Juicebox contest
Findings & Analysis Report

2023-01-09

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Juicebox DAO smart contract system written in Solidity. The audit contest took place between October 18—23, 2022.

Wardens

70 Wardens contributed reports to the Juicebox contest:

1. 0x1f8b
2. 0x4non
3. 0x52
4. 0x5rings
5. 0xNazgul
6. 0xSmartContract
7. Awesome
8. Aymen0909
9. BClabs (nalus and Reptilia)
10. Bnke0x0
11. CodingNameKiki
12. Diana
13. DimSon
14. JC
15. Jeiwan
16. JrNet
17. Lambda
18. LeoS
19. RaoulSchaffranek
20. RaymondFam

21. RedOneN
22. ReyAdmirado
23. Rolezn
24. SaharAP
25. Saintcode_
26. Shinchan (Sm4rty, prasantgupta52 and Rohan16)
27. Trust
28. V_B (Barichek and vlad_bochok)
29. __141345__
30. a12jmx
31. berndartmueller
32. bharg4v
33. brgltd
34. carlitox477
35. cccz
36. ch0bu
37. chaduke
38. cloudjunky
39. cryptostellar5
40. cryptphi
41. csanuragjain
42. d3e4
43. delfin454000
44. emrekocak
45. erictee
46. fatherOfBlocks
47. gogo
48. hansfriese
49. ignacio

50. joestakey
51. karanctf
52. ladboy233
53. lukris02
54. martin
55. mcwildy
56. minhquanym
57. minhtrng
58. peanuts
59. ret2basic
60. sakman
61. seyni
62. slowmoses
63. svskaushik
64. tnevler
65. trustindistrust
66. yixxas
67. zishansami

This contest was judged by Picodes.

Final report assembled by CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities. Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 49 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 34 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Juicebox contest repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 1467 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (5)

[H-01] Making a payment to the protocol with `_dontMint` parameter will result in lost fund for user.

Submitted by yixxas, *also found by* minhquanym *and* cccz

User will have their funds lost if they tries to pay the protocol with `_dontMint = False`. A payment made with this parameter set should increase the `creditsOf[]` balance of user.

In `_processPayment()`, `creditsOf[_data.beneficiary]` is updated at the end if there are leftover funds. However, If `metadata` is provided and `_dontMint == true`, it immediately returns. [JBTiered721Delegate.sol#L524-L590]

```
function _processPayment(JBDidPayData calldata _data) internal
  // Keep a reference to the amount of credits the beneficiary
  uint256 _credits = creditsOf[_data.beneficiary];
  ...
  if (
    _data.metadata.length > 36 &&
    bytes4(_data.metadata[32:36]) == type(IJB721Delegate).inte
  ) {
    ...
    // Don't mint if not desired.
    if (_dontMint) return;
    ...
  }
  ...
  // If there are funds leftover, mint the best available with
  if (_leftoverAmount != 0) {
    _leftoverAmount = _mintBestAvailableTier(
      _leftoverAmount,
      _data.beneficiary,
      _expectMintFromExtraFunds
    );

    if (_leftoverAmount != 0) {
      // Make sure there are no leftover funds after minting i
      if (_dontOverspend) revert OVERSPENDING();

      // Increment the leftover amount.
      creditsOf[_data.beneficiary] = _leftoverAmount;
    } else if (_credits != 0) creditsOf[_data.beneficiary] = 0
  } else if (_credits != 0) creditsOf[_data.beneficiary] = 0;
}
```

Proof of Concept

I've wrote a coded POC to illustrate this. It uses the same Foundry environment used by the project. Simply copy this function to `E2E.t.sol` to verify.

```
function testPaymentNotAddedToCreditsOf() public{
```

```
address _user = address(bytes20(keccak256('user')));
(
  JBDeployTiered721DelegateData memory NFTRewardDeployerData
  JBLaunchProjectData memory launchProjectData
) = createData();

uint256 projectId = deployer.launchProjectFor(
  _projectOwner,
  NFTRewardDeployerData,
  launchProjectData
);

// Get the dataSource
IJBTiered721Delegate _delegate = IJBTiered721Delegate(
  _jbFundingCycleStore.currentOf(projectId).dataSource()
);

address NFTRewardDataSource = _jbFundingCycleStore.currentOf

uint256 _creditBefore = IJBTiered721Delegate(NFTRewardDataSo

// Project is initiated with 10 different tiers with contrib

// Make payment to mint 1 NFT
uint256 _payAmount = 10;
_jbETHPaymentTerminal.pay{value: _payAmount}(
  projectId,
  100,
  address(0),
  _user,
  0,
  false,
  'Take my money!',
  new bytes(0)
);

// Minted 1 NFT
assertEq(IERC721(NFTRewardDataSource).balanceOf(_user), 1);

// Now, we make the payment but supply _dontMint metadata
bool _dontMint = true;
uint16[] memory empty;
_jbETHPaymentTerminal.pay{value: _payAmount}(
  projectId,
  100,
  address(0),
```

```
        _user,
        0,
        false,
        'Take my money!',
        //new bytes(0)
        abi.encode(
          bytes32(0),
          type(IJB721Delegate).interfaceId,
          _dontMint,
          false,
          false,
          empty
          )
      );

      // NFT not minted
      assertEq(IERC721(NFTRewardDataSource).balanceOf(_user), 1);

      // Check that credits of user is still the same as before ev
      assertEq(IJBTiered721Delegate(NFTRewardDataSource).creditsOf

    }
```

Tools Used

Foundry

Recommended Mitigation Steps

Update the `creditsOf[]` in the `if(_dontMint)` check.

```
- if(_dontMint) return;
+ if(_dontMint){ creditsOf[_data.beneficiary] += _value; }
```

[mejango (Juicebox DAO) commented on duplicate issue #157](#):

> mixed feels. `_dontMint` basically says “Save me gas at all costs.”. I see the argument for value leaking being bad though. will mull over.

[drgorillamd (Juicebox DAO) commented on duplicate issue #157](#):

> paying small amounts (under the floor or with `dontMint`) only to save them to later mint is a bit of a nonsense -> it's way cheaper to just not pay, save in an eoa then mint within the same tx.

> I have the feeling the severity is based on seeing `_credit` as a saving account, while it's rather something to collect leftovers.

> Anyway, we changed it, but not sure of high sev on this one, happy to see others' point of view.

Picodes (judge) commented:

> @drgorillamd @mejango I have to say that I don't see why someone would use the `dontMint` flag in the first place. Wasn't the original intent to use this flag specifically to modify `_credit` without minting? In the meantime I'll keep the High label for this one, the `dontMint` functionality being flawed and leading to a loss of funds.

drgorillamd (Juicebox DAO) commented:

> @Picodes `nftReward` is just an extension plugged into a Jb project -> `dontMint` is to avoid forcing users of the project who don't want a nft reward when contributing, i.e. "classic" use of a Jb project. The use case we had in mind was smaller payers, wanting to get the erc20 (or even just donating), without the gas burden of a nft reward (which might, on L1, sometimes be more than the contribution itself). Does that make sense?

Picodes (judge) commented:

> Definitely, thanks for the clarification @drgorillamd.

