CERTIK

CertiK Audit Report for Most

Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Most (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that projects are checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and assessments to each project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles

such as Band Protocol and Tellor. CertiK customizes its engineering tool kits, while applying cutting-edge research on smart contracts, for each client on its project to offer a high quality deliverable. For more information: https://certik.io.

Executive Summary

This report has been prepared for **Most** to discover issues and vulnerabilities in the source code of their **MostERC20** smart contracts in scope. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Testing Summary

SECURITY LEVEL

Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Most.

This audit was conducted to discover issues and vulnerabilities in the source code of MostERC20 contracts.

TYPE	Smart Contracts & Tokens
SOURCE CODE	https://most-dev@github.com/MostProtocol/most-core
PLATFORM	EVM
LANGUAGE	Solidity
REQUEST DATE	August 5, 2020
FINAL DELIVERY DATE	August 11, 2020
METHODS	A comprehensive examination has been performed using Dynamic Analysis, Static Analysis, and Manual Review.

Review Notes

Introduction

CertiK team was contracted by the Most team to audit the design and implementations of their MostERC20 and related smart contracts. The audited files and sha256 checksum are:

- contracts/MostERC20.sol
  `b3f84e9711bc0a0f6597b21fc51818252991730603fffaf28a6521aee35b4af1`
- contracts/interfaces/IERC20.sol
  `2b63f199f838028184efefbcfd6cf2b9192624c3dae5dc1116ecbb15c36a67e8`
- contracts/interfaces/IMostERC20.sol
  `8986218d5837402d32ab01334e26a11694ef60614f0717e66ec433eacc6d5349`
- contracts/libraries/SafeMath.sol
  `e935ba680ae9f3e9c21c7c567ce14aa84f3827a45fd4b42f93b36afc03587be7`
- contracts/libraries/UniswapV2Library.sol
  `4f83e9334f833568fa47b36e9ceca435f6c2962760a0596b043c4e538d0fd9f2`
- contracts/libraries/UniswapV2OracleLibrary.sol
  `508b544096f9d916bef314a517a1f3573243272f34d8c4fbacbc7f256523e74f`

Located in

- https://most-dev@github.com/MostProtocol/most-core

The goal of this audit was to review the Solidity implementation for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

Documentation

The sources of truth regarding the operation of the contracts in scope are **something we would advise to be expanded**. To help aid our understanding of each contract's functionality we referred to in-line comments and naming conventions as well as the relevant markdown documentation.

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Most team or reported an issue.

Summary

The codebase of the project, especially with regards to the MostERC20 and related contracts, attempts to fulfill a use case that is intricate and ambitious and as such, **inefficiencies in both the design and implementation** of the various contracts were identified and properly documented.

While **most of the issues pinpointed were of negligible importance** and mostly referred to coding standards and inefficiencies, **minor, medium flaws** were identified that should be remediated as soon as possible to ensure the contracts of the Most team are of the highest standard and quality.

These inefficiencies and flaws were swiftly dealt with by the development team behind the Most project. We will create and maintain a direct communication channel between us and the Most team to aid in amending the issues identified in the report.

Recommendations

With regards to the codebase, the main recommendation we can make is **the expansion of the documentation to address the functionalities of the contracts** from an external perspective rather than an on-code perspective. Additionally, we advise that all our findings are carefully considered and assimilated in the codebase of the project to ensure the highest code standard is achieved.

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report, enforce linters and / or coding styles as well as correct any spelling errors and mistakes that appear throughout the code **to achieve a high standard of code quality and security**.

Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
Know Solidity compiler issues with 0.6.6	Solidity Issues	Informational	*

Description:

- MissingEscapingInFormatting
  - Not Applicable. This bug only matters when ABIEncoderV2 is enabled
  - Bug is fixed in 0.6.8
- ArraySliceDynamicallyEncodedBaseType
  - Not Applicable. This bug only matters when ABIEncoderV2 is enabled.
  - Bug is fixed in 0.6.8
- ImplicitConstructorCallvalueCheck
  - Not Applicable. This bug only matters when the creation code of a contract that does not define a constructor but has a base that does define a constructor.
  - Bug is fixed in 0.6.8

Recommendation:

No need to change compiler version with current version of code, since the known issues do not affect the security/performance.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
Magic number in variable `tokenBRemaining`	Coding Style	Discussion	MostERC20.sol: L141, L143

Description:

In `rebase()`, variable `tokenBRemaining` was assigned the value of `10 ** uint(IERC20(tokenN).decimals() - 2);`

Recommendations:

Add comments or documentations to explain usage of the magic numbers.

Client Response:

The `-2` is to show concept of `price >= 1.06 * amountB * decimalsB` and `price <= 0.96 * amountB * decimalsB`. Since `106` is defined as the upper bound and `96` is defined as the lower bound, the `2` is a coefficient to get `1.06` and `0.96`.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
Magic number in variable `supplyDelta`	Coding Style	Discussion	MostERC20.sol: L155

Description:

In `rebase()`, variable `supplyDelta` was re-calculated as `supplyDelta = supplyDelta / 10;`

Recommendations:

Add comments or documentations to explain usage of the magic numbers.

Client Response:

The range of supplyDelta is `[-100%, 100%]`, the division of 10 here is a lag coefficient, to adjust the range to be `[-10%, 10%]`.

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION
Overflow desired for variable `timeElapsed`	Coding Style	Informational	MostERC20.sol: L121

Description:

Logically `blockTimestamp - blockTimestampLast` should never be negative unless the function is not called in around 130 years, since the time is taken mod `2 ** 32`. There are also comments of “overflow is desired” and “ensure that at least one full period has passed since the last update”. The two comments.

Recommendations:

Using SafeMath to handle the subtraction or changing the comments to be something like “underflow would not happen in 130 years”, etc.

Client Response:

Agreed that the timestamp would not underflow. For the comments, they were specifying the behavior of the following `require` statement. So if underflow is not possible to happen, it would be good.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION
Address and value checks	Implementation	Informational	MostERC20.sol: L103, L108

Description:

In `transfer()` and `transferFrom()`, the addresses of `from` and `to` are not checked zero addresses, and `value` is not checked to be valid.

Recommendations:

Use `require` statement to check that `from != to`, `to != address(0)`, `from != address(0)` and `value` is valid.

Client Response:

No need to check addresses and values. The code is referring to https://github.com/Uniswap/uniswap-v2-core/blob/master/contracts/UniswapV2ERC20.sol. In the meanwhile, the latest OpenZeppelin ERC20 contracts do not check address and value either. Here `from == address(0)` and `to == address(0)` is expected and should not be prohibited. OpenZeppelin reference: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

Exhibit 6

TITLE	TYPE	SEVERITY	LOCATION
Balance getting non-transferable	Business Model	Minor	MostERC20.sol: L116~L207

Description:

`Gon` balance getting non-transferrable, when over time the `supplyDelta` happens to be negative, then because of `rebase`, the amount of `totalSupply` would decrease correspondingly. Thus `gonsPerFragment` and `gonValue` in `transfer()` would increase, which means 1 `token` equals to more `gons` now, compared to the time when the contract is initialized. It would be possible that for example there is a user who had 1 `token` = 10 `gons` at the beginning, but now since 1 `token` = 20 `gons`, and the user cannot transfer the 1 `token` since the user does not have enough `gons`.

Client Response:

This is expected. When `totalSupply` decreases, the `balance` of the user is also decreased. That's to say if the user had 1 `token` at the beginning, then according to CertiK's example, when `totalSupply` is halved, the balance of the user would be changed to be 0.5 `token`. Such that the attempt of transferring 1 `token` is not possible. Also the logic here is referring to https://github.com/ampleforth/uFragments/blob/master/contracts/UFragments.sol.

Exhibit 7

TITLE	TYPE	SEVERITY	LOCATION
Mismatch output types	Coding Style	Discussion	MostERC20.sol 191, 194

Description:

The return value `amountOut` of function `consult()` is uint. On lines 191 and 194 there are assignments of the output value of `decode144` to `amountOut`, but the output of `decode144` is of type `uint144`.

