SOLIDIFIED

Audit Report for L2LAB FOUNDATION LTD - October 9, 2022

Summary

Audit Report prepared by Solidified covering the crowd sale for the Oases NFT marketplace

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on August 19, 2022, and the results are presented here.

Audited Files

The source code has been supplied in the following source code repository:

Repo: https://github.com/oases-team/oases-crowdfund
Commit hash: `ac1bd8d7d77163d441ffea3309271165b4fd5ac3`

Note that the contracts are intended to be deployed behind a proxy, but this proxy was not part of the audit.

Intended Behavior

The audited codebase implements a crowd sale for NFTs.

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	Medium	-
Level of Documentation	Low	-
Test Coverage	High	-

Issues Found

Solidified found that the crowd sale contracts contain no critical issues, 4 major issues, 11 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	Distributor.sol: transfer used for sending royalties	Major	Fixed
2	Intransparent drawing / minting procedure could introduce errors	Major	Acknowledged
3	OasesERC721.sol: Centralization risk	Major	Partially Fixed
4	CrowdFunding.sol: Treasury manager can withdraw all funds at all times	Major	Acknowledged
5	AdvancedERC721.sol: Total supply is not strictly enforced	Minor	Fixed
6	OasesERC721.sol: No zero address check for royalty receiver	Minor	Fixed
7	OasesERC721.sol: Unnecessary transfer when fee is royalty fee is zero	Minor	Fixed
8	OasesERC721.sol: Frontrunning possible in trade()	Minor	Acknowledged
9	No need to inherit from Upgradeable contracts	Minor	Acknowledged
10	receive() instead of fallback() can be used	Minor	Fixed
11	CrowdFunding.sol: _verifierAddress might be 0	Minor	Fixed
12	CrowdFunding.sol: Maximum number of lots per user circumventable	Minor	Fixed

13	CrowdFunding.sol: Loss of precision when rounding can lead to lost staking rewards	Minor	Fixed
14	CrowdFundingCore.sol: Anyone can issue a new crowd funding and NFT	Minor	Fixed
15	CrowdFunding.sol: A user without lots can call stakeRoyalty	Minor	Fixed
16	AdvancedERC721.sol: Specification of getTotalFundRemaining() unclear	Info	Fixed
17	CrowdFunding.sol: New NFT owners do not receive any staking rewards	Info	Acknowledged
18	Save gas using != 0 rather than > 0	Info	Fixed
19	AdvancedERC721.sol: Does not implement IAdvancedERC721 interface	Info	Fixed
20	Distributor.sol: Unnecessary transfer when ethValue - receiverValue = 0	Info	Acknowledged
21	Distributor.sol: Unnecessary use of payable and transfer()	Info	Fixed

Critical Issues

No critical issues have been found.

Major Issues

1. `Distributor.sol`: `transfer` used for sending royalties

The `fallback()` function of Distributor uses `transfer()` instead of the `transferETH()` function that is used everywhere else. As `transfer()` only comes with a 2300 gas stipend, its use is discouraged. If the royalty receiver is for instance a multi-sig wallet with some complicated business logic in its `receive()` function, the transaction will revert, which means that no ETH is transferred to the crowd funding address.

Recommendation

Use `transferETH()` instead of `transfer()`.

2. Intransparent drawing / minting procedure could introduce errors

`batchMint()` in `CrowdFunding` simply consumes signatures that were signed by the verifier address to allow the winners to mint their tokens. Therefore, it is not verifiable if the drawing procedure is fair. Furthermore, depending on how this is implemented, errors could be introduced. The same lot can be used to retrieve multiple tokens when the user has multiple signatures for it.

Recommendation

Ideally, it should be verifiable for the user how the drawing is done. If this is not desired, it should at least not be possible to use the same lot for retrieving multiple tokens.

3. OasesERC721.sol: Centralization risk

The owner of the `OasesERC721` contract is able to approve addresses that can then transfer all NFTs. Furthermore, the transfer proxy address has this approval per default.

Recommendation

Ideally, no address (that is potentially an externally owned account) should be able to transfer the NFTs of all users. If this is not avoidable, users should be aware of the risk.

Status

Partially fixed: The transfer proxy (that we did not audit) still has this permission, but it is no longer possible to give the permission to other users.

4. CrowdFunding.sol: Treasury manager can withdraw all funds at all times

`emergencyWithdraw()` can be used by the treasury manager at all times to withdraw all of the funds in the contract. Therefore, it could also be used when the sale will not be successful before the user can call `refund()`.

Recommendation

Remove the function or at least require that it is called after the end time plus some time delta.

Minor Issues

5. AdvancedERC721.sol: Total supply is not strictly enforced

In `batchMint()` within `AdvancedERC721`, there is a check that the `tokenId` is smaller or equal to `_MAX_TOTAL_SUPPLY` to ensure that not too many tokens are minted. However, `tokenId` 0 can also be minted (when the verifier address signs the corresponding data for it), which would result in `_MAX_TOTAL_SUPPLY + 1` tokens.

Recommendation

Check that `tokenId` is not 0.

6. `OasesERC721.sol`: No zero address check for royalty receiver

In `OasesERC721`, there is no check for the zero address when setting the fee receiver (neither in the constructor, nor in `setDefaultRoyalty` / `setTokenRoyalty`). The consequences of having a zero address there are severe (as `transferRoyalties` will transfer the ETH to the zero address), it is therefore recommended to check that this never happens.

Recommendation

Add a check for the zero address when setting the royalty receivers.

7. `OasesERC721.sol`: Unnecessary transfer when royalty fee is zero

In transferRoyalties, the `fee` might be zero (e.g., because it is set to zero for a specific token). A transfer will be initiated in these situations, nevertheless.

Recommendation

Only initiate the transfer when `fee` is larger than zero..

