HoneyJar Security Review

Pashov Audit Group

Conducted by: SpicyMeatball, pontifex, Dan Ogurtsov, Shaka

June 5th 2024 - June 10th 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 **0xHoneyJar/honeyjar-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About HoneyJar

The game is a smart contract where players can mint honey jars using either ERC20 tokens or ETH, with special fermented jars awarded randomly as milestones are reached. Players can participate in a Slumber Party, and once certain checkpoints are hit, random numbers determine which honey jars become special fermented jars, providing an additional layer of excitement and reward.

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 - fabf0c542528fdb3b2026e961aa7494ad9221248

fixes review commit hash - 0b3b94e40f41cab26094925f4cab795e97b440e4

Scope

The following smart contracts were in scope of the audit:

- `Gatekeeper`
- `HoneyJarONFT`
- `GameRegistryConsumer`
- `Den`
- `Constants`
- `MultisigOwnable`
- `HoneyJar`
- `CrossChainTHJ`
- `Beekeeper`
- `HoneyJarPortal`
- `GameRegistry`
- `TokenMinter`
- `BearPouch`

7. Executive Summary

Over the course of the security review, SpicyMeatball, pontifex, Dan Ogurtsov, Shaka engaged with HoneyJar to review HoneyJar. In this period of time a total of **21** issues were uncovered.

Protocol Summary

Protocol Name	HoneyJar
Repository	https://github.com/0xHoneyJar/honeyjar-contracts
Date	June 5th 2024 - June 10th 2024
Protocol Type	NFT cross-chain game

Findings Count

Severity	**Amount**
Critical	1
High	2
Medium	5
Low	13
Total Findings	**21**

Summary of Findings

ID	Title	Severity	Status
[C-01]	The communication channel for HoneyJarONFT can be blocked	Critical	Resolved
[H-01]	Funds distribution can fail due to the rounding up	High	Resolved
[H-02]	The limit for jars minted can be exceeded	High	Resolved
[M-01]	Edge case in the next non fermented jar search	Medium	Resolved
[M-02]	Slippage protection for wakeSleepers()	Medium	Acknowledged
[M-03]	On-chain randomness can be exploited	Medium	Acknowledged
[M-04]	The user can temporarily block the HoneyJar contract from receiving LZ messages	Medium	Acknowledged
[M-05]	First user to claim a sleeper of type NATIVE_TYPE gets all rewards	Medium	Acknowledged
[L-01]	Incorrect gas estimation in the HoneyJarPortal contract	Low	Acknowledged
[L-02]	No setter for the isSource flag	Low	Acknowledged
[L-03]	Use two step ownership transfer	Low	Acknowledged
[L-04]	GAME_ADMIN can force checkpoint finishing	Low	Acknowledged
[L-05]	It is not ensured that sleepers have been transferred to the Den	Low	Acknowledged

[L-06]	Den does not use safe methods for ERC20 tokens	Low	Acknowledged
[L-07]	LayerZero fees cannot be paid in ZRO token	Low	Acknowledged
[L-08]	Fermenting more than 256 jars eliminates randomness	Low	Acknowledged
[L-09]	The fermenting of jars can cause gas exhaustion	Low	Acknowledged
[L-10]	Reaching multiple checkpoints in a single mint can delay fermentation	Low	Acknowledged
[L-11]	EOA owner can change LZ settings in the HoneyJar.sol	Low	Acknowledged
[L-12]	Reverts on ETH distribution	Low	Acknowledged
[L-13]	mintTokenId() to block batchMint()	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] The communication channel for `HoneyJarONFT` can be blocked

Severity

Impact: High

Likelihood: High

Description

`HoneyJarONFT` inherits from LayerZero's `NonblockingLzApp` to manage the cross-chain NFT transfers.

Let's break down how the flow of receiving a cross-chain message works:

1. The entry point is the `lzReceive` function of the `LzApp` contract, inherited by `NonblockingLzApp`, that executes the internal function `_blockingLzReceive`.

2. `_blockingLzReceive` (overridden by `NonblockingLzApp`) executes `nonblockingLzReceive` through a low-level call. If the call fails, the message is stored to be processed later.

```
File: NonblockingLzApp.sol

    function _blockingLzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) internal virtual override {
        (bool success, bytes memory reason) = address(this).excessivelySafeCall(
            gasleft(),
            150,
            abi.encodeWithSelector(
              this.nonblockingLzReceive.selector,
              _srcChainId,
              _srcAddress,
              _nonce,
              _payload
            )
        );
        if (!success) {
            _storeFailedMessage
              (_srcChainId, _srcAddress, _nonce, _payload, reason);
        }
    }
```

3. `nonblockingLzReceive` checks that the message sender is the own contract and executes the internal function `_nonblockingLzReceive`, which is overridden by `HoneyJarONFT`.

