configured, we're not checking for Safe
      // accounts
    }
    IOwnerManager ownerManager = IOwnerManager(beneficiary);

+   bytes memory data = ISafe(beneficiary).getStorageAt(0, 1);
+   address singletonAddress;
+
+   assembly {
+        singletonAddress := mload(add(data, 32))
+   }

-   if (
+   if ( singletonAddress != SAFE_SINGLETON_CONTRACT ||

        || ownerManager.getOwners().length != 2 || !ownerManager.isOwner
          (protocolTrustee)
    ) {
      revert BeneficiaryIsNotTrustful();
    }
  }
```

8.2. High Findings

[H-01] Broken mint if market pre-mint less than `p`

Severity

Impact: Medium

Likelihood: High

Description

When market is created, it is allowed that `params.premint` and the `p` configured inside `params.curveParameters` is different, as long as `params.premint` is lower and result in 0 when provided to `getBuyPrice`.

```
function checkParameters(MarketParameters memory params) public view virtual {
    if (!trustedCurves[address(params.priceCurve)]) {
      revert UntrustedCurve();
    }

    // reverts if parameters are out of sanity range
    params.priceCurve.checkParameters
      (params.curveParameters, params.fundingGoal);

    //the "premint" parameter can be different from the curve's premint
    // parameter
    if (params.priceCurve.getBuyPrice
      (0, params.premint, params.curveParameters) != 0) {
      revert ParameterMismatch();
    }

    // check that the deadline isn't too far in the future
    //(eg millisecond issues)
    if (params.deadline > block.timestamp + 10 * 365 days) {
      revert ParameterMismatch();
    }
  }
```

However, when mint is called, it will check against `params.premint`, which is not correct and will cause issue if the actual `p` parameter is greater than `params.premint`.

```
function mint(uint256 tokenId, uint256 amount)
    external
    payable
    nonReentrant
    returns (uint256 gross, uint256 net, uint256 tokensToMint)
  {
    MarketData storage market = markets[tokenId];
    if (market.state != MarketState.OPEN) {
      revert BadState(MarketState.OPEN, market.state);
    }

    MarketParameters memory _marketParams = marketParams[tokenId];

    if (trySettle(tokenId) > MarketState.OPEN) {
      //this allows settling the market implicitly without reverting
      Address.sendValue(payable(_msgSender()), msg.value);
      return (0, 0, 0);
    }

    uint256 tradingFee;
    //when buying, gross > net
    (gross, net, tradingFee) = getBuyPrice(tokenId, amount);

    // revert trades that require too low volume but still allow premints that
    // require 0 value
    if (
>>>   totalSupply
//(tokenId) + amount > _marketParams.premint // @audit - premint here can be lower tha
        && (amount < MINIMUM_TRADE_SIZE || net == 0)
    ) {
      revert TradeSizeOutOfRange();
    }
   // ...
}
```

Users who only want to mint the rest of the free pre-mint from the actual `p` inside curve parameters will always revert because the net will be 0.

Recommendations

Consider checking against the `p` from curve parameters instead of `_marketParams.premint`.

```
+  (, uint256 p) = _marketParams.priceCurve.decodeParameters
+ (_marketParams.curveParameters);
    if (
-      totalSupply(tokenId) + amount > _marketParams.premint
+      totalSupply(tokenId) + amount > p
        && (amount < MINIMUM_TRADE_SIZE || net == 0)
    ) {
      revert TradeSizeOutOfRange();
    }
```

8.3. Medium Findings

[M-01] Revert if supply is less than premint parameter

Severity

Impact: Medium

Likelihood: Medium

Description

The `getTokensNeededToAddEthValue` function reverts if the `supply` is less than the `curve`'s `premint` parameter.

