e

CD SECURITY

# AUDIT REPORT

## Beezie

November 2024


Prepared by

Pelz

yotov721

Arnie

# Introduction

A time-boxed security review of the **Beezie** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

# 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.

# About **Beezie**

Beezie is a protocol that lets users sell tokenized real-world assets - featuring everything from trading cards to watches.

Admins can mint tokenized NFTs to a seller that deposited the collectible asset. The seller can then commit a Drop where the collectibles are revealed at the end.

When users participate in the Drop they are minted a drop token.

Once the Drop is over, unsold items are transferred back to the seller and users can then swap their drop token for the tokenized collectible asset. Gelato Randomness Provider is used to randomize the received collectible if the Drop is fully sold out.

The user can then resell the tokenized asset or claim it's physical item where the collectible NFT is burned.

# Severity classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

**Impact** - the technical, economic, and reputation damage of a successful attack

**Likelihood** - the chance that a particular vulnerability gets discovered and exploited

**Severity** - the overall criticality of the risk

# Security Assessment Summary

***review commit hash* - d1ab65e7e6dc233ebf8825a2e1d99ead0666d9f6**

Scope

The following smart contracts were in scope of the audit:

- `src/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 3 issues
- Medium: 3 issues
- Low & Info: 11 issues

---

# Findings Summary

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-01] | Improper native token handling in `matchBid` and `fulfillOrder` causes transaction failures | High | Acknowledged |
| [H-02] | Revert in `closeSuccessfulDrop` can cause tokens to be permanently stuck | High | Fixed |
| [H-03] | Users can refund before the sale has ended | High | Fixed |
| [M-01] | Lack of expiration and nonce mechanism in signature handling | Medium | Acknowledged |
| [M-02] | Missing type hash when hashing sub-struct breaks EIP-712 compliance | Medium | Acknowledged |
| [M-03] | Arrays hashed wrong when tokenizing breaks EIP-712 | Medium | Acknowledged |
| [L-01] | Incorrect event emission in `BidFulfilled` | Low | Acknowledged |
| [L-02] | Inconsistent use of `_msgSender()` in Meta-Transaction enabled contracts | Low | Fixed |
| [L-03] | Off-by-One error in `matchBid` expiration check | Low | Fixed |
| [L-04] | Missing validation for `saleStartTime` enables invalid drop proposals | Low | Acknowledged |
| [L-05] | Unused variables in `ColonyDropManager` fail to apply constraints | Low | Fixed |
| [L-06] | Use SafeERC20 to transfer tokens | Low | Acknowledged |
| [L-07] | Use `safeTransfer` and `safeMint` for NFTs | Low | Acknowledged |
| [L-08] | Signatures can be replayed across different chains | Low | Acknowledged |

| ID | Title | Severity | Status |
|---|---|---|---|
| [I-01] | Use concrete pragma | Informational | Acknowledged |
| [I-02] | A user may use multiple wallets to bypass the max purchase amount | Informational | Acknowledged |
| [I-03] | Some important functions do not emit events | Informational | Acknowledged |

# Detailed Findings

## [H-01] Improper native token handling in `matchBid` and `fulfillOrder` causes transaction failures

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

The `matchBid` function in `ColonyBidRouter.sol` fails when the `considerationToken` specified in the `order` parameter is a native token (e.g., ETH). This occurs because the function attempts to interact with the `IERC20` interface regardless of whether the token is native or ERC20-compliant:

```
IERC20 considerationToken = IERC20(order.considerationToken);
considerationToken.transferFrom(bid.bidder, address(this), bid.bidAmount);
considerationToken.approve(SEAPORT, bid.bidAmount);
ISeaport(SEAPORT).fulfillBasicOrder{value: msg.value}(order);
```

For native tokens, both the `transferFrom` and `approve` calls will fail because they are not applicable to ETH. This will result in a transaction revert whenever ETH is used as the `considerationToken`.