4. `_nonblockingLzReceive` calls `_creditTill`, that iterates over all the tokens, crediting them to the receiver. In each iteration, it checks if there is enough gas to continue processing the tokens. If not, it returns the index of the next token to process and back to `_blockingLzReceive`, the message is stored to be processed later.

In the implementation of this last step, there is a problem, as after `_creditTill` has been executed there is an additional loop that iterates over the tokens to build an array of tokenIds that will be emitted in an event. Depending on the number of tokens, this loop can consume all remaining gas, causing the transaction to revert.

This error will not be caught by the `_blockingLzReceive` function, effectively blocking the message channel, and not allowing the contract to receive any more messages. It is also important to note that when the message is sent from the source chain, the NFTs are burned, so the user will lose the NFTs.

File: HoneyJarONFT.sol

```
    function _nonblockingLzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64, /*_nonce*/
        bytes memory _payload
    ) internal override {
        // decode and load the toAddress
        (address toAddress, TokenPayload[] memory tokens) = _decodeSendNFT
          (_payload);

        uint256 nextIndex = _creditTill(_srcChainId, toAddress, 0, tokens);
        if (nextIndex < tokens.length) {
            // not enough gas to complete transfers, store to be cleared in
            // another tx
            bytes32 hashedPayload = keccak256(_payload);
            storedCredits[hashedPayload] = StoredCredit
              (_srcChainId, toAddress, nextIndex, true);
            emit CreditStored(hashedPayload, _payload);
        }

        uint256[] memory tokenIds = new uint256[](tokens.length);
@>      for (uint256 i = 0; i < tokens.length; i++) {
            tokenIds[i] = tokens[i].tokenId;
        }

        emit ReceiveFromChain(_srcChainId, _srcAddress, toAddress, tokenIds);
    }

(...)

    function _creditTill(
      uint16_srcChainId,
      address_toAddress,
      uint256_startIndex,
      TokenPayload[]memorytokens
    )
        internal
        returns (uint256)
    {
        uint256 i = _startIndex;
        while (i < tokens.length) {
            // if not enough gas to process, store this index for the next loop
            if (gasleft() < minGasToTransferAndStore) break;

            _creditTo(
              _srcChainId,
              _toAddress,
              tokens[i].tokenId,
              tokens[i].isFermented
            );
            i++;
        }

        // indicates the next index to send of tokenIds,
        // if i == tokenIds.length, we are finished
        return i;
    }
```

Recommendations

```
function _nonblockingLzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64, /*_nonce*/
        bytes memory _payload
    ) internal override {
        // decode and load the toAddress
        (address toAddress, TokenPayload[] memory tokens) = _decodeSendNFT
          (_payload);

-       uint256 nextIndex = _creditTill(_srcChainId, toAddress, 0, tokens);
+       (uint256 nextIndex, uint256[] memory tokenIds) = _creditTill
+ (_srcChainId, toAddress, 0, tokens);
        if (nextIndex < tokens.length) {
            // not enough gas to complete transfers, store to be cleared in
            // another tx
            bytes32 hashedPayload = keccak256(_payload);
            storedCredits[hashedPayload] = StoredCredit
              (_srcChainId, toAddress, nextIndex, true);
            emit CreditStored(hashedPayload, _payload);
        }

-       uint256[] memory tokenIds = new uint256[](tokens.length);
-       for (uint256 i = 0; i < tokens.length; i++) {
-           tokenIds[i] = tokens[i].tokenId;
-       }
-
        emit ReceiveFromChain(_srcChainId, _srcAddress, toAddress, tokenIds);
    }

(...)

    function _creditTill(
      uint16_srcChainId,
      address_toAddress,
      uint256_startIndex,
      TokenPayload[]memorytokens
    )
        internal
-       returns (uint256)
+       returns (uint256, uint256[] memory)
    {
        uint256 i = _startIndex;
+       uint256[] memory tokenIds = new uint256[](tokens.length);
        while (i < tokens.length) {
            // if not enough gas to process, store this index for next loop
            if (gasleft() < minGasToTransferAndStore) break;

            _creditTo(
              _srcChainId,
              _toAddress,
              tokens[i].tokenId,
              tokens[i].isFermented
            );
+           tokenIds[i] = tokens[i].tokenId;
            i++;
        }

        // indicates the next index to send of tokenIds,
        // if i == tokenIds.length, we are finished
-       return i;
+       return (i, tokenIds);
    }
```

Note that with this change `minGasToTransferAndStore` should take into account the additional gas needed for setting each individual element in the `tokenIds` array.