Picodes (judge) commented:

> The final decision for this issue was to keep the high severity because of the leak of value and the possibility that some users use the function thinking it will change `_credit`, despite the fact that it was not the original intent of the code.

mejango (Juicebox DAO) commented:

> We ended up adding credits even when `_dontMint` is true!!
> It was a last minute design decision, initially we marked the issue as “Disagree with severity” and we were planning on keeping the code unchanged since it didnt pose a risk and was working as designed.
> We ended up changing the design, but the wardens’ feedback was ultimately helpful!

[H-02] Minting and redeeming will break for fully minted tiers with `reserveRate != 0` and `reserveRate / MaxReserveRate` tokens burned

Submitted by 0x52

Minting and redeeming become impossible.

Proof of Concept

```
uint256 _numberOfNonReservesMinted = _storedTier.initialQuantity
  _storedTier.remainingQuantity -
  _reserveTokensMinted;

uint256 _numerator = uint256(_numberOfNonReservesMinted * _store

uint256 _numberReservedTokensMintable = _numerator / JBConstants

if (_numerator - JBConstants.MAX_RESERVED_RATE * _numberReserved
  ++_numberReservedTokensMintable;

return _numberReservedTokensMintable - _reserveTokensMinted;
```

The lines above are taken from JBTiered721DelegateStore#_numberOfReservedTokensOutstandingFor and used to calculate and return the available number of reserve tokens that can be minted. Since the return statement doesn’t check that _numberReservedTokensMintable >= _reserveTokensMinted, it will revert under those circumstances. The issue is that there are legitimate circumstances in which this becomes false. If a tier is fully minted then all reserve tokens are mintable. When the tier begins to redeem, _numberReservedTokensMintable will fall under _reserveTokensMinted, permanently

breaking minting and redeeming. Minting is broken because all mint functions directly call _numberOfReservedTokensOutstandingFor. Redeeming is broken because the redeem callback (JB721Delegate#redeemParams) calls _totalRedemtionWeight which calls _numberOfReservedTokensOutstandingFor.

Example:

A tier has a reserveRate of 100 (1/100 tokens reserved) and an initialQuantity of 10000. We assume that the tier has been fully minted, that is, _reserveTokensMinted is 100 and remainingQuantity = 0. Now we begin burning the tokens. Let's run through the lines above after 100 tokens have been burned (remainingQuantity = 100):

_numberOfNonReservedMinted = 10000 - 100 - 100 = 9800

_numerator = 9800 * 100 = 980000

_numberReservedTokensMintable = 980000 / 10000 = 98

Since _numberReservedTokensMintable < _reserveTokensMinted the line will underflow and revert.

JBTiered721DelegateStore#_numberOfReservedTokensOutstandingFor will now revert every time it is called. This affects all minting functions as well as totalRedemptionWeight. Since those functions now revert when called, it is impossible to mint or redeem anymore NFTs.

Recommended Mitigation Steps

Add a check before returning:

```
+   if (_reserveTokensMinted > _numberReservedTokensMintable) {
+       return 0;
+   }

    return _numberReservedTokensMintable - _reserveTokensMinted;
```

mejango (Juicebox DAO) confirmed

Trust (warden) commented:

> The root cause seems to be that there is no tracking of reserve tokens burnt.

mejango (Juicebox DAO) commented:

> @Trust fair. this would require extra storage to track which tokenIDs were minted as reserves. could be a nice-to-have, and also used to prevent this issue.

Picodes (judge) commented:

> Without tracking the number of burnt tokens, the mitigation suggested by the warden avoids the underflow so solves the main issue, which is that minting and redeeming break

> Accounting for `numberOfBurnedReservesFor` may help fixing the math but the underflow would still be possible if only non reserve tokens are burned

[H-03] Outstanding reserved tokens are incorrectly counted in total redemption weight

Submitted by Jeiwan, *also found by* Trust, ladboy233, *and* cccz

The amounts redeemed in overflow redemption can be calculated incorrectly due to incorrect accounting of the outstanding number of reserved tokens.

Proof of Concept

Project contributors are allowed to redeem their NFT tokens for a portion of the overflow (excessive funded amounts). The amount a contributor receives is calculated as overflow * (user's redemption rate / total redemption weight), where user's redemption weight is the total contribution floor of all their NFTs and total redemption weight is the total contribution floor of all minted NFTs. Since the total redemption weight is the sum of individual contributor redemption weights, the amount they can redeem is proportional to their contribution.

However, the total redemption weight calculation incorrectly accounts outstanding reserved tokens (JBTiered721DelegateStore.sol#L563-L566):

```
// Add the tier's contribution floor multiplied by the quantity r
weight +=
  (_storedTier.contributionFloor *
    (_storedTier.initialQuantity - _storedTier.remainingQuantity
  _numberOfReservedTokensOutstandingFor(_nft, _i, _storedTier);
```

Specifically, the *number* of reserved tokens is added to the *weight* of minted tokens. This disrupts the redemption amount calculation formula since the total redemption weight is in fact not the sum of individual contributor redemption weights.

Recommended Mitigation Steps

Two options can be seen:

1. if the outstanding number of reserved tokens is considered minted (which seems to be so, judging by this logic) then it needs to be added to the quantity, i.e.:

```
--- a/contracts/JBTiered721DelegateStore.sol
+++ b/contracts/JBTiered721DelegateStore.sol
@@ -562,8 +562,7 @@ contract JBTiered721DelegateStore is IJBTiered721De
      // Add the tier's contribution floor multiplied by the quantity m
      weight +=
        (_storedTier.contributionFloor *
-         (_storedTier.initialQuantity - _storedTier.remainingQuantity
-       _numberOfReservedTokensOutstandingFor(_nft, _i, _storedTier);
+         (_storedTier.initialQuantity - _storedTier.remainingQuantity
+          _numberOfReservedTokensOutstandingFor(_nft, _i, _storedTier

      unchecked {
        ++_i;
```

2. if it's not considered minted, then it shouldn't be counted at all.

drgorillamd (Juicebox DAO) confirmed

Picodes (judge) upgraded severity:

> As the redeemed amounts are at stake, upgrading to High

[H-04] Reserved token rounding can be abused to honeypot and steal user's funds

Submitted by Trust

When the project wishes to mint reserved tokens, they call mintReservesFor which allows minting up to the amount calculated by DelegateStore's _numberOfReservedTokensOutstandingFor. The function has this line:

```
// No token minted yet? Round up to 1.
if (_storedTier.initialQuantity == _storedTier.remainingQuantity
```

In order to ease calculations, if reserve rate is not 0 and no token has been minted yet, the function allows a single reserve token to be printed. It turns out that this introduces a very significant risk for users. Projects can launch with several tierIDs of similar contribution size, and reserve rate as low as 1%. Once a victim contributes to the project, it can instantly mint a single reserve token of all the rest of the tiers. They can then redeem the reserve token and receive most of the user's contribution, without putting in any money of their own.

Since this attack does not require setting "dangerous" flags like lockReservedTokenChanges or lockManualMintingChanges, it represents a very considerable threat to unsuspecting users. Note that the attack circumvents user voting or any funding cycle changes which leave time for victim to withdraw their funds.

Impact

Honeypot project can instantly take most of first user's contribution.