```
function getTokensNeededToAddEthValue(
  uint256supply,
  uint256ethAmount,
  bytes32curveParameters
) external pure returns (uint256 tokenAmount
    (uint256 m, uint256 p) = decodeParameters(curveParameters);
    UD60x18 pmins = ud(supply - p);
}
```

I will describe the scenario where the `supply` is smaller than the `curve`'s `premint` parameter.

The `premint` parameter of `Market Parameters` can be lower than the `curve`'s `premint` parameter for various purposes. We can know this from the below comments.

```
function checkParameters(MarketParameters memory params) public view virtual {
    //the "premint" parameter can be different from the curve's premint
    // parameter
    if (params.priceCurve.getBuyPrice
      (0, params.premint, params.curveParameters) != 0) {
      revert ParameterMismatch();
    }
}
```

When spawning a new project, the initial tokens are minted to the `sourcer`, and the `total supply` becomes the `premint` of `Market Parameters`.

```
function spawn(MarketParameters memory params) external nonReentrant {
    _mint(params.sourcer, params.tokenId, params.premint, "");
}
```

If the first `minter` intends to mint enough tokens to cover the `funding goal`, the necessary token amounts for this are calculated. However, due to the current `total supply` being less than the `curve`'s `premint` parameter, the transaction will be reverted.

```
function mint(uint256 tokenId, uint256 amount)
    external
    payable
    nonReentrant
    returns (uint256 gross, uint256 net, uint256 tokensToMint)
{
    if (market.collateral + net > _marketParams.fundingGoal) {
       tokensToMint = _marketParams.priceCurve.getTokensNeededToAddEthValue(
          totalSupply(tokenId),
          _marketParams.fundingGoal - market.collateral,
          _marketParams.curveParameters
        );
    }
}
```

Recommendations

Update the `getTokensNeededToAddEthValue` function like below:

```
function getTokensNeededToAddEthValue
  (uint256 supply, uint256 ethAmount, bytes32 curveParameters)
    external
    pure
    returns (uint256 tokenAmount)
{
    (uint256 m, uint256 p) = decodeParameters(curveParameters);
-     UD60x18 pmins = ud(supply - p);
+     UD60x18 pmins = ud(supply > p ? supply - p : 0);
    UD60x18 _m = ud(m);
    UD60x18 rootTerm = _m.mul(_m.mul(pmins.mul(pmins)).add(ud(2 * ethAmount)));
    UD60x18 result = (rootTerm.sqrt()).div(_m).ceil();
-     tokenAmount = result.intoUint256() + p - supply;
+     tokenAmount = result.intoUint256() - pmins;
}
```

[M-02] Donating without transferring contribution

Severity

Impact: Low

Likelihood: High

Description

When the project reaches at least the `FUNDED` state, seed tokens become transferable by anyone, and the amount of contribution will be transferred proportionally to the amount of tokens transferred.

```
function _update
    (address from, address to, uint256[] memory ids, uint256[] memory values)
    internal
    virtual
    override
  {
    if (from != address(0) && to != address(0)) {
      uint256 transferLength = ids.length;
      for (uint256 i; i < transferLength; ++i) {
        MarketData memory market = markets[ids[i]];
        if (
          (market.state < MarketState.FUNDED) && from != marketParams[ids[i]].sourcer)
          revert TransferRestricted();
        }
>>>     uint256 transferContribution =
          values[i] * contributions[ids[i]][from] / balanceOf(from, ids[i]);
        contributions[ids[i]][from] -= transferContribution;
        contributions[ids[i]][to] += transferContribution;
        emit ContributionTransferred(ids[i], from, to, transferContribution);
      }
    }
    super._update(from, to, ids, values);
  }
```

However, due to rounding errors, a griefer can transfer seed tokens to others without transferring their contribution if `values[i] * contributions[ids[i]][from]` is lower than `balanceOf(from, ids[i])`.

This can cause an issue. Consider a scenario when the market reaches the `FUNDED` state, and user A, wants to transfer part of their tokens to another user B. Before the transfer occurs, a griefer transfers their seed tokens to user A. Without knowing that their balance has increased, user A transfers a portion of the tokens to user B. However, due to the increased balance of user A, the amount of contribution user B receives will be less than it should be.

Coded PoC :