8.2. High Findings

[H-01] Funds distribution can fail due to the rounding up

Severity

Impact: Medium

Likelihood: High

Description

When a NFT is sold, `Beekeeper` calls `BearPouch.distribute` to distribute the received funds to the configured recipients according to their share value. The calculation of the amount to be transferred to each recipient uses `FixedPointMathLib.mulWadUp` which rounds up the result to the nearest whole number.

```
File: BearPouch.sol

61              if (xferERC20) {
62                  paymentToken.safeTransferFrom(

64                  );
65              }
66
67              // xfer the ETH
68              if (xferETH) {
69                  SafeTransferLib.safeTransferETH(
70  @>                  distributions[i].recipient, (msg.value).mulWadUp
  (distributions[i].share)
71                  );
72              }
```

Rounding up the amounts to be transferred can lead to a situation where the total amount transferred is greater than the total amount received, causing the transaction to revert and thus, preventing the NFT sale from being completed.

Proof of concept

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "solady/src/utils/FixedPointMathLib.sol";
import "src/GameRegistry.sol";
import "src/BearPouch.sol";
import "src/Constants.sol";

contract PoC is Test {
    GameRegistry gameRegistry;
    BearPouch bearPouch;

    function setUp() public {
        gameRegistry = new GameRegistry();
        gameRegistry.grantRole(Constants.GAME_INSTANCE, address(this));


        uint256 shareAlice = FixedPointMathLib.WAD / 3;
        uint256 shareBob = FixedPointMathLib.WAD - shareAlice;
        distributions[0] = IBearPouch.DistributionConfig({recipient: makeAddr
          ("Alice"), share: shareAlice});
        distributions[1] = IBearPouch.DistributionConfig({recipient: makeAddr
          ("Bob"), share: shareBob});
        bearPouch = new BearPouch(address(gameRegistry), address
          (0), distributions);
    }

    function test_distributeReverts() public {
        vm.expectRevert(SafeTransferLib.ETHTransferFailed.selector);
        bearPouch.distribute{value: 0.099 ether}(0);
        // Alice's share = 0.033 ether
        // Bob's share = 0.066 ether + 1 wei
        // Total = 0.099 ether + 1 wei
    }
}
```

Recommendations

Use `mulWad` instead of `mulWadUp`.

[H-02] The limit for jars minted can be exceeded

Severity

Impact: High

Likelihood: High

Description

All `Beekeeper` external functions that mint jars perform some checks implemented in the internal `_canMintHoneyJar` function. Among other things, this function checks that the sum of the minted jars and the new amount to be minted does not exceed the limit defined by the checkpoints.

```
File: Beekeeper.sol

168         if
  (progress.mintedJars.length + amount_ > progress.checkpoints[progress.checkpoints.le
169             revert MekingTooManyHoneyJars();
170         }
```

If all the checks pass, `_mintHoneyJar` is called to mint the jars. In this function is important to note that `progress.mintedJars` is updated only after `honeyJar.batchMint` is called.

```
File: Beekeeper.sol

251     function _mintHoneyJar(address to, uint256 amount_) internal returns
  (uint256) {
252         uint256 tokenId = honeyJar.nextTokenId();
253         honeyJar.batchMint(to, amount_);
254
255         for (uint256 i = 0; i < amount_; ++i) {
256             progress.mintedJars.push(tokenId);
257             ++tokenId;
258         }
(...)
```

`HoneyJar` uses the internal `_safeMint` function to mint the jars. This function calls the `onERC721Received` function of the receiver contract if it exists. This can be used to re-enter the `Beekeeper` contract and mint more jars before the `progress.mintedJars` array is updated, allowing the mint limit to be exceeded. Another outcome of this attack is that the token ids stored in the `progress.mintedJars` array will be incorrect.

Proof of concept

Add the following contract to the file `Beekeeper.t.sol`.

```
contract Attacker {
    Beekeeper public beekeeper;
    uint256 public mintEthPrice;
    uint256 public reentranceAmount;

    constructor(Beekeeper _beekeeper, uint256 _mintEthPrice) {
        beekeeper = _beekeeper;
        mintEthPrice = _mintEthPrice;
    }

    function mek(uint256 amount, uint256 _reentranceAmount) public payable {
        reentranceAmount = _reentranceAmount;
        beekeeper.mekHoneyJarWithETH{value: amount * mintEthPrice}(amount);
    }

    function onERC721Received(
      address,
      address,
      uint256,
      bytesmemory
    ) public virtual returns (bytes4
        uint256 amount = reentranceAmount;
        if (amount > 0) {
            reentranceAmount = 0;
            beekeeper.mekHoneyJarWithETH{value: amount * mintEthPrice}(amount);
        }
        return this.onERC721Received.selector;
    }
}
```

Add also the following function inside `BeekeeperTest` contract in the same file and run `forge test --mt testMintOverLimit -vv`.