Moreover, the `fulfillOrder` function in the `SeaportProxy` contract does not account for cases where the consideration token is the native token (e.g., ETH). This results in a failure when attempting to process such orders.

The function uses the following logic to handle the consideration token:

```
IERC20 considerationToken = IERC20(order.considerationToken);
considerationToken.transferFrom(_msgSender(), address(this),
totalConsiderationAmount);
considerationToken.approve(SEAPORT, totalConsiderationAmount);
ISeaport(SEAPORT).fulfillBasicOrder{value: msg.value}(order);
```

This implementation assumes the consideration token is an ERC20 token and attempts to call `transferFrom` and `approve` methods on it. However, if the `considerationToken` is the native token (e.g., ETH), these calls will fail because native tokens do not support the ERC20 interface.

The issue leads to failed transactions whenever users attempt to fulfill an order with a native token as the consideration. This significantly limits the functionality of the `fulfillOrder` function and impacts usability.

### Recommendations

Implement a conditional check to handle native tokens separately. For example:

```
if (order.considerationToken == address(0)) {
    // Native token handling
    if (msg.value < bid.bidAmount) {
        revert InsufficientEthSent();
    }
    ISeaport(SEAPORT).fulfillBasicOrder{value: bid.bidAmount}(order);
} else {
    // ERC20 token handling
    IERC20 considerationToken = IERC20(order.considerationToken);
    considerationToken.transferFrom(bid.bidder, address(this),
bid.bidAmount);
    considerationToken.approve(SEAPORT, bid.bidAmount);
    ISeaport(SEAPORT).fulfillBasicOrder(order);
}
```

## [H-02] Revert in `closeSuccessfulDrop` can cause tokens to be permanently stuck

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

The `closeSuccessfulDrop` function in the `ColonyDropManager` contract can fail due to an underflow in the payout calculation logic. This happens when the combined values of `totalPaidOut` and `commissionFee` exceed `totalRaised`. The failure results in a revert, making it impossible to close the drop and release funds. Without an emergency recovery mechanism, any tokens remaining in the contract will be permanently locked.

#### Root Cause

The vulnerability lies in the `_handleCloseSuccesfulDropPayouts` function. Specifically, the issue arises during the calculation of `amountToSendToCreator`:

**Vulnerable Code**

```
uint256 amountToSendToCreator = (totalRaised - _amountRaised.totalPaidOut)
- commissionFee;
```

If the sum of `_amountRaised.totalPaidOut` and `commissionFee` is greater than `totalRaised`, the subtraction causes an underflow. For instance:

1. Assume:
    - `totalRaised = 600`
    - `totalPaidOut = 500`
    - `commissionFee = 120` (20% of `totalRaised`)
2. The calculation for `amountToSendToCreator` becomes:

    ```
    amountToSendToCreator = (600 - 500) - 120; // -20%
    ```

3. Since `amountToSendToCreator` is unsigned, this underflow triggers a revert.

## Recommendation

Ensure that `totalPaidOut` and `commissionFee` do not exceed `totalRaised` before performing the calculation. For example:

```
require(totalPaidOut + commissionFee <= totalRaised, "Invalid payout
parameters");
```

Another way can be to modify the payout logic to prevent underflows:

```
uint256 availableFunds = totalRaised > totalPaidOut ? totalRaised -
totalPaidOut : 0;
uint256 amountToSendToCreator = availableFunds > commissionFee ?
availableFunds - commissionFee : 0;
```

This ensures `amountToSendToCreator` never becomes negative.