Proof of Concept

New project launches, with 10 tiers, of contributions 1000, 1050, 1100, ...

Reserve rate is set to 1% and redemption rate = 100%

User contributes 1100 and gets a Tier 3 NFT reward.

Project immediately mints Tier 1, Tier 2, Tier 4,... Tier 10 reserve tokens, and redeems all the reserve tokens.

Project's total weight = 12250

Reserve token weight = 11150

Malicious project cashes 1100 (overflow) * 11150 / 12250 = ~1001 tokens.

Recommended Mitigation Steps

Don't round up outstanding reserve tokens as it represents too much of a threat.

mejango (Juicebox DAO) acknowledged

Picodes (judge) commented:

> The finding is valid and clearly demonstrates how project owners could bypass the flags and safeguards implemented to trick users into thinking that they'll be safe.

> However, it falls within the "centralization risk" category, and within reports showing "a unique attack path which users were not told upfront about" (see this issue). So I believe Medium severity to be appropriate.

Trust (warden) commented:

> I would just like to state that the way I look at it, this is not a centralization risk, as the counterparty which can perform the exploit is some listed project on Juicebox, rather than Juicebox itself. It is very similar to a high severity finding in Enso Finance, where a strategy creator can rug funds sent to their strategy.

Picodes (judge) commented:

> Kept it high risk out of coherence with https://github.com/code-423n4/2022-05-enso-findings/issues/204, and because this attack would bypass all the safeguards implemented by Juicebox

[H-05] Redemption weight of tiered NFTs miscalculates, making users redeem incorrect amounts - Bug #1

Submitted by Trust, also found by Aymen0909 and 0x52

Redemption weight is a concept used in Juicebox to determine investor's eligible percentage of the non-locked funds. In redeemParams, JB721Delegate calculates user's share using:

```
uint256 _redemptionWeight = _redemptionWeightOf(_decodedTokenIds
uint256 _total = _totalRedemptionWeight();
uint256 _base = PRBMath.mulDiv(_data.overflow, _redemptionWeight
```

_totalRedemptionWeight eventually is implemented in DelegateStore:

```
for (uint256 _i; _i < _maxTierId; ) {
  // Keep a reference to the stored tier.
  _storedTier = _storedTierOf[_nft][_i + 1];
  // Add the tier's contribution floor multiplied by the quantit
  weight +=
    (_storedTier.contributionFloor *
      (_storedTier.initialQuantity - _storedTier.remainingQuanti
    _numberOfReservedTokensOutstandingFor(_nft, _i, _storedTier)
  unchecked {
    ++_i;
  }
}
```

If we pay attention to _numberOfReservedTokensOutstandingFor() call, we can see it is called with tierId = i, yet storedTier of i+1. It is definitely not the intention as for example, recordMintReservesFor() uses the function correctly:

```
function recordMintReservesFor(uint256 _tierId, uint256 _count)
  external
  override
  returns (uint256[] memory tokenIds)
{
  // Get a reference to the tier.
  JBStored721Tier storage _storedTier = _storedTierOf[msg.sender
  // Get a reference to the number of reserved tokens mintable f
```

```
uint256 _numberOfReservedTokensOutstanding = _numberOfReserved'
  msg.sender,
  _tierId,
  _storedTier
);
...
```

The impact of this bug is incorrect calculation of the weight of user's contributions. The `initialQuantity` and `remainingQuantity` values are taken from the correct tier, but `_reserveTokensMinted` minted is taken from previous tier. In the case where `_reserveTokensMinted` is smaller than correct value, for example tierID=0 which is empty, the outstanding value returned is larger, meaning weight is larger and redemptions are worth less. In the opposite case, where lower tierID has higher `_reserveTokensMinted` , the redemptions will receive *more* payout than they should.

Impact

Users of projects can receive less or more funds than they are eligible for when redeeming NFT rewards.

Proof of Concept

1. Suppose we have a project with 2 tiers, reserve ratio = 50%, redemption ratio = 100%:

Tier	Contribution	Initial quantity	Remaining quantity	Reserves minted	Reserves outstanding
Tier 1	50	10	3	1	2
Tier 2	100	30	2	8	2

When calculating totalRedemptionWeight(), the correct result is

50 * (10 - 3) + 2 + 100 * (30-2) + 2 = 3154

The wrong result will be:

50 * (10 -3) + **4** + 100 * (30-2) + **13** = 3167

Therefore, when users redeem NFT rewards, they will get less value than they are eligible for. Note that totalRedemptionWeight() has an *additional* bug where the reserve amount is not multiplied by the contribution, which is discussed in another submission. If it would be calculated correctly, the correct weight would be 3450.

Recommended Mitigation Steps

Change the calculation to:

```
_numberOfReservedTokensOutstandingFor(_nft, _i+1, _storedTier);
```

Additional discussion

Likelihood of impact is very high, because the conditions will arise naturally (different tiers, different reserve minted count for each tier, user calls redeem). Severity of impact is high because users receive less or more tokens than they are eligible for.

Initially I thought this bug could allow attacker to steal entire unlocked project funds, using a mint/burn loop. However, this would not be profitable because their calculated share of the funds would always be at most what they put in, because reserve tokens are printed out of thin air.

mejango (Juicebox DAO) confirmed

Medium Risk Findings (8)

[M-01] Multiples initializations of `JBTiered721Delegate`

Submitted by **0x1f8b**

The `initialize` method of the `JBTiered721Delegate` contract has as a flag that the `_store` argument is different from `address(0)` , however, it can be initialized by anyone with this value to allow the project to continue with its usual initialization, the attacker could have interfered and modified the corresponding values to carry out an attack.

Proof of Concept

Looking at the method below, we highlight in green the parts that need to be initialized to prevent a call to `store=address(0)` from failing.

```
  function initialize(
    uint256 _projectId,
    IJBDirectory _directory,
    string memory _name,
    string memory _symbol,
    IJBFundingCycleStore _fundingCycleStore,
    string memory _baseUri,
    IJBTokenUriResolver _tokenUriResolver,
    string memory _contractUri,
    JB721PricingParams memory _pricing,
    IJBTiered721DelegateStore _store,
    JBTiered721Flags memory _flags
  ) public override {
    // Make the original un-initializable.
    require(address(this) != codeOrigin);
    // Stop re-initialization.
    require(address(store) == address(0));

    // Initialize the sub class.
    JB721Delegate._initialize(_projectId, _directory, _name, _sy

    fundingCycleStore = _fundingCycleStore;
    store = _store;
    pricingCurrency = _pricing.currency;
    pricingDecimals = _pricing.decimals;
    prices = _pricing.prices;

    // Store the base URI if provided.
+   if (bytes(_baseUri).length != 0) _store.recordSetBaseUri(_ba

    // Set the contract URI if provided.
+   if (bytes(_contractUri).length != 0) _store.recordSetContrac

    // Set the token URI resolver if provided.
+   if (_tokenUriResolver != IJBTokenUriResolver(address(0)))
      _store.recordSetTokenUriResolver(_tokenUriResolver);

    // Record adding the provided tiers.
+   if (_pricing.tiers.length > 0) _store.recordAddTiers(_pricin

    // Set the flags if needed.
```

```
    if (
+     _flags.lockReservedTokenChanges ||
+     _flags.lockVotingUnitChanges ||
+     _flags.lockManualMintingChanges ||
+     _flags.pausable
    ) _store.recordFlags(_flags);

    // Transfer ownership to the initializer.
    _transferOwnership(msg.sender);
  }
```

So if the attacker initializes the contract as follows:

- `_baseUri` = ""
- `_contractUri` = ""
- `_tokenUriResolver` = `address(0)`
- `_pricing.tiers` = []
- `_flags` = all `false`

The contract will be initialized and transfered the ownership to `msg.sender`.

