Bear Cave Security Review

Pashov Audit Group

Conducted by: pashov

July 6th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his 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 I try to find as many vulnerabilities as possible. I 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 **Bear Cave** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Bear Cave

Bear Cave is a protocol that can host multiple NFT games on-chain. The protocol has integrated LayerZero technology to make the games cross-chain - while the games' NFTs will be minted on Ethereum mainnet, the actual game will launch on Arbitrum Mainnet. Honey Jar NFTs are minted either by free claim or with either ETH or the payment token (expected to be $OHM). When a given number of Honey Jar NFTs have been minted, a game winner is chosen using Chainlink's VRF technology and he can claim a special ERC721/ERC1155 token as a prize.

Here is how the flow of action works:

1. (Ethereum Mainnet) A game admin calls `addBundle` to configure the winning prize ERC721/ERC1155 tokens for a game
2. (Ethereum Mainnet) A game admin calls `puffPuffPassOut` to actually transfer the tokens into the `HibernationDen` contract and also sends a cross-chain message to Arbitrum Mainnet to start the game there (calling `startGame`)
3. (Arbitrum Mainnet) Now that a game is running, users can mint HoneyJar NFTs via `claim`, `claimAll`, `earlyMekHoneyJarWithERC20`, `earlyMekHoneyJarWithEth`, `mekHoneyJarWithERC20` and `mekHoneyJarWithETH`
4. (Arbitrum Mainnet) On each checkpoint for Honey Jar mints, Chainlink's VRF will be called
5. (Arbitrum Mainnet) VRF randomness gets fulfilled, setting the fermented jars (winning NFTs) on the L2
6. (Arbitrum Mainnet) Anyone can manually call `sendFermentedJars` to send a cross-chain message that will reapply the L2 state related to winning NFTs (fermented jars) to Ethereum Mainnet (calling `setCrossChainFermentedJars`)
7. (Arbitrum Mainnet) Owners of winning HoneyJar NFTs must bridge them to Ethereum mainnet
8. (Ethereum mainnet) Owners of the winning HoneyJar NFTs can call `wakeSleeper` and claim a winning prize ERC721/ERC1155 token

Optional functionality is `HibernationDen::addToParty` which allows for adding more winning prize ERC721/ERC1155 tokens for an already running game

Privileged Roles & Actors

- Game admin - can control and configure most of the protocol. Granted to the Bear Cave team's multi-sig
- Game instance - updates `Gatekeeper`'s internal claimed `HoneyJar` tokens accounting and starts gates for a token. Granted to the `HibernationDen` contract
- Portal - can call `startGame` and `setCrossChainFermentedJars` on L2
- Minter - can mint `HoneyJar` tokens. Granted to both the `HoneyBox` and `HoneyJarPortal` contracts
- Burner - can burn any `HoneyJar` token, no matter who owns it. Currently not granted to anyone, it is expected that it will be granted to the `HibernationDen` contract
- Beekeeper - receives a share of the `HoneyJar` NFT mint fees
- Jani - receives a share of the `HoneyJar` NFT mint fees
- Player - can claim free `HoneyJar` tokens based on eligibility or mint `HoneyJar` tokens with ERC20 tokens or ETH
- Chainlink VRF - supplies randomness so that the "special Honey Jar" NFT is chosen.

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* - 406157655fdd94f90b4606773d3eacf1b9c4c725**

***fixes review commit hash* - 90ea448845b743ad7dcd863a28008671530c64d5**

Scope

The following smart contracts were in scope of the audit:

- `HibernationDen`
- `HoneyJarPortal`

7. Executive Summary

Over the course of the security review, pashov engaged with Bear Cave to review Bear Cave. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Bear Cave
Date	July 6th, 2023

Findings Count

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

Summary of Findings

ID	Title	Severity	Status
[C-01]	State is not properly updated on L2, leading to stuck prize NFTs	Critical	Resolved
[C-02]	Anyone can burn any NFT in the HoneyJarPortal	Critical	Resolved
[H-01]	Accepting input after randomness is requested can be exploited	High	Resolved
[M-01]	The fulfillRandomWords method might revert with out of gas error	Medium	Resolved
[M-02]	Contract HibernationDen can receive ETH but it can't be withdrawn	Medium	Resolved
[M-03]	Centralization attack vectors are present in the code	Medium	Resolved
[L-01]	Not enforced checkpoints gap size	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] State is not properly updated on L2, leading to stuck prize NFTs

Severity

Impact: High, as winning prize NFTs won't be claimable

Likelihood: High, as functionality is never working correctly

Description

The `addToParty` method in `HibernationDen` does not send a cross-chain message, even though it pushes sleepers into the `party.sleepoors` array on L1. Since the array won't be updated on L2, now all the calculations and validations that are done based on the `party.sleepoors.length` value will be incorrect. This essentially means that the sleeper NFTs will be stuck as in `wakeSleeper` there is the following check:

```
if (party.numUsed == party.sleepoors.length) revert PartyAlreadyWoke(bundleId_);
```

Which means that even though there are actually let's say 10 sleepers (they were 7, but then 3 were added), when `numUsed` is 7 then no more sleepers would be claimable - the `wakeSleeper` call will revert will `PartyAlreadyWoke`.