# [H-03] Users can refund before the sale has ended

## Severity

**Impact:** High

**Likelihood:** Medium

## Description

The function `claimRefund` allows a user to claim a refund if the drop was not successful. The problem occurs because of incorrect inequality checks that allow a user to refund while purchases of said drop are still possible.

```
    function claimRefund(uint256 dropId, uint256[] calldata
wrappedTokenIds) external {
        Drop memory drop = _drops[dropId];
        uint256 dropStartTokenId = dropToStartTokenId[dropId];
        if (drop.tokensSold >= drop.minSellOutTokens) {
            _revert(RefundNotAvailable.selector);
        }
        if (block.timestamp < drop.saleEndTime) {
            _revert(RefundNotAvailable.selector);
```

Let's focus on the last if statement, if the timestamp is equal to the end time then the logic execution continues. This is an error because at the time `block.timestamp == drop.saleEndTime` the drop is still live and allows users to make purchases of the drop. We can observe this from the snippet below from the purchaseDrop function

```
        if (block.timestamp > drop.saleEndTime)
_revert(SaleEnded.selector);
```

as we can see there is a time period where `block.timestamp == drop.saleEndTime` that allows both refunding and purchasing of the drop.

## Recommendations

Change the inequality to <=

```
        if (block.timestamp <= drop.saleEndTime) {
            _revert(RefundNotAvailable.selector);
```

# [M-01] Lack of expiration and nonce mechanism in signature handling

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `cancelOrder` function in `seaportProxy.sol` relies on a signature validation mechanism that does not include a deadline or dynamic nonce for signature invalidation. The `CANCEL_TYPEHASH` used in the signature calculation is defined as:

```
bytes32 constant CANCEL_TYPEHASH =
    keccak256("Cancel(address offerer,address zone,uint256 salt,bytes32
offerHash,bytes32 considerationHash)");
```

Although the `salt` provides some variability, it does not inherently ensure uniqueness. If a user inadvertently creates multiple orders with the same `salt`, the signature associated with the `salt` remains valid even after cancellation. This enables a potential reuse of signatures and risks unintended cancellations or order manipulations.

### Vulnerability Details

1. **Reusable Signatures:**
   Once an order is canceled, the associated signature is invalidated. However, a new order with the same `salt` can reuse the old signature, leading to ambiguity and potential abuse.

2. **No Expiration Mechanism:**
   The lack of a deadline in the signature hash means that signatures do not naturally expire, increasing the risk of prolonged validity and potential misuse in future interactions.

3. **Incomplete Invalidation:**
   The `_checkCancelSignature` function does not dynamically consider parameters like a deadline or incremental nonce, making it difficult to effectively manage signature lifecycle.

## Recommendations

1. **Introduce a Deadline:**
   Add a `uint256 deadline` parameter to the `CANCEL_TYPEHASH` and ensure it is part of the signature validation process. For example:

   ```
   bytes32 constant CANCEL_TYPEHASH =
       keccak256("Cancel(address offerer,address zone,uint256
   salt,uint256 deadline,bytes32 offerHash,bytes32 considerationHash)");
   ```

   Include a validation step to ensure the deadline has not passed:

   ```
   if (block.timestamp > deadline) {
       revert SignatureExpired(deadline);
   }
   ```

2. **Implement Nonce Mechanism:**
   Introduce a nonce for each user that is incremented every time an order is created or canceled. Ensure the nonce is included in the signature calculation and update it during order cancellation.

```
uint256 public nonce; // Per-user nonce tracking

function incrementNonce() external {
    nonce++;
}
```

   Modify the `CANCEL_TYPEHASH` to include the nonce:

```
bytes32 constant CANCEL_TYPEHASH =
    keccak256("Cancel(address offerer,address zone,uint256
salt,uint256 nonce,bytes32 offerHash,bytes32 considerationHash)");
```

# [M-02] Missing type hash when hashing sub-struct breaks EIP-712 compliance

## Severity

**Impact:** Low

**Likelihood:** High

## Description

In `SeaportProxy` there is EIP-712 implementation with **nested arrays of structs** in the typed data which does not follow EIP-712.