After that, the owner can call `didPay` with the the fake data provided in JBTiered721Delegate.sol:221 and increase `creditsOf` of anyone JBTiered721Delegate.sol:587 without touching any `store` call.

- The attacker can transfer the ownership to the contract, and the project will be able to initialize the contract again without notice.

Recommended Mitigation Steps

Ensure that the `store` address is not empty.

Picodes (judge) commented:

> I believe the finding to be valid if:

- the attacker initialize the contract with `_store == address(0)` and the parameters as above so it does not revert in the normal process

- the attacker calls initialize to transfer the ownership to himself and modify the storage so he can then call `didPay`
- the attacker calls `didPay` to manipulate `creditsOf`
- finally the attacker calls `initialize` to set `_store` to non zero and at this point it is like if nothing happened although `creditsOf` has been manipulated

drgorillamd (Juicebox DAO) commented:

> Hmm, this would require a spoof directory too (to bypass the isTerminalOf check) -> I'd mitigate with a `check msg.value==data.value` in the abstract delegate contract, ie if someone wants to do this, actually paying the credit is needed

> Def nice finding, ggwp!

Picodes (judge) commented:

> I do agree that Med is more appropriate as it falls within centralization risks as ultimately only the deployer could exploit this.

[M-02] The tier setting parameter are unsafely downcasted from type uint256 to type uint80 / uint48 / uint40 / uint16

Submitted by ladboy233, *also found by* brgltd

https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/JBTiered721Delegate.sol#L240
https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/JBTiered721DelegateStore.sol#L628
https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/JBTiered721DelegateStore.sol#L689

The tier setting parameter are unsafely downcasted from uint256 to uint80 / uint48 / uint16

the tier is setted by owner is crucial because the parameter affect how nft is minted.

the callstack is

`JBTiered721Delegate.sol#initialize -> Store#recordAddTiers`

```
function recordAddTiers(JB721TierParams[] memory _tiersToAdd)
```

what does the struct `JB721TierParams` look like? all parameter in `JB721TierParams` is uint256 type

```
struct JB721TierParams {
  uint256 contributionFloor;
  uint256 lockedUntil;
  uint256 initialQuantity;
  uint256 votingUnits;
  uint256 reservedRate;
  address reservedTokenBeneficiary;
  bytes32 encodedIPFSUri;
  bool allowManualMint;
  bool shouldUseBeneficiaryAsDefault;
}
```

however in side the function

```
// Record adding the provided tiers.
if (_pricing.tiers.length > 0) _store.recordAddTiers(_pricing.ti
```

all uint256 parameter are downcasted.

```
// Add the tier with the iterative ID.
_storedTierOf[msg.sender][_tierId] = JBStored721Tier({
contributionFloor: uint80(_tierToAdd.contributionFloor),
lockedUntil: uint48(_tierToAdd.lockedUntil),
remainingQuantity: uint40(_tierToAdd.initialQuantity),
initialQuantity: uint40(_tierToAdd.initialQuantity),
votingUnits: uint16(_tierToAdd.votingUnits),
reservedRate: uint16(_tierToAdd.reservedRate),
allowManualMint: _tierToAdd.allowManualMint
```

```
});
```

uint256 `contributionFloor` is downcasted to uint80,

uint256 `lockedUntil` is downcasted to uint48

uint256 `initialQuantity` and `initialQuantity` are downcasted to uint40

uint256 `votingUnits` and uint256 `reservedRate` are downcasted to uint16

this means the original setting is greatly truncated.

For example, the owner wants to set the initial supply to a number larger than uint40, but the supply is truncated to type(uint40).max

The owner wants to set the contribution floor price above uint80,but the contribution floor price is truncated to type(uint80).max, the user may underpay the price and get the NFT price at a discount.

Proof of Concept

We can add POC

https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/forge-test/NFTReward_Unit.t.sol#L1689

```
function testJBTieredNFTRewardDelegate_mintFor_mintArrayOfTiers_
    uint256 nbTiers = 1;

    vm.mockCall(
      mockJBProjects,
      abi.encodeWithSelector(IERC721.ownerOf.selector, projectId
      abi.encode(owner)
    );

    JB721TierParams[] memory _tiers = new JB721TierParams[](nbTi
    uint16[] memory _tiersToMint = new uint16[](nbTiers);

    // Temp tiers, will get overwritten later (pass the construc
```

```
uint256 originalFloorPrice = 1000000000000000000000000 ethe

for (uint256 i; i < nbTiers; i++) {
  _tiers[i] = JB721TierParams({
    contributionFloor: originalFloorPrice,
    lockedUntil: uint48(0),
    initialQuantity: 20,
    votingUnits: uint16(0),
    reservedRate: uint16(0),
    reservedTokenBeneficiary: reserveBeneficiary,
    encodedIPFSUri: tokenUris[i],
    allowManualMint: true, // Allow this type of mint
    shouldUseBeneficiaryAsDefault: false
  });

  _tiersToMint[i] = uint16(i)+1;
  _tiersToMint[_tiersToMint.length - 1 - i] = uint16(i)+1;
}

ForTest_JBTiered721DelegateStore _ForTest_store = new ForTes
ForTest_JBTiered721Delegate _delegate = new ForTest_JBTiered
  projectId,
  IJBDirectory(mockJBDirectory),
  name,
  symbol,
  IJBFundingCycleStore(mockJBFundingCycleStore),
  baseUri,
  IJBTokenUriResolver(mockTokenUriResolver),
  contractUri,
  _tiers,
  IJBTiered721DelegateStore(address(_ForTest_store)),
  JBTiered721Flags({
    lockReservedTokenChanges: false,
    lockVotingUnitChanges: false,
    lockManualMintingChanges: true,
    pausable: true
  })
);

_delegate.transferOwnership(owner);

uint256 floorPrice = _delegate.test_store().tier(address(_de
console.log("original floor price");
console.log(originalFloorPrice);
console.log("truncated floor price");
console.log(floorPrice);
```

```
    }
```

note, our initial contribution floor price setting is

```
    uint256 originalFloorPrice = 100000000000000000000000000 ether;

    for (uint256 i; i < nbTiers; i++) {
      _tiers[i] = JB721TierParams({
            contributionFloor: originalFloorPrice,
```

then we run our test

```
    forge test -vv --match testJBTieredNFTRewardDelegate_mintFor_min
```

the result is

```
    [PASS] testJBTieredNFTRewardDelegate_mintFor_mintArrayOfTiers_do
    Logs:
      original floor price
      100000000000000000000000000000000000000000000000
      truncated floor price
      863278115882885135204352

    Test result: ok. 1 passed; 0 failed; finished in 10.43ms
```

clearly the floor price is unsafed downcasted and truncated.

