Forgotten Playground Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, SpicyMeatball, Dan Ogurtsov

March 19th 2024 - March 23th 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 **forgotten-playland-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Forgotten Playground

Forgotten Playland ToyBox is a set of smart contracts that distributes via sales ERC1155 ToyBox which can then be redeemed randomly for ERC1155 Cosmetics. Whenever a new token is minted, randomness is fetched from the server and the contract consumes it to generate an id.

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 - b3b733ed5d7f33ea4bc794ca912bab2c356f471b

fixes review commit hash - ad99fbe727ebd61b59d012ecacdcfd3ca1903081

Scope

The following smart contracts were in scope of the audit:

- `ToyBox`
- `Cosmetics`
- `ZkRandMint`
- `TokenSaver`
- `CustomSale`
- `AbstractERC1155`

7. Executive Summary

Over the course of the security review, 0xunforgiven, SpicyMeatball, Dan Ogurtsov engaged with Forgotten Playground to review Forgotten Playground. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Forgotten Playground
Repository	https://github.com/Merit-Circle/forgotten-playland-contracts
Date	March 19th 2024 - March 23th 2024
Protocol Type	NFT sale

Findings Count

Severity	**Amount**
Critical	2
High	2
Medium	2
Low	5
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Uniswap oracle manipulation to buy for a lower price	Critical	Resolved
[C-02]	Mistake using primary price for customSaleWithPermit()	Critical	Resolved
[H-01]	Bypassing saleUserCap and whitelist	High	Resolved
[H-02]	Force buy ToyBox for anyone who sets approval for contract	High	Resolved
[M-01]	Wrong usage of block's timestamp instead of block's number	Medium	Resolved
[M-02]	ToyBox.sol contract lacks discountTokens setter	Medium	Resolved
[L-01]	Setting burnId after the openingTime can cause Cosmetic tokens to be minted by burning other ToyBox token	Low	Acknowledged
[L-02]	Last drop rate must be 100 otherwise mint would revert always	Low	Resolved
[L-03]	Centralization risk as admin can game the random minting process	Low	Acknowledged
[L-04]	Not checked openingTime and closingTime on setup	Low	Resolved
[L-05]	Multiple calls to bypass maxMintPerTx	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Uniswap oracle manipulation to buy for a lower price

Severity

Impact: High

Likelihood: High

Description

`uniswapV2Router.getAmountsIn()` is used to calculate the amount of `paymentToken` required for the amount in `referenceToken`. This feed is easily manipulated by a large swap in Uniswap pairs. So the attacker can in one transaction:

1. Flashloan `referenceToken`
2. Sell this `referenceToken` in the Uniswap pair buying `paymentToken`. The price of `referenceToken` is decreased up to almost zero.
3. Paying using `paymentToken` to mint in TokenBox. The manipulated price will help to spend a very small amount of `paymentToken` to buy TokenBox priced in `referenceToken`
4. Return flashloaned `referenceToken`.

Recommendations

TWAP is the recommended way of reading the price from Uniswap V2 pairs. But it is also can be manipulated for low liquidity pairs. Consider using centralized oracles like Chainlink. E.g. Chainlink feeds can be provided when allowing a token as paymentToken.

[C-02] Mistake using primary price for `customSaleWithPermit()`

Severity

Impact: High

Likelihood: High

Description

ToyBox uses `customSale()` and `customSaleWithPermit()` for non-primary sales, and these functions should not use data from the primary sales. But customSaleWithPermit() uses the price for the primary sale, which likely is not the intended behavior.

```
function customSaleWithPermit(
      uint256_amount,
      PermitSignaturecalldata_permitSignature,
      bytes32[]calldata_proof
    )
        external
        nonReentrant
        customSaleChecks
          (_permitSignature.owner, _permitSignature.token, _amount, _proof)
    {
        ...
        _collectWithPermit(
            _amount, getFullCustomPrice(
              price,
              _permitSignature.token
            ), saleStruct.referenceToken, _permitSignature
        );
        ...
    }
```

As a result, users signing approvals via permit will receive a different price.

Recommendations

Replace `price` with `saleStruct.price`, as in `customSale()`.

8.2. High Findings

[H-01] Bypassing `saleUserCap` and whitelist

Severity

Impact: Medium

Likelihood: High

Description

`ToyBox.customSaleChecks()` does many checks, but this check for a user is very likely wrong:

```
modifier customSaleChecks(
      address_receiver,
      address_paymentToken,
      uint256_amount,
      bytes32[]calldata_proof
    ) {
        ...
        // If the merkleRoot is set, check if the user is in the list
        if (saleStruct.merkleRoot != bytes32(0)) {
            bytes32 leaf = keccak256(abi.encode(msg.sender));
            if (!MerkleProof.verify(_proof, saleStruct.merkleRoot, leaf)) {
                revert InvalidProof();
            }
        }
        ...
```

The problem is that `msg.sender` is checked, when the "user" here is `_receiver`. It works correctly when they are the same, but it will be wrong in all other cases. Later in the code, `_receiver` is the target for checking `saleUserCap`, not `msg.sender`. It allows minting to different receivers bypassing `saleUserCap`. Moreover, `msg.sender` is checked when calling `customSaleWithPermit()` which is also probably wrong. In addition, these receivers are not checked for being whitelisted.