**Issue 1:**

As stated in EIP-712 #Definition of encodeType:

> If the struct type references other struct types (and these in turn reference even more struct types), then the set of referenced struct types is collected, sorted by name and appended to the encoding. An example encoding is *Transaction(Person from,Person to,Asset tx)Asset(address token,uint256 amount)Person(address wallet,string name)*.

In the case of the protocol the typed data is `Cancel(address offerer,address zone,uint256 salt,bytes32 offerHash,bytes32 considerationHash)` where `offerHash` and `considerationHash` are of types `OfferItem[]` and `ConsiderationItem[]`, but are set as `bytes32`.

**Issue 2:**

Stated in EIP-712 #Definition of encodeData:

> The array values are encoded as the keccak256 hash of the concatenated encodeData of their contents (i.e. the encoding of SomeType[5] is identical to that of a struct containing five members of type SomeType). The struct values are encoded recursively as hashStruct(value).

As stated for the `hashStruct`:

> The hashStruct function is defined as hashStruct(s : 𝕊) = keccak256(typeHash || encodeData(s)) where typeHash = keccak256(encodeType(typeOf(s)))

Basically meaning the hash struct is the hashed concatenated value of the typehash and encoded data. Having all this in mind the issue here is that when the sub-struct array items are hashed the type hash is missing:

```
    function hashOfferItem(OfferItem memory offerItem) internal view
returns (bytes32) {
        return keccak256(
            abi.encode( // missing typehash
                offerItem.itemType,
                offerItem.token,
                offerItem.identifierOrCriteria,
                offerItem.startAmount,
                offerItem.endAmount
            )
        );
    }

    function hashConsiderationItem(ConsiderationItem memory
considerationItem) internal view returns (bytes32) {
        return keccak256(
            abi.encode( // missing typehash
                considerationItem.itemType,
                considerationItem.token,
                considerationItem.identifierOrCriteria,
                considerationItem.startAmount,
                considerationItem.endAmount,
                considerationItem.recipient
            )
        );
    }
```

## Note

Example array structs hashing implementation: https://github.com/0xFloop/eip712-struct-array/blob/master/src/EIP712SigConsumer.sol Example analogical issue: An analogical implementation can be found in this stack overflow post https://ethereum.stackexchange.com/questions/151513/eip712-typeddata-encoding-with-nested-array-of-structs-returning-wrong-signer

## Recommendations

Change the `Cancel` type hash to appending both nested types in alphabetical order

```
+ bytes32 constant CANCEL_TYPEHASH =
keccak256(abi.encodePacked("Cancel(address offerer,address zone,uint256
salt,OfferItem[] offerHash,ConsiderationItem[]
considerationHash)ConsiderationItem(ItemType itemType,address
token,uint256 identifierOrCriteria,uint256 startAmount,uint256
endAmount,address recipient)OfferItem(ItemType itemType,address
token,uint256 identifierOrCriteria,uint256 startAmount,uint256
endAmount)"));
```

Create type hashes for the offer item and consideration item structs

```
+ bytes32 constant OFFER_ITEM_TYPE_HASH =
keccak256(abi.encodePacked("OfferItem(ItemType itemType,address
token,uint256 identifierOrCriteria,uint256 startAmount,uint256
endAmount)"));
+ bytes32 constant CONSIDERATION_ITEM_TYPE_HASH =
keccak256(abi.encodePacked("ConsiderationItem(ItemType itemType,address
token,uint256 identifierOrCriteria,uint256 startAmount,uint256
endAmount,address recipient)"));
```

Add type hash when hashing separate array items:

```
    function hashOfferItem(OfferItem memory offerItem) internal view
returns (bytes32) {
        return keccak256(
            abi.encode(
+               OFFER_ITEM_TYPE_HASH
                offerItem.itemType,
                offerItem.token,
                offerItem.identifierOrCriteria,
                offerItem.startAmount,
                offerItem.endAmount
            )
        );
    }

    function hashConsiderationItem(ConsiderationItem memory
considerationItem) internal view returns (bytes32) {
        return keccak256(
            abi.encode(
+               CONSIDERATION_ITEM_TYPE_HASH
                considerationItem.itemType,
                considerationItem.token,
                considerationItem.identifierOrCriteria,
                considerationItem.startAmount,
                considerationItem.endAmount,
```

```
                considerationItem.recipient
            )
        );
    }
```