```
function testMintOverLimit() public {
    // Setup
    uint256 MAX_MEK_JARS = 4;
    uint256[] memory checkpoints = new uint256[](2);
    checkpoints[0] = 2;
    checkpoints[1] = MAX_MEK_JARS;
    beekeeper.setCheckpoints(0, checkpoints);
    gameRegistry.startGame(address(beekeeper));
    beekeeper.startGame(block.chainid, numSleepers);

    // Attack
    uint256 amountOverLimit = 2;
    Attacker attacker = new Attacker(beekeeper, MINT_PRICE_ETH);
    attacker.mek{value: (MAX_MEK_JARS + amountOverLimit) * MINT_PRICE_ETH}
      (MAX_MEK_JARS, amountOverLimit);

    assertEq(honeyJar.balanceOf(address
      (attacker)), MAX_MEK_JARS + amountOverLimit);
    Beekeeper.Apiculture memory progress = beekeeper.getProgress();
    for (uint256 i = 0; i < progress.mintedJars.length; i++) {
        console2.log("mintedJars[%s] = %s", i, progress.mintedJars[i]);
    }
}
```

Console output:

```
Logs:
  mintedJars[0] = 1
  mintedJars[1] = 2
  mintedJars[2] = 0
  mintedJars[3] = 1
  mintedJars[4] = 2
  mintedJars[5] = 3
```

Recommendations

Add a reentrancy guard to all external functions that mint jars, that is, `earlyMekHoneyJarWithERC20`, `earlyMekHoneyJarWithETH`, `mekHoneyJarWithETH`, `mekHoneyJarWithERC20`, `claim` and `adminMint`.

8.3. Medium Findings

[M-01] Edge case in the next non fermented jar search

Severity

Impact: Medium

Likelihood: Medium

Description

If the randomizer selects a jar that is already fermented, a collission occurs.

```
function _setFermentedJars(uint256[] memory randomNumbers) internal {
        ---SNIP---
        for (uint256 i = 0; i < numFermentedJars; i++) {
            fermentedIndex = randomNumbers[i] % numHoneyJars;
            uint256 fermentedJarId = honeyJarIds[fermentedIndex];
            if (honeyJar.isFermented(fermentedJarId)) {
                // Special case in the event of a collision -- not optimized for
                // gas
>>              fermentedJarId = _findNextJar(fermentedJarId);
            }
```

In this case the `Beekeeper` contract attempts to find another jar that wasn't fermented:

```
function _findNextJar(uint256 jarId) internal returns (uint256) {
        uint256 supplyCap = honeyJar.supplyCap();

>>      for (uint256 i = jarId + 1; i < supplyCap; i++) {
            if (honeyJar.isFermented(i)) continue;
            return i;
        }
        return type(uint256).max; // if you can't find any.
    }
```

It searches from the currently selected id to the supply cap. If no jars are found matching the condition, type (uint256).max is returned. This algorithm is not quite correct since the loop doesn't start from the first tokenId.

Consider the following scenario: we have a supply cap 10 and 10 tokens minted, tokenId = 10 is already fermented. The randomizer again selects tokenId = 10 and `_findNextJar` is called, since 10-th is the last token it will return `type(uint256).max`, even though tokens with IDs 1-9 are not fermented.

Now let's examine the same scenario, but with `checkpoints[progress.checkpoints.length - 1] < supplyCap`. Calling `_findNextJar` will ferment a non-existent jar. Since the game has ended and new jars can't be minted, one sleeper reward will remain unobtained.

Recommendations

```
+       for (uint256 i = 0; i < supplyCap; i++) {
```

[M-02] Slippage protection for `wakeSleepers()`

Severity

Impact: Medium

Likelihood: Medium

Description

All Jars are equal in terms of value and can be used to receive tokens locked in Sleepers. Users cannot choose exact Sleepers, the next available Sleeper is always used. It is considered a game mechanic. The problem is that User Alice can sign a transaction expecting to receive a Sleeper 2, but during the transaction execution, someone frontrun the user and now Alice spends a Jar to receive Sleeper 3.

Recommendations

Allows users to indicate the Sleeper index they wanted when signing the transaction.

[M-03] On-chain randomness can be exploited

Severity

Impact: Medium

Likelihood: Medium

Description

Once a certain number of HoneyJar NFTs are minted, some amount of tokens are selected to be turned into fermented jars. These fermented jars can later be used to receive rewards from sleepers in the `Den` contract.

```
function _mintHoneyJar(address to, uint256 amount_) internal returns
      (uint256) {
        uint256 tokenId = honeyJar.nextTokenId();
        honeyJar.batchMint(to, amount_);

        for (uint256 i = 0; i < amount_; ++i) {
            progress.mintedJars.push(tokenId);
            ++tokenId;
        }

        // Find the special honeyJar when a checkpoint is passed.
        uint256 numMinted = progress.mintedJars.length;
        if (numMinted >= progress.checkpoints[progress.checkpointIndex]) {
>>          _fermentJars();
        }

        return tokenId - 1; // returns the lastID created
    }
```

The ID of the token that'll be fermented is calculated using a following algorithm (simplified algorithm for one jar and no collisions):

- get a random number `randomBytes = uint256(keccak256(abi.encodePacked(block.timestamp, block.difficulty, msg.sender)))`
- get the index of the fermented jar `id = randomBytes % num of minted jars`
- get the id from the minted jars array

Fermentation takes place in the same transaction as minting. This allows a savvy user to create a contract that will receive minted jars and revert transactions until the fermented jar is received.

Example of the minter contract:

```
contract Reroll {
    Beekeeper beekeeper;
    uint256 offset;

    constructor(Beekeeper _beekeeper) {
        beekeeper = _beekeeper;
    }

    function predict() internal returns(uint256 id){
        uint256 rand = uint256(keccak256(abi.encodePacked(
            block.timestamp,
            block.difficulty,
            address(this)
        )));
        Beekeeper.Apiculture memory progress = beekeeper.getProgress();
        id = rand % (progress.mintedJars.length + offset);
    }

    function setOffset(uint256 _offset) external {
        offset = _offset;
    }

    function onERC721Received
      (address,address,uint256 id,bytes calldata) external returns(bytes4) {
        uint256 gibmedat = predict();
        if(id != gibmedat) revert("YIKES");
        return this.onERC721Received.selector;
    }
}
```

Coded POC for the `Beekeeper.t.sol` :

```
function testReroll() public {
        address alice = address(0xa11cE);
        Reroll reroll = new Reroll(beekeeper);

        vm.deal(alice, 100 ether);
        vm.deal(address(reroll), 100 ether);

        _puffPuffPassOut();
        uint256 newSleepers = 10;
        uint256 mintAmount = 100;
        uint256[] memory checkpoints = new uint256[](3);
        checkpoints[0] = 20;
        checkpoints[1] = 40;
        checkpoints[2] = mintAmount;

        // Set the game
        gameRegistry.stopGame(address(beekeeper));
        beekeeper.setCheckpoints(0, checkpoints);
        beekeeper.setNumSleepers(newSleepers);
        gameRegistry.startGame(address(beekeeper));

        // First minter
        vm.prank(alice);
        beekeeper.mekHoneyJarWithETH{value: MINT_PRICE_ETH * 19}(19);
        // Second minter will get the fermented jar even though Alice has 19
        // jars
        vm.warp(1);
        vm.startPrank(address(reroll));
        reroll.setOffset(1);
        vm.expectRevert("YIKES");
        beekeeper.mekHoneyJarWithETH{value: MINT_PRICE_ETH}(1);
        // Try again
        vm.warp(2);
        uint256 minted = beekeeper.mekHoneyJarWithETH{value: MINT_PRICE_ETH}(1);
        assertEq(honeyJar.isFermented(minted), true);
    }
```

Recommendations

Consider generating random numbers off-chain, either using services like Chainlink VRF or using your own internal server.

[M-04] The user can temporarily block the `HoneyJar` contract from receiving LZ messages

Severity

Impact: Medium

Likelihood: Medium

Description

The `HoneyJar` token uses `ONFT721.sol` and `NonblockingLzApp.sol` contracts for cross-chain communication via the Layer-Zero protocol. When it receives the message from the endpoint `NonblockingLzApp` overrides the `_blockingLzReceive` function:

```
// LzApp.sol
    function lzReceive(
        uint16 _srcChainId,
        bytes calldata _srcAddress,
        uint64 _nonce,
        bytes calldata _payload
    ) public virtual override {
        // lzReceive must be called by the endpoint for security
        require(_msgSender() == address
          (lzEndpoint), "LzApp: invalid endpoint caller");
        ---SNIP---
>>      _blockingLzReceive(_srcChainId, _srcAddress, _nonce, _payload);
    }

// NonblockingLzApp.sol
    function _blockingLzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) internal virtual override {
>>      (bool success, bytes memory reason) = address(this).excessivelySafeCall(
            gasleft(),
            150,
            abi.encodeWithSelector(
              this.nonblockingLzReceive.selector,
              _srcChainId,
              _srcAddress,
              _nonce,
              _payload
            )
        );
        if (!success) {
>>          _storeFailedMessage
  (_srcChainId, _srcAddress, _nonce, _payload, reason);
        }
    }
```

There it calls the `nonBlockingReceive` function, which eventually mints a token with a specified ID to the receiver. If `nonBlockingReceive` reverts the payload is stored to be retried later:

```
function _creditTo(
      uint16,
      address _toAddress,
      uint256 _tokenId,
      bool _isFermented
    ) internal override {
        require(
            !_exists(_tokenId) || (_exists(_tokenId) && ERC721.ownerOf
              (_tokenId) == address(this)), "invalid token ID"
        );

        // Token shouldn't exist
>>      _safeMint(_toAddress, _tokenId);
        if (_isFermented) fermented.set(_tokenId);
    }
```

Since the `_safeMint` function initiates a callback to the receiving contract, a problem may arise where all the transaction gas is spent inside that callback. If this happens, there will be no gas left to store the payload and then we will end up in the catch block in `Endpoint.sol`:

```
function receivePayload(
      uint16 _srcChainId,
      bytes calldata _srcAddress,
      address _dstAddress,
      uint64 _nonce,
      uint _gasLimit,
      bytes calldata _payload
    ) external override receiveNonReentrant {
        ---SNIP---
        // block if any message blocking
        StoredPayload storage sp = storedPayload[_srcChainId][_srcAddress];
>>      require(sp.payloadHash == bytes32(0), "LayerZero: in message blocking");

        try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}
          (_srcChainId, _srcAddress, _nonce, _payload) {
            // success, do nothing, end of the message delivery
        } catch (bytes memory reason) {
            // revert nonce if any uncaught errors/exceptions if the ua chooses
            // the blocking mode
>>          storedPayload[_srcChainId][_srcAddress] = StoredPayload(uint64
  (_payload.length), _dstAddress, keccak256(_payload));
            emit PayloadStored
              (_srcChainId, _srcAddress, _dstAddress, _nonce, _payload, reason);
        }
```

`HoneyJar` will then be unable to receive messages from the source chain until the owner calls `forceResumeReceive` and removes the payload.

Please take a look at the detailed explanation of this vulnerability in this article: https://www.trust-security.xyz/post/learning-by-breaking-a-layerzero-case-study-part-3

Recommendations

Consider leaving at least 30k gas to successfully store the payload in the `catch` block:

```
function _blockingLzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) internal virtual override {
      (bool success, bytes memory reason) = address(this).excessivelySafeCall(
+           gasleft() - 30000, // in theory gasLeft
+ () is always greater than 30000 because of checks in the send
            150,
            abi.encodeWithSelector(
              this.nonblockingLzReceive.selector,
              _srcChainId,
              _srcAddress,
              _nonce,
              _payload
            )
        );
        if (!success) {
          _storeFailedMessage
            (_srcChainId, _srcAddress, _nonce, _payload, reason);
        }
    }
```

[M-05] First user to claim a sleeper of type `NATIVE_TYPE` gets all rewards

Severity

Impact: High

Likelihood: Low

Description

Sleepers held by the `Den` contract can be of different types. The type `NATIVE_TYPE` rewards the holder of a fermented jar with the native token of the chain (e.g. ether).

There can be multiple sleepers of the same type in a party, each of them with a different `amount` of rewards. However, when a sleeper of the type `NATIVE_TYPE` is transferred out, the amount transferred is the total balance of the contract, not the amount of the specific sleeper.

This means that the first user that claims a sleeper of the type `NATIVE_TYPE` will receive the rewards of all the sleepers of that type and subsequent users will receive nothing.

Recommendations

```
} else if (NATIVE_TYPE == tokenType) {
-           SafeTransferLib.safeTransferAllETH
- (to); // winner gets all eth in the contract
+           SafeTransferLib.safeTransferETH(to, sleeper_.amount);
        } else {
```

8.4. Low Findings

[L-01] Incorrect gas estimation in the `HoneyJarPortal` contract

The `sendStartGame` function includes extra gas cost of `1000 * numSleepers` in the gas estimation before sending the message over the Layer Zero protocol.

```
function sendStartGame(
      addresspayablerefundAddress_,
      uint256destChainId_,
      uint256numSleepers_
    )
        external
        payable
        override
        onlyRole(Constants.GAME_INSTANCE)
    {
        uint16 lzDestId = lzChainId[destChainId_];
        if (lzDestId == 0) revert LzMappingMissing(destChainId_);

        bytes memory adapterParams = msgAdapterParams[MessageTypes.START_GAME];

        // Will check adapterParams against minDstGas
        _checkGasLimit(
            lzDestId,
            uint16(MessageTypes.START_GAME),
            adapterParams,
>>          1000 * numSleepers_ // Padding for each NFT being stored
        );
```

However, on the target chain, `Beekeeper.startGame` doesn't use much gas and does not include `numSleepers` in the calculations.

```
function startGame(
      uint256srcChainId_,
      uint256numSleepers_
    ) external override onlyRole(Constants.PORTAL
        if (progress.checkpoints.length == 0) revert ProgressNotInitialized();
        if (progress.checkpoints.length > numSleepers_) revert InvalidInput
          ("startGame::checkpoints");
        party.assetChainId = srcChainId_;
        party.mintChainId = getChainId
        //(); // On the destination chain you MUST be able to mint.
        party.numSleepers = numSleepers_;
        canMakeJars = true;
        emit SlumberPartyStarted();
        return;
    }
```

As a result, the caller will have to pay more gas than is needed to execute `startGame` on the target chain.

[L-02] No setter for the `isSource` flag

The `isSource` parameter is used to check and prevent the Beekeeper contract from being instantiated on the same chain as the Den contract.