Tools Used

Foundry, Manual Review

Recommended Mitigation Steps

We recommend the project either change the data type in the struct

```
    struct JB721TierParams {
```

```
    uint256 contributionFloor;
    uint256 lockedUntil;
    uint256 initialQuantity;
    uint256 votingUnits;
    uint256 reservedRate;
    address reservedTokenBeneficiary;
    bytes32 encodedIPFSUri;
    bool allowManualMint;
    bool shouldUseBeneficiaryAsDefault;
  }
```

or safely downcast the number to make sure the number is not shortened unexpectedly.

https://docs.openzeppelin.com/contracts/3.x/api/utils#SafeCast

drgorillamd (Juicebox DAO) commented:

> Thank you for the real poc:)

Picodes (judge) commented:

> The warden showed how due to casting the original parameters could be truncated

[M-03] Changing default reserved token beneficiary may result in wrong beneficiary for tier

Submitted by Lambda

When the `reservedTokenBeneficiary` of a tier is equal to `defaultReservedTokenBeneficiaryOf[msg.sender]`, it is not explicitly set for this tier. This generally works well because in the function `reservedTokenBeneficiaryOf(address _nft, uint256 _tierId)`, `defaultReservedTokenBeneficiaryOf[_nft]` is used as a backup when `_reservedTokenBeneficiaryOf[_nft][_tierId]` is not set. However, it will lead to the wrong beneficiary when `defaultReservedTokenBeneficiaryOf[msg.sender]` is later changed, as this new beneficiary will be used for the tier, which is not the intended one.

Proof Of Concept

`defaultReservedTokenBeneficiaryOf[address(delegate)]` is originally set to `address(Bob)` when the following happens:

1. A new tier 42 is added with `_tierToAdd.reservedTokenBeneficiary = address(Bob)`. Because this is equal to `defaultReservedTokenBeneficiaryOf[address(delegate)]`, `_reservedTokenBeneficiaryOf[msg.sender][_tierId]` is not set.
2. The owner calls `setDefaultReservedTokenBeneficiary` to change the default beneficiary (i.e., the value `defaultReservedTokenBeneficiaryOf[address(delegate)]`) to `address(Alice)`.
3. Now, every call to `reservedTokenBeneficiaryOf(address(delegate), 42)` will return `address(Alice)`, meaning she will get these reserved tokens. This is of course wrong, the tier was explicitly created with Bob as the beneficiary.

Recommended Mitigation Steps

Also set `_reservedTokenBeneficiaryOf[msg.sender][_tierId]` when it is equal to the current default beneficiary.

mejango (Juicebox DAO) confirmed

drgorillamd (Juicebox DAO) commented:

> Edge case but valid imo! Nice finding!

mejango (Juicebox DAO) commented:

> yep. valid imo too!

[M-04] Iterations over all tiers in recordMintBestAvailableTier can render system unusable

Submitted by Lambda, *also found by* brgltd

`JBTiered721DelegateStore.recordMintBestAvailableTier` potentially iterates over all tiers to find the one with the highest contribution floor that is lower than `_amount`. When there are many tiers, this loop can always run out of gas, which will cause some transactions (the ones that have a high `_leftoverAmount` within `_processPayment`) to always revert. The (implicit) limit for the number of tiers is 2^16 - 1, so it is possible that this happens in practice.

Proof Of Concept

Let's say that 1,000 tiers are registered for a project. Small payments without a leftover amount or a small amount will be succesfully processed by `_processPayment`, because `_mintBestAvailableTier` is either not called or it is called with a small amount, meaning that `recordMintBestAvailableTier` will exit the loop early (when it is called with a small amount). However, if a payment with a large leftover amount (let's say greater than the highest contribution floor) is processed, it is necessary to iterate over all tiers, which will use too much gas and cause the processing to revert.

Recommended Mitigation Steps

Use a binary search (which requires some architectural changes) for determining the best available tier. Then, the gas usage grows logarithmically (instead of linear with the current design) with the number of tiers, meaning that it would only be ~16 times higher for 65535 tiers as for 2 tiers.

drgorillamd (Juicebox DAO) commented on duplicate issue #226:

> Disagree with:
>
> > Over time maxTierIdOf for a nft address gets large due to several increments
>
> There is no several increments outside of adding new tiers by the project owner (this is similar to adding new token in an erc1155 - there is no such check in, for instance, OZ https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/ERC1155.sol), this is a project owner choice, not faulty logic.

Picodes (judge) commented on duplicate issue #226:

> User funds could be at stake as `redeemParams` would revert because of the for loop in `totalRedemptionWeight`. A limit value would be indeed a good safeguard.

[M-05] NFT not minted when contributed via a supported payment terminal

Submitted by Jeiwan, also found by cccz

A contributor won't get an NFT they're eligible for if the payment is made through a payment terminal that's supported by the project but not by the NFT delegate.

Proof of Concept

A Juicebox project can use multiple payment terminals to receive contributions (JBController.sol#L441-L442). Payment terminals are single token payment terminals (JBPayoutRedemptionPaymentTerminal.sol#L310) that support only one currency (JBSingleTokenPaymentTerminal.sol#L124-L132). Since projects can have multiple terminals, they can receive payments in multiple currencies.

However, the NFT delegate supports only one currency (JBTiered721Delegate.sol#L225):

```
function initialize(
  uint256 _projectId,
  IJBDirectory _directory,
  string memory _name,
  string memory _symbol,
  IJBFundingCycleStore _fundingCycleStore,
  string memory _baseUri,
  IJBTokenUriResolver _tokenUriResolver,
  string memory _contractUri,
  JB721PricingParams memory _pricing,
  IJBTiered721DelegateStore _store,
  JBTiered721Flags memory _flags
) public override {
  // Make the original un-initializable.
  require(address(this) != codeOrigin);
  // Stop re-initialization.
  require(address(store) == address(0));
```

```
    // Initialize the sub class.
    JB721Delegate._initialize(_projectId, _directory, _name, _symb

    fundingCycleStore = _fundingCycleStore;
    store = _store;
    pricingCurrency = _pricing.currency; // @audit only one curren
    pricingDecimals = _pricing.decimals;
    prices = _pricing.prices;

    ...
  }
```

When a payment is made in a currency that's supported by the project (via one of its terminals) but not by the NFT delegate, there's an attempt to convert the currency to a supported one (JBTiered721Delegate.sol#L527-L534):

```
if (_data.amount.currency == pricingCurrency) _value = _data.amo
else if (prices != IJBPrices(address(0)))
  _value = PRBMath.mulDiv(
    _data.amount.value,
    10**pricingDecimals,
    prices.priceFor(_data.amount.currency, pricingCurrency, _dat
  );
else return;
```

However, since `prices` is optional (it can be set to the zero address, as seen from the snippet), the conversion step can be skipped. When this happens, the contributor gets no NFT due to the early `return` even though the amount of their contribution might still be eligible for a tiered NFT.