# [M-03] Arrays hashed wrong when tokenizing breaks EIP-712

## Severity

**Impact:** Low

**Likelihood:** High

## Description

The `ColonyCollectibles::tokenize()` function lets an admin tokenize a number of tokens for a given receiver/s. It implements EIP-712 signature verification to execute the mint.

The problem is the way the data for the digest is hashed. The `tokenize` function calls `getTokenizeDigest` which hashes the data as follows (where `receivers` and `amounts` are arrays):

```
return MessageHashUtils.toEthSignedMessageHash(
  keccak256(
    abi.encodePacked(
      address(this),
      block.chainid,
      signerCounter,
      receivers,
      amounts,
      dbIdentifier
    )
  )
);
```

This is wrong because as stated in EIP-712 *The array values are encoded as the keccak256 hash of the concatenated encodeData of their contents (i.e. the encoding of SomeType[5] is identical to that of a struct containing five members of type SomeType).* Or in other words each array element has to be hashed separately before being added to the digest

Note that the same issue is present in the `claimPhysicals` function where the `tokenIds` array is hashed directly. Also, present in the `TransferMiddleware::createTransferWithAuthorizationDigest` and `ColonyDropManager::createBuybackDigest` functions.

## Recommendations

Add functions that hash the elements of an array separately and call them from `getTokenizeDigest`

```
    function hashAddressArray(address[] memory receivers) internal pure
returns (bytes32) {
        bytes32[] memory hashes = new bytes32[](receivers.length);
        for (uint256 i = 0; i < receivers.length; ++i) {
            hashes[i] = keccak256(abi.encode(receivers[i]));
        }
        return keccak256(abi.encodePacked(hashes));
    }

    function hashUintArray(uint256[] memory amounts) internal pure returns
(bytes32) {
        bytes32[] memory hashes = new bytes32[](amounts.length);
        for (uint256 i = 0; i < amounts.length; ++i) {
            hashes[i] = keccak256(abi.encode(amounts[i]));
        }
        return keccak256(abi.encodePacked(hashes));
    }
```

```
    function getTokenizeDigest(
        address[] calldata receivers,
        uint256[] calldata amounts,
        uint64 dbIdentifier
    ) public view returns (bytes32) {
        return toEthSignedMessageHash(
            keccak256(
                abi.encode(
                    TOKENIZE_DIGEST_TYPEHASH,
                    address(this),
                    block.chainid,
                    signerCounter,
-                   receivers
-                   amounts
+                   hashAddressArray(receivers),
+                   hashUintArray(amounts),
                    dbIdentifier
                )
            )
        );
    }
```

# [L-01] Incorrect event emission in `BidFulfilled`

## Description

The `BidFulfilled` event in the `BidRouter` contract incorrectly emits the `offerAmount` in place of the `bidAmount`. This discrepancy arises from the way the `bidAmount` is calculated, which involves summing

the `considerationAmount` and the amounts from `additionalRecipients` in the `matchBid` function.

The `BidFulfilled` event is defined as:

```
event BidFulfilled(
    address bidder,
    address fulfiller,
    uint256 salt,
    address paymentToken,
    uint256 bidAmount,
    address collection,
    uint256 tokenId
);
```

However, during emission, the `offerAmount` from the `order` is used instead of the correctly calculated `bidAmount`:

```
emit BidFulfilled(
    bid.bidder,
    order.offerer,
    bid.salt,
    order.considerationToken,
    order.offerAmount, // Incorrect: This should be `bid.bidAmount`
    order.offerToken,
    order.offerIdentifier
);
```

Since the `bidAmount` accounts for all the consideration amounts (including those of additional recipients), emitting `offerAmount` instead can lead to discrepancies in event logs and potentially misleading external systems or users relying on the emitted event for accurate data.