```
function setBeekeeper(address beekeeperAddress_) external onlyRole
      (Constants.GAME_ADMIN) {
>>      if (isSource) {
            revert SourceChain();
        }
        beekeeper = IBeekeeper(beekeeperAddress_);

        emit BeekeeperSet(beekeeperAddress_);
    }
```

Unfortunately, there is no setter for the `isSource`, so it can only be false. Consider specifying `isSource` as an immutable in the constructor.

[L-03] Use two step ownership transfer

Two step ownership transfer prevents the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner's permissions actively accept via a contract call of its own. MultisigOwnable.sol one step ownership transfer:

```
function transferRealOwnership(address newRealOwner) public onlyRealOwner {
>>      realOwner = newRealOwner;
    }
```

Use two step ownership transfer instead:

```
+   address private _pendingOwner;
<...>
    function transferRealOwnership(address newRealOwner) public onlyRealOwner {
-       realOwner = newRealOwner;
+       _pendingOwner = newRealOwner;
    }

+   function acceptOwnership() public {
+       address sender = msg.sender;
+       require (pendingOwner
+ () == sender, "MultisigOwnable: caller is not the new real owner");
+       realOwner = sender;
+       delete _pendingOwner;
+   }
```

[L-04] `GAME_ADMIN` can force checkpoint finishing

The `Beekeeper.setAdminMint` lets the `GAME_ADMIN` role increase the `adminMintAmount` variable even if the game is enabled. As mentioned in the protocol documentation about the Hibernation Games "If Honeycomb never sells out, then the OG Bong Bear will forever slumber in the Honey Jar Cave. The contracts, and thus the cave, are immutable" (https://0xhoneyjar.mirror.xyz/soVN56Jla_Y9x2USB9UO2Pw3T0ALiHwAbI0oxC5AA0M). But this invariant can be broken by the `GAME_ADMIN` role.

```
function setAdminMint(uint256 adminMintAmount_) external onlyRole
      (Constants.GAME_ADMIN) {
        adminMintAmount = adminMintAmount_;

        emit AdminMintAmount(adminMintAmount_);
    }
```

Consider checking the game status and revert if the game is enabled.

[L-05] It is not ensured that sleepers have been transferred to the Den

The setup of a `Den` requires an array of sleepers to be added to a party. This operation is performed by the admin with a call to `setSlumberParty` or `addToParty` functions.

Both functions receive a boolean parameter that is used to determine if the sleeper is to be transferred by the caller or not. When this value is set to `false` it is expected that the sleeper has already been transferred to the `Den` contract or it will be transferred later.

However, there is no check in the `startGame` function that validates if the sleepers have been transferred to the `Den` contract. So, a game could be started with sleepers that have not been transferred to the `Den` contract, which can provoke the `wakeSleeper` function to fail.

[L-06] `Den` does not use `safe` methods for `ERC20` tokens

Some ERC20 tokens can fail silently on transfer, which can lead to loss of funds. While the `SafeERC20` library is assigned to the `IERC20` interface, the contract does not use the `safeTransfer` and `safeTransferFrom` functions to prevent this.

```
File: Den.sol

196         if (from == address(this)) {
197             // transferFrom does an allowance check which is not needed for
// the contract owned assets.
198             IERC20(sleeper_.tokenAddress).transfer(to, sleeper_.amount);
199         } else {
200             IERC20(sleeper_.tokenAddress).transferFrom
  (from, to, sleeper_.amount);
```

[L-07] LayerZero fees cannot be paid in ZRO token

Hardcoding the `_zroPaymentAddress` field to address(0) disallows the protocol from using ZRO token as a fee payment option. Consider passing the _zroPaymentAddress field as an input parameter to allow flexibility on the fee payments.

This is explicitly stated in the LayerZero integration checklist.

> Do not hardcode address zero (address(0)) as zroPaymentAddress when estimating fees and sending messages. Pass it as a parameter

instead.

```
File: HoneyJarPortal.sol

155         _lzSend(lzDestId, payload, refundAddress_, address
  (0x0), adapterParams, msg.value);
```

[L-08] Fermenting more than 256 jars eliminates randomness

In `Beekeeper._getRandomWords`, when `numWords_` is greater than 256, all seeds will be zero, eliminating any kind of randomness and benefiting the first jars minted.