```
function testTransferGriefContributions() public {
     (uint256 tokenId,) = defaultMarket(ophelia);

    userMint(alice, tokenId, 2000 ether);
    userMint(bob, tokenId, 3000 ether);
    userMint(charlie, tokenId, 5001 ether);

    uint256 alicesContribution = ipSeed.contributions(tokenId, alice);
    uint256 aliceBalance = ipSeed.balanceOf(alice, tokenId);
    uint256 halfAliceBalanceInitial = aliceBalance / 2;

         uint256 expectedTransferedContribution = halfAliceBalanceInitial * alicesCont
    console.log("alice contribution before :");
    console.log(alicesContribution);
    console.log("alice balance before :");
    console.log(aliceBalance);

    //charlie donate to alice but not tranferring contribution
    uint256[] memory ids = new uint256[](200);
    uint256[] memory amounts = new uint256[](200);
    for (uint i; i < 200; i++) {
       ids[i] = tokenId;
       amounts[i] = 600;
    }
    vm.startPrank(charlie);
    ipSeed.safeBatchTransferFrom(charlie, alice, ids, amounts, "");
    vm.stopPrank();

    assertEq(alicesContribution, ipSeed.contributions(tokenId, alice));

    console.log("alice contribution after :");
    console.log(ipSeed.contributions(tokenId, alice));
    console.log("alice balance after :");
    console.log(ipSeed.balanceOf(alice, tokenId));

    uint256 bobContributionBefore = ipSeed.contributions(tokenId, bob);

    // alice transfer half of token amount before donation from charlie
    vm.startPrank(alice);
    ipSeed.safeTransferFrom
      (alice, bob, tokenId, halfAliceBalanceInitial, "");
    vm.stopPrank();

    uint256 receivedContribution = ipSeed.contributions
      (tokenId, bob) - bobContributionBefore;

    console.log("expected contribution : ");
    console.log(expectedTransferedContribution);
    console.log("transferred contribution after donation : ");
    console.log(receivedContribution);

  }
```

Test output :

```
expected contribution :
  204060810121000000
  transferred contribution after donation :
  204060810120999987
```

It can be observed that the received contribution for the same amount of token will be less than expected.

Recommendations

Inside `_update`, consider making sure that transferred contributions should not equal 0 when `contributions` value is non-0.

[M-03] 10% maximum pre-mint check can be bypassed

Severity

Impact: Medium

Likelihood: Medium

Description

When users create market, it will need to be checked by `LinearCurve.checkParameters` if the provided configuration is correct. `m` must within the predefined range and `p` (pre-mint) amount must not exceed 10% of total supply.

```
function checkParameters
    (bytes32 curveParameters, uint256 fundingGoal) external pure override {
    (uint256 m, uint256 p) = decodeParameters(curveParameters);
    if ((m < 1e4) || (m >= 1e19)) {
      revert CurveParametersOutOfRange();
    }

    uint256 fundingGoalInTokens = supplyAtCollateral
      (curveParameters, fundingGoal);

    //shouldnt allow more than 10% premints
    if (((100 * p) / fundingGoalInTokens) > 10) {
      revert CurveParametersOutOfRange();
    }
  }
```

`supplyAtCollateral` calculates the amount of seed tokens given parameters and collateral amount using the provided formula. The check is then performed to ensure that the result of `(100 * p)` / `fundingGoalInTokens` does not exceed 10. However, under certain configurations, it can be bypassed due to precision loss.

PoC :

Add this unit test to test contract :

```
function testPreMintExceed10Percent() public {
      uint256 slope = curve.computeSlope(10 ether, 9000 ether, 990 ether);
      uint256 fundingGoalInTokens =
      curve.supplyAtCollateral(bytes32(abi.encodePacked(uint128(slope), uint128
        (990 ether))), 10 ether);

