GasBad Equivalence Verification Report

Version 1.0

Lokapal

February 17, 2025

GasBad Equivalence Verification Report

Lokapal

February 17, 2025

Prepared by:

- Lokapal

Lead Auditor:

- Ricardo Pintos

Table of Contents

Protocol Summary

We are building a “gas bad” marketplace. An NFT marketplace, but we are going for a gas optimized version.

To do this, we are writing 2 types of smart contracts:

1. Reference contracts in solidity
2. Optimized contracts in solidity / assembly

We will be deploying `GasBadNftMarketplace.sol` to the Ethereum mainnet, but are using `NftMarketplace.sol` as a reference point.

Disclaimer

The LOKAPAL team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L

	Impact			
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

- In Scope:

```
- src
  # GasBadNftMarketplace.sol
  # NftMarketplace.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to:
  - Ethereum
- Tokens:
  - None

Roles

- Buyer: Someone who buys an NFT from the marketplace
- Seller: Someone who sells and NFT on the marketplace

Executive Summary

This security review was conducted as part of Cyfrin Updraft's Formal Verification Security course.

Known Issues

- The seller can front-run a bought NFT and cancel the listing
- The seller can front-run a bought NFT and update the listing
- We should emit an event for withdrawing proceeds
- There are MEV/Front Running issues all over the place

Formal Verification Equivalence Report

Objective

The main objective of this report is to formally verify that the Solidity and Solidity/Assembly implementations have the same functionality.

Methodology

We use the Certora Prover to run the formal verification. In this protocol, Certora works with these files:

- `GasBadNft.spec`: The Certora Prover verifies that a smart contract satisfies a set of rules written in a language called Certora Verification Language (CVL) using a `Spec` file. The syntax of CVL is similar to Solidity, but CVL contains additional features that are useful for writing specifications.
- `GasBadNft.conf`: `Conf` files are an alternative way for setting arguments for the `certoraRun` tool. In terms of functionality using `conf` files is identical to the use of the CLI Options. Instead of calling `certoraRun` with a list of shell flags, some or all the flags can be stored in a JSON file.

Contracts Analyzed

- NftMarketplace.sol: This contract allows users to list NFTs for sale.
- GasBadNftMarketplace.sol: This is the gas optimized version of the `NftMarketplace.sol` contract

Properties Verified

- `anytime_mapping_updated_emit_event`: This invariant checks if the amount of events emitted is always higher than the amount of times the `s_listings` mapping is updated.

- `calling_any_function_should_result_in_each_contract_having_the_same_state`: This rule is the core property of this report. It verifies whether calling a function in one contract returns the same output as calling the same function in the other contract.

Verification Results

This is the **Certora Job Report**. The following are the results for each property:

- `anytime_mapping_updated_emit_event`: The invariant holds true after running the Prover.
- `calling_any_function_should_result_in_each_contract_having_the_same_state`: The assertions that compare different functions failed the sanity check in this rule. However, the assertions comparing the same functions in both contracts passed, which was the intended goal of this report.

Edge Cases and Limitations

The following are the assumptions implemented for the report.

`DISPATCHER(true)`

Some functions implement the `safeTransferFrom` call with an unknown contract. So, the Prover havocs all the storage variables, disrupting the invariant. We need to set the `safeTransferFrom` as a **summary** method adding the `DISPATCHER(true)` flag.

Certora Docs: If a function with a `DISPATCHER` summary is called, the Prover will assume that the receiver of the call is one of the known contract implementations containing the given signature; the call will then behave the same way that a normal method call on the receiver would. The Prover will consider examples with every possible implementing contract, but multiple `DISPATCHER` method calls on the same receiver address in the same example will use the same receiver contract.

`"optimistic_loop": true`

We want to avoid any loops that could disrupt the Prover. So, we need to set the `"optimistic_loop"` option to true.

Certora Docs: The Certora Prover unrolls loops - if the loop should be executed three times, it will copy the code inside the loop three times. After we finish the loop's iterations, we add an assertion to verify we have actually finished running the loop. For example, in a while (a < b) loop, after the loop's unrolling, we add assert a >= b. We call this assertion the loop unwind condition. This option changes the assertions of the loop unwind condition to requirements (in the case above require a >= b). That means, we ignore all the cases where the loop unwind condition does not hold, instead of considering them as a failure.

`envfree`

We don't need to try breaking environment variables in the `getListing` and `getProceeds` methods. So, we can add the `envfree` tag on their declarations.

Certora Docs: Following the returns clause of an exact methods entry is an optional `envfree` tag. Marking a method with `envfree` has two effects. First, calls to the method from CVL do not need to explicitly pass an environment value as the first argument. Second, the Prover will verify that the method implementation in the contract being verified does not depend on any of the environment variables.

`"optimistic_fallback": true`

The `withdrawProceeds` function makes an external call. The Prover does not know whether a fallback function could modify any state variables, so it havocs them. We need to set the `optimistic_fallback` option as true.

Certora Docs: This option determines whether to optimistically assume unresolved external calls with an empty input buffer (length 0) cannot make arbitrary changes to all states. By default, unresolved external calls with an empty input buffer will havoc all the storage state of external contracts. When –optimistic_fallback is enabled, the call will either execute the fallback function in the specified contract, revert, or execute a transfer. It will not havoc any state.

`"rule_sanity": "basic"`

We could run the Prover without sanity checks. But that returns a job report that approves every function combination. This defeats the objective of equivalence testing. Additionally, with the `basic` flag the Prover will only show as `Verified` the checks that test the same functions.

Certora Docs: This option enables sanity checking for rules. The `--rule_sanity` option may be followed by one of `none`, `basic`, or `advanced`. With `--rule_sanity basic` or just `--rule_sanity` without a mode, the vacuity check and the trivial invariant check are performed.

Conclusion

After running the Certora Prover, we confirmed that the contracts are functionally identical. The Assembly implementation for gas optimization did not impact the original contract methods' behavior.

Disclaimer: This report is not a substitute for a full audit addressing broader security concerns. Any future modifications to these contracts will require a new gas optimization audit.