## Recommendations

Update the `emit BidFulfilled` statement to use the correct `bidAmount` from the `Bid` struct:

```
emit BidFulfilled(
    bid.bidder,
    order.offerer,
    bid.salt,
    order.considerationToken,
    bid.bidAmount, // Correct: Use the calculated bid amount
    order.offerToken,
    order.offerIdentifier
);
```

# [L-02] Inconsistent use of `_msgSender()` in Meta-Transaction enabled contracts

## Description

The `ColonyCollectibles` and `ColonyDrops` contracts inherit the `ERC2771Context`, enabling meta-transactions by overriding the behavior of `msg.sender` to be replaced with `_msgSender()`. However, some critical checks in the contracts inconsistently use `msg.sender` instead of `_msgSender()`.

### Affected Code in `ColonyCollectibles`

1. **Minter Check**:

```
function _onlyMinter() internal view {
    if (!_isMinter[msg.sender] && msg.sender != owner()) { // @audit-issue
should use _msgSender()
        revert NotMinterOrOwner();
    }
}
```

2. **Owner Check**:

```
function _onlyOwner() internal view {
    if (msg.sender != owner()) { // @audit-issue should use _msgSender()
        revert NotOwner();
    }
}
```

### Affected Code in `ColonyDrops`

1. **Drop Manager Check**:

```
function _onlyColonyDropManager() public view {
    if (msg.sender != address(dropManager)) { // @audit-issue should use
_msgSender()
        revert SenderNotColonyDropManager();
    }
}
```

## Recommendations

To ensure compatibility with meta-transactions, replace all instances of `msg.sender` with `_msgSender()` in functions that depend on the sender's identity.

# [L-03] Off-by-One error in `matchBid` expiration check

## Description

In the `matchBid` function within `colonyBidRouter.sol`, the expiration check:

```
if (block.timestamp > bid.expiration) {
    _revert(BidInvalidExpiration.selector);
}
```

contains an **off-by-one error**. This condition permits bids to be matched at the exact block where the expiration time is reached (`block.timestamp == bid.expiration`). Such behavior extends the validity of the bid by one block, allowing it to be matched even though it should have expired.

This could lead to unintended bid fulfillment, especially in scenarios where the timing of bids is critical.

## Recommendations

Update the expiration check to use `>=` to ensure that bids are invalidated precisely when the expiration time is reached:

```
if (block.timestamp >= bid.expiration) {
    _revert(BidInvalidExpiration.selector);
}
```

This adjustment ensures that bids cannot be matched once their expiration time has been reached, aligning with the intended logic.

# [L-04] Missing validation for `saleStartTime` enables invalid drop proposals

## Description

The `commitDrop` function in `colonyDropManager.sol` does not validate the `saleStartTime` parameter. This allows proposers to:

1. **Set `saleStartTime` to `0`:** A zero start time bypasses proper initialization, potentially allowing unauthorized edits to drop.
2. **Set `saleStartTime` to a past timestamp:** A start time earlier than `block.timestamp` can immediately invalidate the drop or cause unintended behavior.

### Code Example

```
if ($._drops[dropId].saleStartTime != 0) {
    _revert(DropAlreadyExists.selector);
}
```

The current check only prevents duplicate drops but does not ensure `saleStartTime` is valid and this check can still be bypassed if the proposer sets the `startTimestamp` to 0.

### Recommendations

Add the following checks in `commitDrop` to ensure a valid `saleStartTime`:

```
if (_dropData.saleStartTime < block.timestamp) {
    _revert(SaleStartMustBeInFuture.selector);
}
```

By validating `saleStartTime`, you can prevent malicious or erroneous drop proposals and ensure smooth contract operation.