      (uint256 tokenId,) = defaultMarket(ophelia);

      userMint(alice, tokenId, 9000 ether);


      MarketData memory _marketData = ipSeed.getMarketData(tokenId);

      console.log("market collateral collected : ");
      console.log(_marketData.collateral);

      console.log("supplyAtCollateral result : ");
      console.log(fundingGoalInTokens);
      assertTrue(_marketData.state == MarketState.FUNDED);
    }
```

Modify `DefaultParams` to the following :

```
function defaultParams
    (uint256 tokenId, address sourcer, address beneficiary, LinearCurve curve)
    public
    view
    returns (MarketParameters memory params)
  {
    uint256 slope = curve.computeSlope(10 ether, 9000 ether, 990 ether);

    params = MarketParameters({
      tokenId: tokenId,
      projectId: projectId,
      sourcer: sourcer,
      beneficiary: beneficiary,
      priceCurve: curve,
      curveParameters: bytes32(abi.encodePacked(uint128(slope), uint128
        (990 ether))),
      fundingGoal: 10 ether,
      premint: 990 ether,
      deadline: uint64(block.timestamp + defaultDeadline)
    });
  }
```

From the test, the pre-mint minted amount will be 11% (extra 90 * 1e18 amount of seeds token) instead of the capped 10%.

Recommendations

Increase the precision when checking the percentage to minimize the loss of precision.

```
-    if (((100 * p) / fundingGoalInTokens) > 10) {
+    if (((10000 * p) / fundingGoalInTokens) > 1000) {
       revert CurveParametersOutOfRange();
     }
```

One more fix option:

```
function checkParameters
    (bytes32 curveParameters, uint256 fundingGoal) external view override {
    ...
    uint256 fundingGoalInTokens = supplyAtCollateral
      (curveParameters, fundingGoal);

    //shouldnt allow more than 10% premints
-   if (((100 * p) / fundingGoalInTokens) > 10) {
+   if ((100 * p) > 10 * fundingGoalInTokens) {
      revert CurveParametersOutOfRange();
    }
  }
```

8.4. Low Findings

[L-01] Collateral not returned if there is a disagreement

When the negotition is failed, the beneficiary can call `negotiationFailed` function to returns the collateral back. The function has the `onlyBeneficiary` modifier, which means only the beneficiary can call this function. However, if the beneficiary is a Gnosis Safe multisig with threshold 2 and the sourcer refuses to agree to the return of the collateral, effectively taking it "hostage," the collateral cannot be rightfully returned. This deadlock occurs because the execution of such a transaction mandates signatures from both the sourcer and the protocol trustee.

```
function negotiationFailed(uint256 tokenId)
    external
    payable
    nonReentrant
    onlyBeneficiary
    //(tokenId) // @audit Need both sourcer and prtocol trustee to sign
  {
    ...
  }
```

[L-02] Dust amount left after all refund is made

When the market is expired, the `refund` function calculates the refund amount based on the user's contribution to the funding goal. The refund amount is calculated by dividing the user's contribution by the accrued capital and multiplying it by the claimable capital. The result is rounded down, which may result in dust amount being left in the contract after all refund is made.

It can be avoided by tracking total claimed contribution and total refunded payout. If the total claimed contribution is equal to the `accruedCapital`, it means it is the last refund and the remaining claimable capital (claimableCapital - total refunded payout) can be refunded to the last user.

```
function refund(uint256 tokenId, address contributor, uint16 feeBps)
    private
    returns (uint256 payout, uint256 tradingFee)
  {
    uint256 contribution = contributions[tokenId][contributor];
    MarketData storage market = markets[tokenId];
    //in an open market users simply get their contribution back
    //in expired markets they receive a share of the claimable collateral
    // proportional to their original contribution to the funding goal
    if (market.state > MarketState.OPEN) {
      payout =
      //(contribution * market.claimableCapital) / market.accruedCapital; // @audit re
    } else {
    ...
  }
```

However, the impact is very small since the dust amount is negligible.