Recommended Mitigation Steps

Short term, consider reverting when a different currency is used and `prices` is not set. Long term, consider supporting multiple currencies in the NFT delegate.

drgorillamd (Juicebox DAO) disputed

> This is poor project management from the project owner (not adding the appropriate price feed), not a vulnerability

> And there is no revert here as to not freeze the Juicebox project (NFT reward is an add-on, there is a full project running behind)

[Picodes (judge) commented]:

> As this finding:
>
> - would lead to a leak of value
> - is conditional on the project owner's mistake (that seems not so unlikely as they may think that one currency is enough and that they don't need to set `prices`)
> - but ultimately lead to a loss of funds for users
>
> I believe Medium severity to be appropriate

[M-06] Beneficiary credit balance can unwillingly be used to mint low tier NFT

Submitted by minhquanym

In the function `_processPayment()`, it will use provided `JBDidPayData` from `JBPaymentTerminal` to mint to the beneficiary. The `_value` from `JBDidPayData` will be sum up with previous `_credits` balance of beneficiary. There are 2 cases that beneficiary credit balance is updated in previous payment:

1. The payment received does not meet a minting threshold or is in excess of the minted tiers, the leftover amount will be stored as credit for future minting.
2. Clients may want to accumulate to mint higher tier NFT, they might specify that the previous payment should not mint anything. (Currently it's incorrectly implemented in case `_dontMint=true`, but sponsor confirmed that it's a bug)

In both cases, an attacker can pay a small amount (just enough to mint lowest tier NFT) and specify the victim to be the beneficiary. Function `__processPayment()` will use credit balance of beneficiary from previous payment to mint low-value tier.

For example, there are 2 tiers

1. Tier A: mintingThreshold = 20 ETH, votingUnits = 100
2. Tier B: mintingThreshold = 10 ETH, votingUnits = 10

Obviously tier A is much more better than tier B in term of voting power, so Alice (the victim) might want to accumulate her credit to mint tier A.

Assume current credit balance `creditsOf[Alice] = 19 ETH`. Now Bob (the attacker) can pay `1 ETH` and specify Alice as beneficiary and mint `2` Tier B NFT. Alice will have to receive `2` Tier B NFT with just `20 voting power` instead of `100 voting power` for a Tier A NFT.

Since these NFTs can be used in a governance system, it may create much higher impact if this governance is used to make important decision. E.g: minting new tokens, transfering funds of community.

Proof of Concept

Function `didPay()` only check that the caller is a terminal of the project

```
function didPay(JBDidPayData calldata _data) external payable vi
    // Make sure the caller is a terminal of the project, and th
    if (
      msg.value != 0 ||
      !directory.isTerminalOf(projectId, IJBPaymentTerminal(msg.
      _data.projectId != projectId
    ) revert INVALID_PAYMENT_EVENT();

    // Process the payment.
    _processPayment(_data);
}
```

Attacker can specify any beneficiary and use previous credit balance

```
// Keep a reference to the amount of credits the beneficiary alr
uint256 _credits = creditsOf[_data.beneficiary];

// Set the leftover amount as the initial value, including any c
uint256 _leftoverAmount = _value + _credits;
```

Recommended Mitigation Steps

Consider adding a config param to allow others from using beneficiary's credit balance. Its value can be default to `false` for every address. And if beneficiary want to, they can toggle this state for their address to allow other using their credit balance.

mejango (Juicebox DAO) acknowledged

> fancy. i think accumulating credits to "save up" is out of scope for this contract's design. Still a pretty cool pattern to note, thank you!

> yeah: if you are saving up for a specific nft, save up elsewhere, not through the credit system.

minhquanym (warden) commented:

> Thanks for your comments. Just put a note cause my writing might be vague. Saving up is just 1 case that I listed. The other case, funds are left after minting a specific tier in the docs.

> If a payment received does not meet a minting threshold or is in excess of the minted tiers, the balance is stored as a credit which will be added to future payments and applied to mints at that time.

[M-07] Deactivated tiers can still mint reserve tokens, even if no non-reserve tokens were minted.

Submitted by **Trust**

Tiers in Juicebox can be deactivated using the adjustTiers() function. It makes sense that reserve tokens may be minted in deactivated tiers, in order to be consistent with already minted tokens. However, the code allows the first reserve token to be minted in a deactivated tier, *even* though there was no previous minting of that tier.

```
function recordMintReservesFor(uint256 _tierId, uint256 _count)
  external
  override
  returns (uint256[] memory tokenIds)
{
  // Get a reference to the tier.
```

```
    JBStored721Tier storage _storedTier = _storedTierOf[msg.sender
    // Get a reference to the number of reserved tokens mintable f
    uint256 _numberOfReservedTokensOutstanding = _numberOfReserved
      msg.sender,
      _tierId,
      _storedTier
    );
    if (_count > _numberOfReservedTokensOutstanding) revert INSUFF
    ...
    for (uint256 _i; _i < _count; ) {
    // Generate the tokens.
    tokenIds[_i] = _generateTokenId(
      _tierId,
      _storedTier.initialQuantity - --_storedTier.remainingQuantit
    );
    unchecked {
      ++_i;
    }
  }



  function _numberOfReservedTokensOutstandingFor(
    address _nft,
    uint256 _tierId,
    JBStored721Tier memory _storedTier
  ) internal view returns (uint256) {
    // Invalid tier or no reserved rate?
    if (_storedTier.initialQuantity == 0 || _storedTier.reservedRa
    // No token minted yet? Round up to 1.
    // ******************* BUG HERE *********************
    if (_storedTier.initialQuantity == _storedTier.remainingQuanti
    ...
  }
```

Using the rounding mechanism is not valid when the tier has been deactivated, since we know there won't be any minting of this tier.

Impact

The reserve beneficiary receives an unfair NFT which may be used to withdraw tokens using the redemption mechanism.

Recommended Mitigation Steps

If Juicebox intends to use rounding functionality, pass an argument *isDeactivated* which, if true, deactivated the rounding logic.

mejango (Juicebox DAO) acknowledged

Picodes (judge) commented:

> The finding illustrates how a reserve token could be minted for a removed tier, and this token used to redeem funds.

cccz (warden) commented:

> This one seems to be a subset of this finding
> https://github.com/code-423n4/2022-10-juicebox-findings/issues/191

Picodes (judge) commented:

> Thank you for flagging, I will think about it!

Picodes (judge) commented:

> Although it is in the same lines and functionalities, I don't think this one is a subset of #191: this one is about the fact that you can still mint when it's deactivated, and #191 is about the rounding feature itself

[M-08] The tier reserved rate is not validated and can surpass `JBConstants.MAX_RESERVED_RATE`

Submitted by berndartmueller

https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/JBTiered721DelegateStore.sol#L1224-L1259
https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/JBTiered721DelegateStore.sol#L566
https://github.com/jbx-protocol/juice-nft-

rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/abstract/JB721Delegate.sol#L142

If the reserved rate of a tier is set to a value > `JBConstants.MAX_RESERVED_RATE`, the `JBTiered721DelegateStore._numberOfReservedTokensOutstandingFor` function will return way more outstanding reserved tokens (up to ~6 times more than allowed - **2^16 - 1** due to the manual cast of `reservedRate` to `uint16` divided by `JBConstants.MAX_RESERVED_RATE = 10_000`). This inflated value is used in the `JBTiered721DelegateStore.totalRedemptionWeight` function to calculate the cumulative redemption weight of all tokens across all tiers.

