SOLIDIFIED

Audit Report for Anibear - October 5, 2022

Summary

Audit Report prepared by Solidified covering the Anibear smart contracts.

Process and Delivery

Independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on September 30, 2022, and the results are presented here.

Audited Files

The source code has been supplied in a compressed ZIP format with SHA256 hash:
`805107c01a10c8cf2d5851b5990445bc30a6239c608d2eca142dabe76d292f60`

Files audited:

```
contracts
├── AnicubePFP.sol
└── ERC721Admin.sol
```

Update: the Anicube team provided a new version with fixes to issues encountered on October 5, 2022 in a compressed ZIP format with SHA256 hash:
`8533d20d6efe95e7b31a05dc69b87f2777fb11e8801115f992018632dbebd482`

Files audited:

```
contracts
├── Anibear.sol
└── ERC721Admin.sol
```

Intended Behavior

The contracts implement the Anibear NFT.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Low	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	N/A	-

SOLIDIFIED

Audit Report for Anibear - October 5, 2022

Issues Found

Solidified found that the Anibear contracts contain 0 critical issues, 2 major issues, 3 minor issues, and 6 informational notes.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Non-compliant ERC20 token transfers might fail without reverting	Major	Resolved
2	The maximum NFT supply can be exceeded by the owner and the admin	Major	Resolved
3	Native prices can be set to zero	Minor	Resolved
4	NFT minter can lose accidentally sent native tokens	Minor	Resolved
5	The owner and admin can increase the NFT price at any time and force users to pay more	Minor	Acknowledged
6	baseURI can be pre-maturely revealed	Note	Resolved
7	The function ownerMint does not take into account any limits	Note	Acknowledged
8	The initialization of state variables is error-prone	Note	Acknowledged
9	The function flipSaleState might be error-prone	Note	Resolved
10	No license included in the Anibear.sol contact	Note	Resolved
11	Miscellaneous Comments	Note	Resolved / Acknowledged

Critical Issues

No critical issues have been found.

Major Issues

1. Non-compliant ERC20 token transfers might fail without reverting

If `PaymentType.TOKEN` is selected as the payment method, the function `safeMint` transfers the corresponding number of tokens per NFT, multiplied by the number of NFTs the buyer wants to purchase. However, the transfer can fail for non-compliant ERC20 tokens without reverting and thus, with no way of knowing that the transfer in line `149` did not happen. Subsequently the execution flow will continue as normal, minting the tokens for the buyer (if all other conditions are met), essentially free. Also, the withdrawal of such non-compliant tokens in the function `withdrawToken` could fail, return false and go unnoticed.

Note

This issue was flagged as major because, even though its implications are actually crucial, the tokens used are whitelisted from the owner / admin, thus it is not expected to allow payments from non-compliant ERC20 tokens.

Recommendation

We recommend using OpenZeppelin's `SafeERC20.safeTransferFrom` function.

Status

Resolved

2. The maximum NFT supply can be exceeded by the owner and the admin

The `Anibear` contract has a fixed maximum token supply which is strictly enforced in the `safeMint` function. However, the special minting functions `ownerSafeMint` and `ownerMint`, only callable by the owner and the admin, can be used to mint more NFTs than the maximum supply. This can be possible as the current token supply is iteratively incremented and allows bypassing the maximum supply check by reentering the function with the `ERC721.onERC721Received` callback.

Recommendation

We recommend adding reentrancy protection to the functions `ownerMint` and `ownerSafeMint`.

Status

Resolved

Minor Issues

3. Native prices can be set to zero

The native prices for each miniting method are set through the `updateNativePrice` function. The aforementioned function is not called when the contract is initialized, thus the values remain zero for all minting methods. If a minting method goes live by flipping its state using `flipSaleState` before the native prices are set correctly for that minting method, users can mint for free, without sending native tokens to `safeMint`.

Note

This is prevented in the case of a token as a payment method since `safeMint` ensures that price is more than zero.

Recommendation

We recommend ensuring `updateNativePrice` parameter `_price` cannot be set to zero. Also, consider either initializing those values from the constructor, or making sure `msg.value` is not zero for `mintingMethod.PUBLIC`. Alternatively, consider ensuring that the state of the `mintingMethod` cannot be flipped to `true` if `nativePrices[mintingMethod]` is zero.

Status

Resolved

4. NFT minter can lose accidentally sent native tokens

Depending on the provided `_paymentType`, NFTs can be minted with `safeMint` by paying with the native token or with accepted ERC-20 tokens. However, if a user intends to pay with ERC-20 tokens by using the appropriate payment type `PaymentType.TOKEN`, but accidentally sends native tokens, the user will lose the native tokens sent.