Client Response:

Casting from `uint144` to `uint256` is automatic, just like converting `uint8` to `uint256`.

Exhibit 8

TITLE	TYPE	SEVERITY	LOCATION
Uninitialized variable	Implementation	Informational	MostERC20.sol: L49

Description:

Variable `blockTimestampLast` is not initialized.

Recommendations:

Recommend to initialize it to be `now`.

Client Response:

The contract would call function `initialize()` to perform the initialization during 24 hours, as the first calling window for `rebase()`, and `blockTimestampLast` is initialized in L78. If there is the case that `initialize()` is not correctly called, the direct call of `rebase()` would not affect the logic, since the `pair` in L120 would be `0`, which would fail the `rebase`.

Exhibit 9

TITLE	TYPE	SEVERITY	LOCATION
Unnecessary repeated computation	Gas optimization	Discussion	MostERC20.sol: L144, L146, L148

Description:

The value `10**(tokenBDecimals - 2)` is used three times on lines 144, 146, 148.

Recommendations:

Precompute this value and store it to a memory variable so no need to perform this exponentiation three times.

Client Response:

Fixed in commit `aaf804a`, the 08/09/2020 update.

Exhibit 10

TITLE	TYPE	SEVERITY	LOCATION
Return value	Gas optimization	Discussion	MostERC20.sol: L151

Description:

In the `else` case on line 151 we can directly return `totalSupply`, emit the event and exit the function because it is the only case where `supplyDelta == 0` is satisfied. Hence we can save a bit of gas by not performing division on line 155.

Client Response:

For code consistency, since this one is not affecting contract security, will not change this part.

Exhibit 11

TITLE	TYPE	SEVERITY	LOCATION
Unintuitive price adjustment	Logics	Discussion	MostERC20.sol: L161-165

Description:

`priceAverage` is the price of `token0` in terms of `token1`. If `priceAverage > UPPER_BOUND * 10 ** (tokenBDecimals - 2)` then the `priceAverage` is too high and needs adjustments, but in this case `supplyDelta` is negative and line 162 `totalSupply = totalSupply.sub(uint256(supplyDelta.abs()))` would decrease the supply of token0 and push the `priceAverage` even higher.

Client Response:

After `rebase()` succeeds, if we call oracle immediately, indeed `priceAverage` would change significantly. However, since we can only call `rebase()` every 24 hours, and the average price provided by oracle is the average price within 24 hours, the final `priceAverage` for the next call of `rebase()` would only be affected by transactions happening in the market. It could be either increased or decreased. Reference:
https://github.com/ampleforth/uFragments/blob/master/contracts/UFragments.sol
https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/examples/ExampleOracleSimple.sol

Exhibit 12

TITLE	TYPE	SEVERITY	LOCATION
Token relations	Logics	Discussion	MostERC20.sol: L70 MostHelper.sol: L30

Description:

In `MostHelper.sol` there are two `MostToken` contracts `mostTokenA` and `mostTokenB`. each of which contains two other contract addresses `token0, token1.` It is assumed there are only two tokens, where each of mostToken saves its own address in `token0` and the address of the other token in `token1` as reflected in `MostERC20.sol` line 68. In that case we should check that the token pairs in `mostTokenA` and `mostTokenB` correspond to each other.

Client Response:

This confusion is because of our abuse of `IMostERC20`. `tokenB` is always a normal token, and `tokenA` is always `MostToken`. We wrapped here since we need to use decimals of `tokenB` in the following part. We found there is an existing one in `IMostERC20`, so we used it directly. Now we have the code changed back to `IERC20`.

In conclusion, there are only two tokens, `tokenA` and `tokenB`. If `tokenA` is changed to be `mostToken`, then `token0` and `token1` means `tokenA` and `tokenB`. We have also noticed that we can directly use `tokenB` instead of passing it into the contract.

Fixed in commit `aaf804a`, the 08/09/2020 update.