```
File: Beekeeper.sol

        for (uint256 i = 0; i < numWords_; i++) {
            uint256 mask = (1 <<
            //(256 / numWords_)) - 1; // Generate mask for splitting the bytes
            seeds[i] = randomBytes >> (256 - (i + 1) *
            //(256 / numWords_)) & mask; // Extract and apply mask
        }
```

Consider limiting the number jars to be fermented in a single transaction.

[L-09] The fermenting of jars can cause gas exhaustion

`Beekeeper._setFermentedJars` iterates over `numFermentedJars` to find the next jar to ferment. If the chosen `fermentedJarId` corresponds to a jar that has already been fermented, a new iteration happens in `_findNextJar` until a jar that has not been fermented is found.

Depending on the number of jars to be fermented and the number of jars that have already been fermented, this can lead to a high number of iterations, that can cause eventual gas exhaustion, preventing the minting of new jars and the fermentation of minted jars.

Consider limiting the number of jars to be fermented in a single transaction and implementing a more efficient way to find the next jar to ferment. This

could be done for example by keeping track of all jars that have been minted and not yet fermented in a list.

[L-10] Reaching multiple checkpoints in a single mint can delay fermentation

When tokens are minted in `Beekeeper` there is a possibility that more than one checkpoint is reached. This will effectively skip checkpoints and delay the fermentation of jars until the last checkpoint is reached.

This is acknowledged in the natspec of the `_mintHoneyJar` function:

```
/// @dev if the amount_ is > than multiple checkpoints, accounting WILL mess up.
```

There is also a mention of an intended remedy for this issue in the natspec of the `Apiculture` struct in the `IBeekeeper` interface:

```
/// @dev the gap between checkpoints MUST be big enough so that a user can't
// mint through multiple checkpoints.
```

However, this solution might be hard to implement, as creating a gap big enough would require a very large supply and would potentially cause new checkpoints never to be reached.

A potential solution could be to add a check in the `_mintHoneyJar` function to ensure that the amount of tokens minted does not exceed the number of tokens that can be minted in a single checkpoint. However, this would also require the ability for users to make partial claims so that they can claim all their tokens without exceeding the checkpoint limit.

[L-11] EOA owner can change LZ settings in the `HoneyJar.sol`

Since the `HoneyJar.sol` inherits the `multisigOwnable.sol`, it has two owners: `realOwner` is the multisig that controls the token's admin functions and `owner` an EOA that is used for OpenSea collections management.

In the snippets below it can be observed that `realOwner` is used for admin functions:

```
function setBaseURI(string calldata baseURI_) external onlyRealOwner {

    function setGenerated(bool generated_) external onlyRealOwner {
```

However, if we check the inherited contracts `HoneyJarONFT` and `LzApp` we can see that important admin functions are controlled by the lower `owner`, which is considered less secure since it's an EOA:

HoneyJarONFT.sol

```
function setMinGasToTransferAndStore
      (uint256 _minGasToTransferAndStore) external onlyOwner {

    // ensures enough gas in adapter params to handle batch transfer gas amounts
    // on the dst
    function setDstChainIdToTransferGas(
      uint16_dstChainId,
      uint256_dstChainIdToTransferGas
    ) external onlyOwner {

    // limit on src the amount of tokens to batch send
    function setDstChainIdToBatchLimit
      (uint16 _dstChainId, uint256 _dstChainIdToBatchLimit) external onlyOwner {
```

LzApp.sol

```
function setSendVersion(uint16 _version) external override onlyOwner {

    function setReceiveVersion(uint16 _version) external override onlyOwner {

    function forceResumeReceive(
      uint16_srcChainId,
      bytescalldata_srcAddress
    ) external override onlyOwner {

    // _path = abi.encodePacked(remoteAddress, localAddress)
    // this function set the trusted path for the cross-chain communication
    function setTrustedRemote
      (uint16 _remoteChainId, bytes calldata _path) external onlyOwner {

    function setTrustedRemoteAddress(
      uint16_remoteChainId,
      bytescalldata_remoteAddress
    ) external onlyOwner {

    function setPrecrime(address _precrime) external onlyOwner {

    function setMinDstGas(
        uint16 _dstChainId,
        uint16 _packetType,
        uint _minGas
    ) external onlyOwner {

    // if the size is 0, it means default size limit
    function setPayloadSizeLimit
      (uint16 _dstChainId, uint _size) external onlyOwner {
```

Consider overriding the above functions so that only `realOwner` can call them.

[L-12] Reverts on ETH distribution

BearPounch.distribute() distributes funds based on shares. ETH distributions are also allowed via `SafeTransferLib.safeTransferETH()`. This function still can revert if the recipient reverts.

```
// The regular variants:
    // - Forwards all remaining gas to the target.
    // - Reverts if the target reverts.
    // - Reverts if the current contract has insufficient balance.
```

The recipient can use reverts to blocklist jar some buyers.

Consider using a "claim" approach when no ETH is transferred, and the contracts increase the internal balance allowed to claim by the recipient.

[L-13] `mintTokenId()` to block `batchMint()`

`HoneyJar.mintTokenId()` allows minting the exact `tokenId` provided. This is a risky approach, because `Beekeper` uses `batchMint()` that mint N tokens from the current amount. If a minted token from `HoneyJar.mintTokenId()` is met during this batch minting, the transaction will revert with no way to mint new tokens in `Beekeeper`.

Consider implementing rules to safely mint `tokenId`s or consider removing the function.