SOLIDIFIED

Audit Report for NIFTEX - March 04, 2021

Summary

Audit Report prepared by Solidified covering the NIFTEX smart contract.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on February 26, 2021, and the results are presented here.

Audited Files

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

https://github.com/metalithio/niftex-v2-contracts

Commit number: `76d1407ff5a6dc0d2593dc76dcd33208a9b82b2a`

Intended Behavior

The smart contracts implement a protocol for shared NFT ownership, including a sharded wallet, wallet governance, and sale and distribution models.

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	High	-
Level of Documentation	Medium-High	-
Test Coverage	Medium	-

Issues Found

Solidified found that the NIFTEX contracts contain no critical issue, no major issue, 3 minor issues, in addition to 5 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	BuyoutModule.sol: External calls before state change in closeBuyout()	Minor	Resolved
2	ShardedWalletFactory.sol: Missing zero checks in mintWallet()	Minor	Resolved
3	BondingCurve3.sol: Bonding curve potentially vulnerable to flashloans if used with incorrect parameters	Minor	Pending
4	BondingCurve.sol: Tautologies in pre-condition checks supplyShards() and supplyEther()	Note	-
5	ShardeWallet.sol: Ignored return value of external call in moduleExecute()	Note	-
6	ModuleBase.sol: Unused modifier	Note	-
7	ActionModule.sol: Consider declaring functions schedule() & execute() as external instead of public	Note	-
8	Consider providing require error messages	Note	-

Critical Issues

No critical issues have been found.

Major Issues

No major issues have been found.

Minor Issues

1. `BuyoutModule.sol`: External calls before state change in `closeBuyout()`

This function deletes internal data-structures after the ETH value is sent out (line 69). If the receiver is a smart contract unexpected side effects might occur through the receiver interacting with the protocol before the state is changed. This is unlikely to cause a reentrancy issue due to the permissioning in this case, but it is generally best to avoid state changes after external calls.

Recommendation

Consider delaying the external call until after the state changes.

Update

Resolved

2. `ShardedWalletFactory.sol`: Missing zero checks in `mintWallet()`

There are no pre-conditions checks to ensure essential parameters are not `address(0)`. This may lead to incorrect or ownerless wallets.

Recommendation

Consider adding zero checks to parameters where appropriate.

Update

Resolved

3. BondingCurve3.sol: Bonding curve potentially vulnerable to flashloans if used with incorrect parameters

Depending on the parameters used, the bonding curve can potentially be griefable via flash loans. The attack vector arises from a smart contract written to be triggered to buy/sell shards. In that case, an attacker could take out a flashloan to buy a large number of shards, shifting the curve, deposit in the vault contract to make the platform/protocol trigger a buy, then sell shards back into the curve and take the profit generated by scalping the vault contract.

Recommendation

Consider including a check that disallows buys and sells in the same block by the same `tx.origin`.

Notes

4. BondingCurve.sol: Tautologies in pre-condition checks supplyShards() and supplyEther()

This function permon the following checks:

```
require(
_x.sub(shardAmount).sub(_shardSuppliers._totalSuppliedShardsPlusFeesToSuppliers) >= 0
);

require(
      (_k.div(_x)).sub(address(this).balance) >= 0
);
```

In both cases, the operations are performed on unit datatypes, which by default are `>= 0`. Since the use of safeMath ensures the desired effect of preventing underflows, the checks are correct but confusing.

Recommendation

Perform traditional underflow checks for code readability.

5. `ShardeWallet.sol`: Ignored return value of external call in `moduleExecute()`

The return value of `Address.functionCallWithValue()` is ignored.

Recommendation

Consider forwarding the return value.

6. `ModuleBase.sol`: Unused modifier

The modifier `onlyAuthorize()` is never used. Instead each module implements its own version of access control.

Recommendation

Consider removing the unused modifier.

7. `ActionModule.sol`: Consider declaring functions `schedule()` & `execute()` as `external` instead of `public`

Since functions `schedule()` & `execute()` can potentially take large arrays as arguments, declaring them as `external` instead of `public` could save a significant amount of gas. This would allow the array arguments to be directly read from `calldata` instead of being copied to `memory` first.

8. Consider providing `require` error messages

Consider providing `require` error messages for a better smart contract user experience when a transaction reverts.

Disclaimer