SOLIDIFIED

Audit Report for Neon - May 13, 2021

Summary

Audit Report prepared by Solidified covering the Neon NFT and auction smart contracts.

Process and Delivery

Two (2) independent Solidified experts performed an unbiased and isolated audit of the code in several rounds. The debrief took place on 13 May 2021.

Audited Files

The source code has been supplied in the form of a GitHub repository:

https://github.com/kdzapp/neon-contracts

Commit number: `006018583e4a6a655360813990 98d0a994794842`

The scope of the audit was limited to the following files:

```
contracts
├── Auction.sol
├── EscrowPayment.sol
└── NeonERC1155.sol
```

Intended Behavior

The smart contracts implement an ERC-1155 token with a related auction and escrow contract.

Code Complexity and Test Coverage

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 equate to a higher risk. Certain bugs are more easily detected in unit testing than a security audit and vice versa. It is, therefore, more likely that undetected issues remain if the test coverage is low or non-existent.

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

Issues Found

Solidified found that the Neon contracts contain 3 critical issues, 2 major issues, 1 minor issue, in addition to 2 informational notes.

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

Issue #	Description	Severity	Status
1	Auction.sol: Anyone can steal Neon tokens held by the contract.	Critical	Pending
2	A bidder can block all subsequent bids and win the auction by default	Critical	Pending
3	Auction.sol: Reentrancy with calls to payout() and refund() can cause contract to be drained	Critical	Pending
4	Funds are lost if an auction is not ended by the seller or the contract owner	Major	Pending
5	Auction.sol: Tokens transferred to the contract using ERC1155 safeBatchTransferFrom() function will be lost/locked	Major	Pending
6	A bidder can block auctions to be canceled or his bid to be expired	Minor	Pending
7	NeonERC1155.sol: IPFS URL could be stored in OpenZeppelin base	Note	-
8	NeonERC1155.sol: function mintTokens(): the recipient becomes token creator, not the msg.sender.	Note	-

Critical Issues

1. Auction.sol: Anyone can steal Neon tokens held by the contract.

By calling the function `onERC1155Received()` anyone can create a new auction and become a seller for any token which is already owned by the contract (has another auction going on).
The malicious actor can then `cancelAuction()` and get the token sent to him.

Recommendation
Consider adding a check to the function `onERC1155Received()` which requires that the `msg.sender` is the `Neon` token contract.

2. A bidder can block all subsequent bids and win the auction by default

The auction relies on the execution of a refund operation to the previous lead bidder whenever a higher bid is received. However, it's quite easy for a bidder to block all subsequent bids from being accepted in a Denial of Service attack. If the bid leader wallet is a smart contract, it can be implemented in a way to revert instead of accepting the refund payment. This will make the whole transaction revert, and never accept the new big.
The same is true to a lesser extent for function `payout()`, since, for example, a creator could block the whole operation by not accepting the funds. However, in this case, there would probably be no incentive to do so.

Recommendation
Let the previous auction leader pull the refund payment from the contract in a separate transaction instead of pushing the refund out.

3. Auction.sol: Reentrancy with calls to payout() and refund() can cause contract to be drained

Throughout the auction contract, calls to `payout()` and `refund()` result in external calls, after which state changes occur. This may lead to reentrancy vulnerabilities, which could be exploited by a smart contract calling back into the auction code.
Two exploitation scenarios have been identified related to the function `cancelAuction()`. This can be used by a malicious seller to steal Ethereum held in `Auction.sol` contract.

Scenario 1:
1. The seller is s smart contract which creates an auction for a `Neon` token;
2. The seller makes a bid on the auction;
3. The seller calls `cancelAuction()`. The `EscrowPayment.sol` contract tries to refund previously bid Ethereum to the seller. During this refund call, the seller recursively calls `cancelAuction()` multiple times until a desired amount of Ethereum is extracted from the `Auction.sol` contract.

Scenario 2:
1. The seller is s smart contract which creates an auction for a `Neon` token;
2. The seller makes a bid on the auction;
3. When someone else makes a bid, the `EscrowPayment.sol` contract would refund previously bid Ethereum to the seller. During this refund call, the seller calls `cancelAuction()` - this would result in the seller getting twice as much Ethereum as previously bid. The new bidder would lose his funds.

Recommendation
Consider using a reentrancy guard and/or not performing state changes after an external call.

Major Issues

4. Auction.sol: Funds are lost if an auction is not ended by the seller or the contract owner

Funds are only paid out or refunded if an auction holder decides to cancel the auction or accept a bid. Alternatively, the contract owner can call `expireBid()`. Bidders have to rely on this and have no way to recover their funds by other means.

Recommendation
Consider placing a time limit on bids to be accepted and allowing users to cancel a bit after this limit has been reached.

5. Auction.sol: Tokens transferred to the contract using ERC1155 safeBatchTransferFrom() function will be lost/locked

The `Auction.sol` contract silently accepts batch transfers but does not register new auctions for them. Thus, the tokens received by batch transfers will be locked in the contract with no way to retrieve them back.

Recommendation
Override the base `ERC1155Holder.onERC1155BatchReceived()` function and either revert or implement correct accounting of received tokens.

Minor Issues

6. Auction.sol: A bidder can block auctions to be canceled or his bid to be expired

A bidder can prevent an auction from being canceled by calling `cancelAuction()` or the contract owner from expiring a bid with `expireBid()`. This is due to the same DoS scenario described in the previous issue. The bidder can just revert the refund transaction. This will lead the auctioned tokens to be trapped in the contract. This issue is aggravated by the fact that a bit of 1 Wei is enough to perform such an attack.
This issue also applies to creators being able to block auctions by making royalty payments fail.

Recommendation
Let the ETH receiver pull the payment from the contract in a separate transaction instead of pushing it out.

Informational Notes

7. `NeonERC1155.sol`: IPFS URL could be stored in OpenZeppelin base

The contract provides a mapping to store URLs. However, the ERC-1155 OpenZepplein contracts used as a base for this contract already support this.

Recommendation
Consider using the OZ Metadata extension (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/extensions/IERC1155MetadataURI.sol)

8. `NeonERC1155.sol`: function `mintTokens()`: the `recipient` becomes token `creator`, not the `msg.sender`.

The contract keeps track of creators but automatically assigns this to the `recipient` parameter when minting. This seems unintentional since a creator might mint the token for someone else.

Recommendation

Consider assigning the caller of the `mint()` function the creator role.

Disclaimer