This higher-than-expected redemption weight will lower the `reclaimAmount` calculated in the `JB721Delegate.redeemParams` function. Depending on the values of `_data.overflow` and `_redemptionWeight`, the calculated `reclaimAmount` can be 0 (due to rounding down, see here) or a smaller than anticipated value, leading to burned NFT tokens from the user and no redemptions.

Impact

The owner of an NFT contract can add tiers with higher than usual reserved rates (and mint an appropriate number of NFTs to bypass all conditions in the `JBTiered721DelegateStore._numberOfReservedTokensOutstandingFor`), which will lead to a lower-than-expected redemption amount for users.

Proof of Concept

JBTiered721DelegateStore._numberOfReservedTokensOutstandingFor

```
function _numberOfReservedTokensOutstandingFor(
  address _nft,
  uint256 _tierId,
  JBStored721Tier memory _storedTier
) internal view returns (uint256) {
  // Invalid tier or no reserved rate?
  if (_storedTier.initialQuantity == 0 || _storedTier.reservedRa

  // No token minted yet? Round up to 1.
  if (_storedTier.initialQuantity == _storedTier.remainingQuanti

  // The number of reserved tokens of the tier already minted.
```

```
    uint256 _reserveTokensMinted = numberOfReservesMintedFor[_nft]

    // If only the reserved token (from the rounding up) has been r
    if (_storedTier.initialQuantity - _reserveTokensMinted == _sto
      return 0;

    // Get a reference to the number of tokens already minted in th
    uint256 _numberOfNonReservesMinted = _storedTier.initialQuanti
      _storedTier.remainingQuantity -
      _reserveTokensMinted;

    // Store the numerator common to the next two calculations.
    uint256 _numerator = uint256(_numberOfNonReservesMinted * _sto

    // Get the number of reserved tokens mintable given the number
    uint256 _numberReservedTokensMintable = _numerator / JBConstan

    // Round up.
    if (_numerator - JBConstants.MAX_RESERVED_RATE * _numberReserv
      ++_numberReservedTokensMintable;

    // Return the difference between the amount mintable and the am
    return _numberReservedTokensMintable - _reserveTokensMinted;
  }
```

JBTiered721DelegateStore.totalRedemptionWeight

The `JBTiered721DelegateStore._numberOfReservedTokensOutstandingFor` function is called from within the `JBTiered721DelegateStore.totalRedemptionWeight` function. This allows for inflating the total redemption weight.

```
  function totalRedemptionWeight(address _nft) public view overrid
    // Keep a reference to the greatest tier ID.
    uint256 _maxTierId = maxTierIdOf[_nft];

    // Keep a reference to the tier being iterated on.
    JBStored721Tier memory _storedTier;

    // Add each token's tier's contribution floor to the weight.
    for (uint256 _i; _i < _maxTierId; ) {
      // Keep a reference to the stored tier.
      _storedTier = _storedTierOf[_nft][_i + 1];
```

```
        // Add the tier's contribution floor multiplied by the quant
        weight +=
          (_storedTier.contributionFloor *
            (_storedTier.initialQuantity - _storedTier.remainingQuan
          _numberOfReservedTokensOutstandingFor(_nft, _i, _storedTie

        unchecked {
          ++_i;
        }
      }
    }
```

JBTiered721Delegate._totalRedemptionWeight

`JBTiered721DelegateStore.totalRedemptionWeight` is called in the `JBTiered721Delegate._totalRedemptionWeight` function.

```
    function _totalRedemptionWeight() internal view virtual override
      return store.totalRedemptionWeight(address(this));
    }
```

abstract/JB721Delegate.redeemParams

This `JBTiered721Delegate._totalRedemptionWeight` function is then called in the `JB721Delegate.redeemParams` function, which ultimately calculates the `reclaimAmount` given an overflow and `_decodedTokenIds`.

`uint256 _base = PRBMath.mulDiv(_data.overflow, _redemptionWeight, _total);` in line 142 will lead to a lower `_base` due to the inflated denumerator `_total`.

```
    function redeemParams(JBRedeemParamsData calldata _data)
      external
      view
      override
      returns (
        uint256 reclaimAmount,
        string memory memo,
```

```
    JBRedemptionDelegateAllocation[] memory delegateAllocations
  )
{
  // Make sure fungible project tokens aren't being redeemed too
  if (_data.tokenCount > 0) revert UNEXPECTED_TOKEN_REDEEMED();

  // Check the 4 bytes interfaceId and handle the case where the
  if (
    _data.metadata.length < 4 || bytes4(_data.metadata[0:4]) !=
  ) {
    revert INVALID_REDEMPTION_METADATA();
  }

  // Set the only delegate allocation to be a callback to this c
  delegateAllocations = new JBRedemptionDelegateAllocation[](1);
  delegateAllocations[0] = JBRedemptionDelegateAllocation(this,

  // If redemption rate is 0, nothing can be reclaimed from the
  if (_data.redemptionRate == 0) return (0, _data.memo, delegate

  // Decode the metadata
  (, uint256[] memory _decodedTokenIds) = abi.decode(_data.metad

  // Get a reference to the redemption rate of the provided toke
  uint256 _redemptionWeight = _redemptionWeightOf(_decodedTokenI

  // Get a reference to the total redemption weight.
  uint256 _total = _totalRedemptionWeight(); // @audit-info Uses

  // Get a reference to the linear proportion.
  uint256 _base = PRBMath.mulDiv(_data.overflow, _redemptionWeig

  // These conditions are all part of the same curve. Edge condi
  if (_data.redemptionRate == JBConstants.MAX_REDEMPTION_RATE)
    return (_base, _data.memo, delegateAllocations);

  // Return the weighted overflow, and this contract as the dele
  return (
    PRBMath.mulDiv(
      _base,
      _data.redemptionRate +
        PRBMath.mulDiv(
          _redemptionWeight,
          JBConstants.MAX_REDEMPTION_RATE - _data.redemptionRate
          _total
        ),
```

```
            JBConstants.MAX_REDEMPTION_RATE
        ),
        _data.memo,
        delegateAllocations
      );
    }
```

Recommended mitigation steps

Consider validating the tier reserved rate `reservedRate` in the `JBTiered721DelegateStore.recordAddTiers` function to ensure the reserved rate is not greater than `JBConstants.MAX_RESERVED_RATE`.

mejango (Juicebox DAO) confirmed

Low Risk and Non-Critical Issues

For this contest, 49 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **berndartmueller** received the top score from the judge.

The following wardens also submitted reports: **brgltd**, **SaharAP**, **minhtrng**, **0xSmartContract**, **joestakey**, **d3e4**, **peanuts**, **svskaushik**, **bharg4v**, **delfin454000**, **Trust**, **Aymen0909**, **V_B**, **a12jmx**, **0x4non**, **cryptostellar5**, **Diana**, **ReyAdmirado**, **__141345__**, **ret2basic**, **cryptphi**, **tnevler**, **Jeiwan**, **carlitox477**, **lukris02**, **erictee**, **mcwildy**, **hansfriese**, **RaymondFam**, **ignacio**, **LeoS**, **0xNazgul**, **ch0bu**, **karanctf**, **slowmoses**, **RaoulSchaffranek**, **yixxas**, **RedOneN**, **fatherOfBlocks**, **Lambda**, **BClabs**, **cloudjunky**, **Rolezn**, **seyni**, **0x1f8b**, **ladboy233**, **csanuragjain**, *and* **chaduke**.