Recommendations

Replace `msg.sender` with `_receiver` in `customSaleChecks()`.

[H-02] Force buy ToyBox for anyone who sets approval for contract

Severity

Impact: High

Likelihood: Medium

Description

In ToyBox contract and `primarySaleWithPermit()` and `customSaleWithPermit()` code doesn't check that `msg.sender` is equal to the `permitSignature.owner`. Also valid permission signature is not enforced in `trustlessPermit()` so If someone has set approval for the ToyBox contract, it would be possible to call those functions with spoofed permission signature and buy ToyBox token for them without their permission. Attacker can spend all the users' tokens that gave spending allowance and also buy ToyBox when price is not fair.

Recommendations

Code should verify `msg.sender` to be equal to the `permitSignature.owner`.

8.3. Medium Findings

[M-01] Wrong usage of block's timestamp instead of block's number

Severity

Impact: Low

Likelihood: High

Description

When code wants to get last block hash it uses `blockhash(block.timestamp - 1)`.

```
bytes32 pseudoRandomness =
            keccak256(abi.encode(blockhash
              (block.timestamp - 1), msg.sender, nonces[msg.sender])) >> 3;
```

But according to Solidity docs:

```
blockhash(uint blockNumber) returns
  (bytes32): hash of the given block - only works for 256 most recent blocks
```

As a result `blockhash(block.timestamp - 1)` would be calculated for a nonexisting block and would always result in 0 so the `pesudoRandomness` would be more predictable and not according to the docs.

Recommendations

Change code to `blockhash(block.number - 1)`.

[M-02] `ToyBox.sol` contract lacks `discountTokens` setter

Impact: Low

Likelihood: High

Description

When user buys a ToyBox token from the sale, a discount is applied to the price

```
function getFullPrice
      (uint256 _price, address _paymentToken) internal view returns (uint256) {
>>      uint256 _discount = discountTokens[_paymentToken];
        if (_discount > 0) {
            _price = _price - (_price * _discount / 10000);
        }
        return _price;
    }

    function getFullCustomPrice
      (uint256 _price, address _paymentToken) internal view returns (uint256) {
>>      uint256 _discount = saleTokenDiscounts[customSaleActive][_paymentToken];
        if (_discount > 0) {
            _price = _price - (_price * _discount / 10000);
        }
        return _price;
    }
```

The `discountTokens` and `saleTokenDiscounts` mappings store the discount percentages for primary and custom sales, respectively. However, the sale manager is unable to customize discounts for primary sales using `discountTokens` due to the absence of a setter function, unlike for custom sale discounts managed through `setSaleTokenDiscounts/setSaleTokenDiscount` functions.

Recommendations

Consider either adding functions to set `discountTokens[_paymentToken]` or remove the step of calculating a discount for primary sales in `primarySale()` and `primarySaleWithPermit()`.

8.4. Low Findings

[L-01] Setting `burnId` after the `openingTime` can cause Cosmetic tokens to be minted by burning other ToyBox token

The default value of `burnId` is zero and if admin set it's value after `openingTime` then Cosmetic tokens can be minted by burning ToyBox token `id=0` too.

[L-02] Last drop rate must be 100 otherwise mint would revert always

There is no check in the `setRarityDistribution()` to make sure last drop rate is 100 and mint would revert if the last drop rate is not 100.

[L-03] Centralization risk as admin can game the random minting process

Admin can cause different issues during minting process like:

1. Changing the commitment to make sure to receive Ultra Rare tokens.
2. Doesn't generate proof for some users.

[L-04] Not checked `openingTime` and `closingTime` on setup

Consider additional checks on `openingTime` and `closingTime` setup that `openingTime < closingTime` in `ToyBox.setTimes()`, `ToyBox.setCustomSale()` and `ZKRandMint.setTimes()`.

[L-05] Multiple calls to bypass maxMintPerTx

`maxMintPerTx` name and comments suggest that the variable should limit the amount purchased per transaction.

```
// Can't mint more per tx than allowed
        if (_amount > maxMintPerTx) {
            revert ExceedsMaxMintPerTx();
        }
```

In fact, it is a limitation per call. As a result, the simplest strategy to bypass the limit is just calling multiple times per transaction.

Ensure that this behavior is intentional. If it is not, add logic to set limits correctly for transactions.