Recommendation

We recommend asserting that no native tokens are received when the payment type `PaymentType.TOKEN` is used.

Status

Resolved

5. The owner and admin can increase the NFT price at any time and force users to pay more

---

The price of an NFT varies based on the payment type and the minting method. The different costs are stored in the storage variables `nativePrices` and `tokenPrices`. However, the prices can be increased anytime without a timelock enforced or slippage protection. This can be used to force users to unwillingly pay more for minting an NFT with `safeMint`.

Recommendation

We recommend adding a maximum price for the NFT as a slippage mechanism that the minter can set in the `safeMint` function. This way, if the price of the NFT increases above the maximum price, the transaction will revert and the NFT is not minted.

Status

Acknowledged. Team's response: *"We understand the issue and the proposed plan from Animoca is to update the price before sale goes live and has no intention to flip sale states then update prices again. [...] We are locking in and announcing the token prices before the sale goes live and will not be updating the prices/increasing or decreasing price during the sale period. The ownership of the contract will be transferred to Animoca"*.

Informational Notes

6. `baseURI` can be pre-maturely revealed

There is a clear intention to not reveal the metadata of the NFTs initially. Therefore, a `preRevealURI` is used and the `toggleReveal` function is called when the metadata should be revealed. However, the `baseURI` can be set by the owner or admin with the use of the function `setBaseURI`, before it should have been revealed. This will make the `baseURI` and subsequently `tokenURIs` available pre-maturely (leaked).

Recommendation

We recommend only allowing `baseURI` to be set if the revealed flag is set to `true`.

Status

Resolved

7. The function ownerMint does not take into account any limits

The function `ownerMint` allows the owner to mint any number of NFTs up to the total remaining supply balance, without taking into account if any of the minting methods is active, if the `MAX_MULTIMINT` limit is exceeded or if the receiver has exceeded the `MAX_MINTS_PER_ADDRESS`. Also, if the function is intended to be used before public sale is active, it might also need to ensure that the receiver being in the allowlist and the allowlist has not exceeded its max supply. Except the total supply, no other counters are updated.

Recommendation
Although this function is intended to be for “airdropping” purposes, it is best practice to enforce some reasonable limits to increase trust in the community that the function will not be used extensively to the disadvantage of other investors. “Airdropping” often distributes just one ERC721 token per whitelisted user.

Status
Acknowledged

8. The initialization of state variables is error-prone

The constructor passes parameters to initialize the state variable mappings `MAX_SUPPLY`, `MAX_MULTIMINT`, `MAX_MINTS_PER_ADDRESS` and `MAX_SUPPLIES`. The nine values are passed as a `uint256` array named `params`. As no distinct variables or arrays are used for each state variable it makes the initialization error-prone, risking to set the immutable values mistakenly. For example, by intuition someone could assume that `params[6]` is `MAX_SUPPLIES[MintingMethod.PUBLIC]` instead of `MAX_SUPPLIES[MintingMethod.ALLOWLIST_A]`, as the maximum supply of the public minting method is deliberately uninitialized.

Recommendation
We recommend passing distinct, named parameters to set the state variables.

Status
Acknowledged

9. The function `flipSaleState` might be error-prone

The state of each minting method is modified by the function `flipSaleState` which essentially reverts the current sale state of each minting method. Whilst initially all sale methods are set to `false` and the owner / admin can correspondingly change their state to `true`, having a flip function is error prone. If unsure, the owner / admin should first query the current state of the minting method and then revert if necessary.

Recommendation
We recommend passing a second `boolean` parameter to the function to set the minting method state to `true` or `false` accordingly.

Status
Resolved

10. No license included in the `Anibear.sol` contact

Since making the source code available has legal implications, it is best recommended to add licensing in all Solidity files. Thus, SPDX license has become mandatory since version `0.6.8`

Recommendation
Before publishing, we recommend adding a comment containing `SPDX-License-Identifier: <SPDX-License>` to each source file. Use `SPDX-License-Identifier: UNLICENSED` for non-open-source code. Please see https://spdx.org for more information.

Status
Resolved

11. Miscellaneous Comments

The following are suggestions to improve the overall code quality and readability.

- Unnecessary import: Openzeppelin's `PaymentSplitter` (Resolved)
- Unnecessary `using Strings for uint256` statement in line `234`, the `toString` method is accessed by `Strings.toString(tokenId)`, not by `tokenId.toString()` (Resolved)
- Storage variables can be declared `immutable` to save gas:
  - `MAX_SUPPLY`
  - `MAX_MULTIMINT`

  (Resolved)
- We recommend emitting events for important state variable changes if off-chain monitoring is required (Acknowledged)

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Anicube Entertainment Limited or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Oak Security GmbH