8. `OasesERC721.sol`: Frontrunning possible in `trade()`

It is possible to front-run buy orders of users by buying the NFT and immediately setting the price higher. This can defeat the purpose of refunding a user when he paid too much. In such situations, he might get front-run and therefore still not receive any ETH back.

Recommendation

While this is difficult to avoid with the current design, the user should be made aware of it.

9. No need to inherit from Upgradeable contracts

`AdvancedERC721Factory` and `CrowdFundingCore` inherit from `OwnableUpgradeable`. However, as they are not upgradeable / behind a proxy, this is not necessary and introduces unnecessary complications.

Recommendation

Use `Ownable` instead of `OwnableUpgradeable`.

10. `receive()` instead of `fallback()` can be used

`CrowdFunding` and `Distributor` both only have a `fallback()` function and no `receive()` function, but the `fallback()` function is only called with empty calldata. While this works, it is recommended to use the `receive()` function for this (as it is specifically designed to receive ETH when the calldata is empty), which will also make the current compiler warning go away.

Recommendation

Use the `receive()` function instead of the `fallback()` function.

11. `CrowdFunding.sol`: `_verifierAddress` might be 0

When setting the `_verifierAddress` (either in the constructor or by using `setVerifierAddress`), it is not verified that the address is not 0. The consequences of this would be severe, any invalid signature could then be used to mint the NFTs. Therefore, it is recommended to check that this address is never 0.

Recommendation

When setting the address, check that it is not equal to 0.

12. CrowdFunding.sol: Maximum number of lots per user circumventable

It is enforced that an individual address can buy `_LOT_LIMITED_NUMBER` at a max. However, as this check is per address, it is easily circumventable by sybil attacks, i.e. by calling `buyLots()` with different addresses.

Recommendation
Consider using a whitelisting scheme if the maximum number of lots per user should be enforced.

13. CrowdFunding.sol: Loss of precision when rounding can lead to lost staking rewards

In `stakeRoyalty()`, the added `amount` is always divided by the number of sold lots. Depending on how often the function is called and how large the amounts are, this can lead to a significant loss of precision, which means that the corresponding staking rewards are not retrievable (by no one, the owner could only retrieve them with `emergencyWithdraw()`, which would also remove the rewards of the other users).

Recommendation
Instead of dividing on every `stakeRoyalty()` call, store the whole amount and only divide by the number of lots when claiming. When there are many `stakeRoyalty()` calls with low amounts, this will result in significantly higher precision.

14. CrowdFundingCore.sol: Anyone can issue a new crowd funding and NFT

`issueNFTWithCrowdFunding` is callable by anyone and will create a new `CrowdFunding` contract and NFT. While this is not problematic in itself, it can make the life of phishers easier. A malicious attacker could deploy its own `CrowdFunding` contract like this and trick users into paying into it. Because it was created by an Oases contract, user might think it is legitimate.

Recommendation

Consider restricting the creation of new `CrowdFunding` / NFT contracts.

15. `CrowdFunding.sol`: A user without lots can call `stakeRoyalty`

`stakeRoyalty` is callable by anyone, meaning a user without lots can deposit ether without getting any return by calling `claim`. If the intent is that only an Oases contract should call `stakeRoyalty`, then include a check that only the Oases owned contract address can call `stakeRoyalty`.

Recommendation

Consider restricting who can call the `stakeRoyalty` function. Either, restrict users without lots from calling `stakeRoyalty`, or restrict the function to only be called by the appropriate contract.

Informational Notes

16. `CrowdFunding.sol`: Specification of `getTotalFundRemaining()` unclear

`getTotalFundRemaining()` always multiplies the lot price by the sold lots (before a withdrawal has happened). However, this is not necessarily the amount that can be withdrawn. When the crowd sale does not succeed (i.e., less lots than the NFT supply are sold), the amount is 0. Therefore, depending on how this value is used, the returned number might be wrong. Even if the sale succeeds, this amount is not retrievable before `_lockedExpirationTime`.

Recommendation

Document the purpose of this function. If it should return the amount that is retrievable at the moment of the call, update the implementation accordingly.

17. `CrowdFunding.sol`: New NFT owners do not receive any staking rewards

`claim()` uses the mapping `_lotBook` to determine the staking rewards a user gets. However, the staking rewards can be continuously increased after the crowd sale and the owner of a NFT might change after the sale (for instance, using the `trade` function). In such scenarios, the old owner will still be able to claim all staking rewards for the NFT, whereas the new owner cannot claim any. While this might be intended (as not only owners, but all participants get staking rewards), it is recommended to document this behavior clearly.

18. Save gas using != 0 rather than > 0

In various places throughout the code, a `require(x > 0);` statement is used to check that a state variable has been deleted or uninitialized.

Recommendation

Consider saving gas by using a `require(x != 0);` check instead.

19. AdvancedERC721.sol: Does not implement IAdvancedERC721 interface

Line 214 in `Crowdfunding.sol` uses `IAdvancedERC721` to call the `batchMint` function, but the interface is not implemented by `AdvancedERC721`. If the signature of the function in the interface changes this could result in an error.

Recommendation

Consider implementing the `IAdvancedERC721` interface.

20. Distributor.sol: Unnecessary transfer when ethValue - receiverValue = 0

If `distributedBasisPoint` equals 10000, then `ethValue - receiverValue` in `fallback()` will equal 0 resulting in an unnecessary transfer of 0 eth.

Recommendation

Consider adding a check to ensure ethValue - receiverValue > 0 before transferring eth to `_crowdFundingAddress`.

21. Distributor.sol: Unnecessary use of payable

`_royaltyReceiverAdress` is marked `payable` when this is not done anywhere else in the code.

Recommendation

Consider being consistent and remove `payable`.

Disclaimer