SOLIDIFIED

Audit Report for L2LAB FOUNDATION LTD - May 09, 2022

Summary

Audit Report prepared by Solidified covering 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 May 6, 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-contracts
Commit hash: `cac092b5e8399467c1bd1534189a8efc73fdec63`

Intended Behavior

The audited codebase implements an NFT market place.

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	Medium	-
Code readability and clarity	Low	-
Level of Documentation	Low	No documentation provided
Test Coverage	High	-

Issues Found

Solidified found that the Oases contracts contain no critical issues, 1 major issue, 5 minor issues, and 8 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	OrderVerifier.sol: Signature validation for smart contracts in verifyOrder() may fail, although the signature is valid	Major	Pending
2	Centralized design Requires trust in deployer and requires very careful key management	Minor	Pending
3	Accidental overpayment is not refunded	Minor	Pending
4	No Limit On Royalties	Minor	Pending
5	No Limit On Protocol Fees	Minor	Pending
6	transferPayment() underpayment	Minor	Pending
7	OasesMatchingCore.sol: Different logic for who can cancel orders in cancelOrders() than who can match in matchOrders()	Note	Pending
8	SignatureLibrary.sol: EIP-2098 not supported	Note	Pending
9	ERC20TransferProxy.sol not checking safeTransferFrom return value	Note	Pending
10	ERC721Oases.sol: Inconsistent usage of PriceChanged event	Note	Pending
11	OasesCashierManager.sol: Missing null check in setRoyaltiesRegistry	Note	Pending
12	ERC721LazyMintTransferProxy.sol: No explicit initializer	Note	Pending

13	Possibility of Frontrunning	Note	Pending
14	Gas Optimizations	Note	Pending

Critical Issues

No critical issues have been found.

Major Issues

1. `OrderVerifier.sol`: Signature validation for smart contracts in `verifyOrder()` may fail, although the signature is valid

`_hashTypedDataV4(orderHash).recover(signature)` reverts for invalid signatures. However, `recover()` may deem a signature as invalid which would pass the `IERC1271Upgradeable(order.maker).isValidSignature()` check, as the ERC 1271 standard does not impose any standards for the signature.

Recommendation

Either catch the reversion or use `SignatureCheckerUpgradeable.isValidSignatureNow` which combines both checks.

Minor Issues

2. Centralized design Requires trust in deployer and requires very careful key management

During our audit process, we found several external functions with `onlyOwner` modifiers that can be executed by the deployer (Owner) only. Moreover, the smart contracts are upgradable, which can make the ecosystem centralized, unless there is a DAO mechanism in place for the admin selection. Note that apart from the centralization problem if the private key of the deployer gets compromised, the whole Oases ecosystem of Oasescould be at risk.

Recommendation

We would recommend applying a multi-sig control layer to the `onlyOwner` functions to protect the system in the case of accidental key loss or compromise.

3. Accidental overpayment is not refunded

In the trade function of `ERC721Oases`, if a user overpays when purchasing, the original owner will receive the overpayment.

Recommendation

Require `msg.value` to match the price, or send the excess funds back to the buyer as is done in `OasesMatchingCore`.

4. No Limit On Royalties

There is no upper bound on royalties. In the trade function of `ERC721Oases`, the royalties can take the total amount of payment, leaving nothing for the original owner to receive.

Recommendation

Consider a limit on the total amount of royalties on an ERC721, or a reversion in the event there is nothing left for the original owner to receive.

5. No Limit On Protocol Fees

There is no limit on the protocol fees in `OasesCashierManager`, and there is no requirement that the amount paid for an NFT is not fully consumed by the protocol fees.

Recommendation

Consider a limit on the protocol fees and/or a reversion in the event there is nothing left for the original owner to receive.

6. transferPayment() underpayment

transferPayment() in `OasesCashierManager` checks where rest is > 0, rather than amountToPay as is done in Rarible, leading to underpayment when rest is 0, or inadvertent reversion when rest is < 0.

Recommendation

Match the Rarible implementation by using amountToPay in the if statement, rather than rest.

Informational Notes

7. OasesMatchingCore.sol: Different logic for who can cancel orders in cancelOrders() than who can match in matchOrders()

In `matchOrders()`, it is possible to execute on behalf of a user if a valid signature is provided. However, the order can only be canceled if the transaction is executed by the maker in `cancelOrders()`. This means that an order that was created using a signature cannot be canceled with a signature.

Recommendation

If this behavior is not desired, an option to cancel an order with a signature should be added.

8. SignatureLibrary.sol: EIP-2098 not supported

Compact Signature Representation (https://eips.ethereum.org/EIPS/eip-2098) is not supported by the currently used signature library.

Recommendation

Consider using a newer library that supports these signatures.

9. ERC20TransferProxy.sol not checking safeTransferFrom return value

The contract uses safeTransferFrom without checking the return value.

Recommendation

Consider a require statement around ERC20TransferProxy's safeTransferFrom as is done in Rarible.

10. ERC721Oases.sol: Inconsistent usage of PriceChanged event

When the price is set to 0 via `transferFrom()` or `safeTransferFrom()`, the event `PriceChanged` is emitted. In contrast, when it is set to 0 in `trade()`, it is not emitted.

Recommendation

Consider using `_setPrice()` in `trade()` to have a consistent behavior.

11. OasesCashierManager.sol: Missing null check in setRoyaltiesRegistry

There is no check for the null address in `setRoyaltiesRegistry()`.

Recommendation

Consider adding a null check to the function.

12. ERC721LazyMintTransferProxy.sol: No explicit initializer

In contrast to other contracts that also only call `__Operators_init()` (e.g., `ERC20TransferProxy` or `NFTTransferProxy`), `ERC721LazyMintTransferProxy` does not have an initializer on its own and you have to remember to call `__Operators_init()` after deployment.

Recommendation

Consider adding an initializer to avoid errors.

13. Possibility of Frontrunning

The actual exchange rate of a trade can depend on which order is right and which one is left. Therefore, there can be scenarios where it is profitable to front-run a matchOrders() call (possibly with changed left and right sides) to get an asset cheaper, and then execute a second trade (again with carefully calculated left and right sides).

Recommendation

The design could be changed such that the left / right side is always the same (e.g., by using the lower hash as the left side). While this would not eliminate all frontrunning possibilities, the ones that extract profit by changing the sides would no longer be possible.

14. Gas Optimizations

Custom errors: We have noticed that you use the traditional way of error handling in your smart contracts. However, since your solidity version is 0.8, you can significantly reduce the gas consumption of your contracts by using custom errors.

Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries). Hence, we would like to recommend making custom errors and using them instead of `"String"` to reduce the gas consumption in the smart contracts. If you use a string to show the error, the amount of the gas depends on the string length.

Lack of using calldata in external functions: Memory's lifetime is limited to a function call and is meant to be used to temporarily store variables and their values. Values stored in memory do not persist on the network after the transaction has been completed. `calldata` is similar to `memory` in that it is a data location where items are stored. It is a special data location that contains the function arguments, only available for external function call parameters. Replacing `memory` with `calldata` in external functions is suggested to reduce gas consumption.

Lack of external functions: In various places throughout the code, `external` can be used in place of `public`.

Recommendation

Consider applying the recommended gas optimizations.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of L2LAB FOUNDATION LTD 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