[L-01] `JBTiered721Delegate.tokenURI` should throw an error if `_tokenId` is not a valid NFT

According to `EIP-721` and specifically, the metadata extension, the `tokenURI` function should throw an error if `_tokenId` is not a valid NFT. Contrary, the current implementation returns an empty string.

Findings

JBTiered721Delegate.sol#L140

```
function tokenURI(uint256 _tokenId) public view override returns
  // A token without an owner doesn't have a URI.
  if (_owners[_tokenId] == address(0)) return ''; // @audit-info

  // Get a reference to the URI resolver.
  IJBTokenUriResolver _resolver = store.tokenUriResolverOf(addre

  // If a token URI resolver is provided, use it to resolve the
  if (address(_resolver) != address(0)) return _resolver.getUri(

  // Return the token URI for the token's tier.
  return
    JBIpfsDecoder.decode(
      store.baseUriOf(address(this)),
      store.encodedTierIPFSUriOf(address(this), _tokenId)
    );
}
```

Recommended mitigation steps

Consider throwing an error if `_tokenId` is not a valid NFT.

[L-02] Decoding an IPFS hash using a fixed hash function and length of the hash

An IPFS hash specifies the hash function and length of the hash in the first two bytes of the hash. The first two bytes are **0x1220**, where **12** denotes that this is the SHA256 hash function and **20** is the length of the hash in bytes (32 bytes).

Although SHA256 is 32 bytes and is currently the most common IPFS hash function, other content could use a hash function that is larger than 32 bytes. The current implementation limits the usage to the SHA256 hash function and a hash length of 32 bytes.

Findings

libraries/JBIpfsDecoder.sol#L28

```
function decode(string memory _baseUri, bytes32 _hexString)
  external
  pure
  returns (string memory)
{
  // Concatenate the hex string with the fixed IPFS hash part (0:
  bytes memory completeHexString = abi.encodePacked(bytes2(0x122

  // Convert the hex string to an hash
  string memory ipfsHash = _toBase58(completeHexString);

  // Concatenate with the base URI
  return string(abi.encodePacked(_baseUri, ipfsHash));
}
```

Recommended mitigation steps

Consider using a more generic implementation that can handle different hash functions and lengths and allow the user to choose.

[L-03] The tier id can potentially surpass 16 bits leading to token id collisions

The token id is composed of the given tier id `_tierId` and the number of the token `_tokenNumber` in the tier. The tier id is limited to 16 bits, which means that there can **theoretically** only exist 65,535 tiers *(this is very unlikely as this would have more serious consequences on other parts of the system and will cause a serious denial of service caused by unbounded loops. Still, theoretically, it's possible and there is no check in place)*.

If more than 65,535 tiers exist, the 16 bits reserved for the tier id will be surpassed and overwritten by `_tokenNumber`. This will lead to token id collisions with other tiers with a lower tier id.

Findings

[JBTiered721DelegateStore._generateTokenId]

```
function _generateTokenId(uint256 _tierId, uint256 _tokenNumber)
  internal
```

```
    pure
    returns (uint256 tokenId)
  {
    // The tier ID in the first 16 bits.
    tokenId = _tierId;

    // The token number in the rest.
    tokenId |= _tokenNumber << 16;
  }
```

Recommended mitigation steps

Consider reverting if the `_tierId` is > 16 bits.

drgorillamd (Juicebox DAO) commented:

> [L-01]: Doc
> [L-02]: Mitigated by a custom uri resolver (if/when ipfs hashes change their length and/or algo)
> [L-03]: Mitigated

Gas Optimizations

For this contest, 34 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Jeiwan** received the top score from the judge.

The following wardens also submitted reports: brgltd, 0xSmartContract, JC, zishansami, lukris02, berndartmueller, bharg4v, CodingNameKiki, Aymen0909, sakman, 0x4non, Shinchan, ReyAdmirado, JrNet, Saintcode_, Awesome, __141345__, DimSon, emrekocak, 0x5rings, cryptostellar5, Diana, carlitox477, mcwildy, RaymondFam, LeoS, ch0bu, Bnke0x0, trustindistrust, gogo, martin, 0x1f8b, *and* chaduke.

[G-01] Optimize NFT delegate deployments by using proxy

https://github.com/jbx-protocol/juice-nft-rewards/blob/f9893b1497098241dd3a664956d8016ff0d0efd0/contracts/JBTiered721DelegateDeployer.sol#L115

The cost of NFT delegate deployments can be significantly reduced by deploying proxies instead of clones of the implementation.

Proof of Concept

This function is used to deploy new NFT delegates (JBTiered721DelegateDeployer.sol#L115):

```
function _clone(address _targetAddress) internal returns (addres
  assembly {
    // Get deployed/runtime code size
    let _codeSize := extcodesize(_targetAddress)

    // Get a bit of freemem to land the bytecode, not updated as
    let _freeMem := mload(0x40)

    // Shift the length to the length placeholder, in the constr
    let _mask := mul(_codeSize, 0x1000000000000000000000000000000
    // Insert the length in the correct sport (after the PUSH3 /
    let _initCode := or(_mask, 0x620000006000816000d8239f3fe00000

    // Store the deployment bytecode
    mstore(_freeMem, _initCode)

    // Copy the bytecode (our initialise part is 13 bytes long)
    extcodecopy(_targetAddress, add(_freeMem, 13), 0, _codeSize)

    // Deploy the copied bytecode
    _out := create(0, _freeMem, _codeSize)
  }
```

It copies the code of an existing contract (`JBTiered721Delegate` , `JB721TieredGovernance` , or `JB721GlobalGovernance`) and deploys a new contract with the same code. This is a costly operation because each of the three contracts is a big contract with a lot of code. It'll be much cheaper to deploy non-upgradable proxies instead.

Recommended Mitigation Steps

Consider using the Clones library from OpenZeppelin—it deploys and absolutely minimal non-upgradable proxy contract. Such proxies, however, cannot be verified

on Etherscan. Some more info.

Picodes (judge) commented:

> Depending on the number of deployments this could be the biggest gas saving so far.

drgorillamd (Juicebox DAO) commented:

> @Picodes (judge) we didn't use proxies for 2 reasons (it would have obviously been easier;):
>
> - this is shifting the gas burden -> each call cost an extra call() cost to the users (on a cold address, that's at least 2600)
> - the saving of deploying a proxy is a one off, for the project owner, while the gas saved on every call is cumulative through time (and might end up being bigger)
>
> + even if using a non-upgradeable proxy, some users have concern with such (I know, ux/docs/education is out of scope;)
>
> In summary, not convinced this would be the biggest gas saving, on an overall basis

Picodes (judge) commented:

> Indeed it totally depends on the usage!
>
> Giving this option to users could easily save a lot of gas for projects that expect only a few transactions. I also selected this report as it's the only one suggesting this.
>
> The deployment of the clone contract would be only `<50k` gas and then per call `<2k` (700 for the `DELEGATECALL` , `2600` for the cold address and then the memory expansion) so it'd be worth it for projects with less than a few hundred transactions.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top