## [L-05] Unused variables in `ColonyDropManager` fail to apply constraints

### Description

the protocol lets proposers create drops for their NFTs. The proposers call the `ColonyDropManager::commitDrop` function in which they pass the ID for the drop and additional data. There are two constants defined in the `ColonyDropManager` contract that are never used:

```
uint256 constant MAX_ITEMS_PER_DROP = 100_000;
uint256 constant FORBIDDEN_DROP_ID = 0;
```

This will let proposers create a drop with id 0 or a drop with more than 100_000 NFTs, which according to the constants should not be possible.

### Recommendations

Integrate the constants in the code or remove them

## [L-06] Use SafeERC20 to transfer tokens

### Description

The protocol lets users bid on orders through the `ColonyBidRouter::matchBid` function where the pay token is an ERC20. The problem is that the token may be a weird ERC20 that does not revert on transfer. Also, the return value of the transfer is NOT checked, which could cause problems if the token does not return any value on transfer.

If the return value is not checked this could lead to a loss of funds for users, as the token may be paused

### Recommendations

Use the `SafeERC20` library implementation from OpenZeppelin and call `safeTransfer` or `safeTransferFrom` when transferring ERC20 tokens.

## [L-07] Use `safeTransfer` and `safeMint` for NFTs

### Description

On several occasions, the protocol transfers NFTs to users using the `transferFrom` function. One of them is the `ColonyBidRouter::matchBid` function where the NFT is transferred to the bidder. The problem with using `transferFrom` is that the receiver wallet may be a multisig or some other non-EOA that does NOT support NFTs. This would lead to the user losing the NFT. Note that this opens the possibility of reentrancy so add a `nonReentrant` guard as well.

Other functions include `ColonyCollectibles::bulkTransferFrom`, `ColonyDropManager::_executeSwapAndGetLeavesInnerTransfer` and use `safeMint` in `ColonyDrops::bulkMint`

### Recommendations

Use safeTransfer and safeMint to handle NFT transfers.

## [L-08] Signatures can be replayed across different chains

### Description

The protocol is meant to migrate from to a different chain, this means that signatures should include a domain separator to ensure that cross-chain replay attacks are not possible. Currently, the protocol does not implement such safeguards as can be seen from a signature verification from the snippet below.

```
        if (!SignatureChecker.isValidSignatureNow(bid.bidder,
bidTypedDataHash, bidSignature)) {
            _revert(BidInvalidSignature.selector);
```

Due to the likelihood, the severity is still low. This is because the protocol does implement deadline checks which should somewhat stop this attack from taking place, but not entirely.

### Recommendations

Add logic to ensure that sigs cannot be replayed across different chains

## [I-01] Use concrete pragma

### Description

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. The project is currently using multiple floatable versions - `pragma solidity ^0.8.13;` and `pragma solidity 0.8.4;`. Furthermore, consider using a newer version of the compiler as the latest available version is 0.8.26.

## [I-02] A user may use multiple wallets to bypass the max purchase amount

### Description

In the `ColonyDropManager::purchaseDrop` function there is a check if the user has purchased more than the max amount of NFTs for the given drop already to revert.

```
UserDropStats storage userDropStats = _userDropStats[purchaser][id];
uint256 amountPurchased = userDropStats.numPurchased;

if (amountPurchased + amount > drop.maxPurchasesPerWallet) {
    _revert(TooManyPurchases.selector);
}
```

This condition however can be easily bypassed by a user using multiple different wallets to purchase NFTs of the drop.

## [I-03] Some important functions do not emit events

### Description

Many functions in `colonyDropManager.sol` does not emit any event. Given the importance of these functions, it is important to emit events when a successful call to these functions happens. Such functions include `swap`, `exchangeWrappedTokensForCollectibles`, `closeSuccesfulDrop`, etc.

### Recommendations

Emitting events for important functions is important and allows for better indexing.