Recommendations

Make sure to send a cross-chain message to add the sleepers to the `party.sleeepoors` array on the L2.

[C-02] Anyone can burn any NFT in the `HoneyJarPortal`

Severity

Impact: High, because the NFTs won't be retrievable from the portal anymore

Likelihood: High, because it does not require any preconditions to be exploited

Description

The `_debitFrom` method in `HoneyJarPortal` allows burning of any `HoneyJar` NFT, given its ID. This method is freely callable through `ONFT721Core`'s `sendFrom` method, which calls `_send` which calls the `_debitFrom` method without an access control check. This results in the ability for anyone to burn any HoneyJar NFT, no matter who holds it. There is the following check in the method:

```
if (_from != _msgSender()) revert OwnerNotCaller();
```

which shows an access control intention, but is actually redundant as the `_from` argument is not used in the burning process.

Recommendations

The code should check that the caller actually owns the NFT that is about to get burned - the current check does not do this. Update it accordingly.

8.2. High Findings

[H-01] Accepting input after randomness is requested can be exploited

Severity

Impact: High, as it can result in a griefing attack on an expected game winner

Likelihood: Medium, as it requires minting a new HoneyJar NFT

Description

The VRF security considerations docs explicitly mention that if an outcome in the contract depends on user-supplied inputs (in this case minting HoneyJar NFTs) and randomness, then the contract should stop accepting any additional user-supplied inputs after it submits the randomness request. The problem here is that in `fulfillRandomWords` the `_setFermentedJars` method is called where the number of HoneyJar NFTs minted for a bundle is used for the process of choosing the winning NFT - this means that the `fulfillRandomWords` transaction can be front-ran with a HoneyJar NFT mint and the winner will be different. This can result in a griefing attack for an expected winner of a game.

Recommendations

Decouple the randomness request and the user input from each other. Use only the user input that has been submitted pre-requesting randomness.

8.3. Medium Findings

[M-01] The `fulfillRandomWords` method might revert with out of gas error

Severity

Impact: High, as randomness won't be fulfilled

Likelihood: Low, as it requires misconfiguration of gas

Description

The `fulfillRandomWords` method in `HibernationDen` calls the internal `_setFermentedJars` method which loops over the `fermentedJars` array and also has an external call. This is a potential problem as this code might require a lot of gas and make the `fulfillRandomWords` method revert which is problematic for a VRF integration (it is listed in the VRF Security Considerations docs).

Another such issue in the method is this code:

```
if (party.assetChainId != getChainId() && address(honeyJarPortal) != address
  (0) && address(this).balance != 0) {
    uint256 sendAmount = address(this).balance / party.checkpoints.length;
    honeyJarPortal.sendFermentedJars{value: sendAmount}(
        address(this), party.assetChainId, party.bundleId, fermentedJars
    );
}
```

The problem is that when `party.assetChainId != getChainId() && address(honeyJarPortal) != address(0)` are true, then the only thing left to go into the `if` statement is `address(this).balance != 0` - this is easily exploitable as anyone can send 1 wei of ETH into the contract, which will make the expression evaluate to `true`. This will add additional gas cost overhead as it will execute an external call that has more logic, and also the cross-chain call is almost certainly failing as the `sendAmount` is possible to have rounded down to zero (if `address(this).balance` was 1 but `party.checkpoints.length` was more than 1).

Recommendations

Consider caching the randomness received and then letting an externally owed account for example to actually make the `_setFermentedJars` call so it can set the correct gas. Also check that the balance is enough to do the `sendFermentedJars` call, not just that the balance is non-zero.

[M-02] Contract `HibernationDen` can receive ETH but it can't be withdrawn

Severity

Impact: Medium, as it will result in stuck funds, but they will just have the value of gas refunded

Likelihood: Medium, as it will happen when there is a refund from a cross-chain call

Description

The `HibernationDen` contract has a `receive` method. This is mostly expected to be used for `LayerZero` refunds as the comment above the method says. The problem is that this gas refunds ETH won't be withdrawable as there is no method for ETH withdraw in the contract. Another issue is that anyone can mistakenly send ETH to `HibernationDen` and it will be stuck there.

Recommendations

Add a method that can withdraw ETH from the `HibernationDen` contract.

[M-03] Centralization attack vectors are present in the code

Severity

Impact: High, as some accounts can brick the game

Likelihood: Low, as it requires a malicious or compromised owner/admin account

Description

There are multiple centralization attack vectors present in the contracts. Examples are:

- `HibernationDen::setVRFConfig` - updating this with arbitrary values can break the game
- `HoneyJarPortal::setHibernationDen` - updating it to a random address will DoS the game
- `HoneyJarPortal::setAdapterParams` - using too low of a `gasLimit` value will result in out of gas reverts

Recommendations

Make the methods callable only once or add them to the constructors/initializer methods.

8.4. Low Findings

[L-01] Not enforced checkpoints gap size

For the `checkpoints` field in the `SlumberParty` struct we have the following comment:

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

This is just an assumption, as the place where the `checkpoints` value is set (`HibernationDen::addBundle`) does not actually enforce this. This couldn't actually lead to a serious problem but it is best that the comment is removed and actual checks & validations are implemented in the code, so the gap between the set checkpoints can't be too small.