Marble
Smart Contracts

Mikhail Vladimirov and Dmitry Khovratovich

18th March 2019

Part II

This document describes the audit process of the Marble smart contracts performed by ABDK Consulting.

1. Introduction

We've been asked to review the Marble smart contract given in a private access to the Marble repository at the following commit. We were also provided with business requirements by email.

1.2 Uniswap Issues

We were asked to audit the contracts assuming that the Uniswap price oracle is secure and correct. We have identified the following assumptions that need to be satisfied so that Uniswap price makes sense:

1. The ETH/Token reserve ratio is a good approximation to the price at other exchanges and can be used to estimate how much collateral can be sold for in the Marble lending contract.
2. The reserve amounts can not be 0 nor too high (2^64 or higher) otherwise overflow, underflow, or zero division might happen and break the availability.
3. The reserve ratio is not volatile.
4. Someone regularly updates the price oracle.

1.3 Median Issues

We were also asked to check the security of the approach that takes the median of medians of pokes as a current price. Besides the issues with the heap (see below), we have identified the following problems with the Marble approach:

1. Once a malicious price gets into 8 checkpoints, it will take at least 50 minutes to get it out.
2. If updates are not frequent, checkpoints will be medians of 1-2 values only.

3. The current price will get into the contract with at least 50 minutes delay.

Given these, we do not recommend using the median approach for high-volatile and low-volume trades, as the stored price might be very far from the actual one if collateral is to be traded.

2. CheckpointHeap

In this section we describe issues related to the smart contract defined in the CheckpointHeap.sol . The main problem is that the data structure of the `CheckpointHeap` library looks too complicated for the task it solves. The median for unsorted array of up to 14 elements may be easily calculated via quickselect, bubblesort algorithms or by comparator networks, and this will be much cheaper than maintaining min-max heap in storage.

2.1 Critical Flaws

This section lists critical flaw, which was found in the smart contract.

Line 266: malicious user may cause overflow at all multiply operations in the line by submitting very large ETH and/or token reserve values, thus preventing other users from submitting correct pokes.

2.2 Arithmetic Overflow Issues

This section lists issues of the smart contract related to the arithmetic overflows.

1. Line 270,274: overflow is possible in multiplication and addition operations.
2. Line 278: the underflow is possible in this line.

2.3 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 17: interlacing min and max heap instead of storing them one after another will make this unnecessary.
2. Line 20,21: using fixed-size array of length 256 will save storage space.
3. Line 25: the `heapIndex` is used only in `insertAtindex` to know heap index of checkpoint with certain index in queue. Actually, this method is only used to replace the latest checkpoint added to the queue, so it would be enough to store index of tail checkpoint only, rather than indexes of all checkpoints.
4. Line 28: there is no reason to store block timestamp for each checkpoint, it would be enough to store timestamp for the first and the last checkpoints.
5. Line 34: the comment is incorrect. Correct condition would be `(maxQueueSize + 1) & (maxQueueSize + 2) == 0`.

6. Line 47: adding first checkpoint into max heap instead of min heap will make this assignment cheaper, because new value will be zero.
7. Line 54: the value `data.tail` is read from storage multiple times. Consider reading once and caching in memory.
8. Line 61: the `newCheckpointIsGreaterThanMinHeapCheckpoint` value is always calculated, but is used only if newCheckpointIsLessThanMaxHeapCheckpoint is false. Consider calculating only when needed.
9. Line 187,230: the code in this lines is very similar to `minHeapifyUp` and `minHeapifyDown` defined above. Consider merging into one method with extra parameters.
10. Line 254,265,269,273,277: functions `swap`, `lessThan`, `left`, `right` and `parent` should be private.

3. OracleToken

In this section we describe issues related to the smart contract defined in the OracleToken.sol.

3.1 Unclear Behaviour

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

Line 11: the `token` variable is never used. Perhaps, there is no need to keep it in the code.

3.2 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

Line 16-18: it would be cheaper to make `name` a compile-time constant rather that storage variable.

3.3 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

Line 30: probably, there should be `redeem` instead of `redeam`.

4. OracleSubscribers

In this section we describe issues related to the smart contract defined in the OracleSubscribers.sol

4.1 Readability Issues

This section lists cases where the code is correct, but too involved and/or complicated to verify or analyze.

> Line 21: `365 days` would be more readable instead of `365 * 24 * 60 * 60`.

4.2 Critical Flaws

This section lists critical flaws, which was found in the smart contract.

1. Line 33: it would be more fair to collect before increasing balance. Consider situation:
    - Alice pays fee for two months, but do not use oracle.
    - 1 year passes.
    - Alice wants to use oracle and to be able to do so pays fee for another two months.
    - Smart contracts collect fee for 4 months, and still does not allow Alice to use oracle.
2. Line 45: the `sub` decreases user's balance, but does not send ether back to the user.

4.3 Major Flaws

This section lists major flaws, which was found in the smart contract.

1. Line 32: the return value from `collect` is ignored.
2. Line 58: in case `owedAmount` exactly equals current balance, the whole balance will be collected. The payment timestamp will not be reset, so next time user will try to resupply his balance, fee will be charged.

4.4 Unclear Behaviour

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

1. Line 11: time in Ethereum is measured in seconds. Fee per second is more appropriate.
2. Line 44: the second argument after `min` is missing.
3. Line 62: the assignment `= 0` will make `getOwedAmount` to return 0 next time. Is it OK?

4.5 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 37: the `SUBSCRIPTION_FEE.div(12)` should be compile-time constant.
2. Line 58: the value `balances[asset][subscriber]` is read multiple times, consider reading once and caching in memory.
3. Line 49: the rest of the `collect` method is indented as if it is always executed, while this is not the case. Consider putting the rest of the method into else{...} branch.

4.6 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

Line 11: the Ether price could raise or drop several times, making this fee absurdly small or prohibitanly large. Consider making this variable, or paying fee in tokens whose fair price will be discovered on exchanges.

5. UniswapPriceOracle

In this section we describe issues related to the token defined in the UniswapPriceOracle.sol

5.1 Critical Flaws

This section lists critical flaws, which was found in the smart contract.

Line 102: the `address payable exchange = factory.getExchange(token)` operation retrieves the balance of the exchange contract in Ether and tokens. Depending on the `IUniswapExchange` implementation, both values can be

manipulated by a malicious trader more or less easily. This may result in a manipulated checkpoint value. Also, an external audit of `UniSwap.exchange` implementation might be needed.

5.2 Arithmetic Overflow Issues

This section lists issues of the smart contract related to the arithmetic overflows.

1. Line 72,75,78,81: the multiplication will throw exception in case of overflow.
2. Line 236,237: overflow in `mul` operation will revert transaction.
3. Line 239: the function `wdiv` internally multiplies first argument by `10^18`. Assuming that token has 18 decimals, `side1` and `side2` will have 36 decimals. Multiplying by 10^18 will produce number with 54 decimals. It leaves only about 23 decimals till 2^256. It makes overflow very probable when working with low-cost tokens, or tokens with large number of decimals.

5.3 Unclear Behaviour

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

Line 72: the division will throw for zero reserve amount and zero source amount. Is it OK?

5.4 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

Line 29,142: probably, there should be `OracleToken` instead of `address`.

1. Line 50: the method `getPrices` does not modify blockchain state and should probably be declared as `view`.
2. Line 66: the `isHuman()` is calculated twice. Consider putting both `require` statements into `if (!isHuman()) {...}` block.
3. Line 78: rounding error that occurs here is then multiplied by destination token reserve, which could lead to precision degradation. Consider refactoring formula like this: `destAmount = srcAmount * srcETHReserve " destTokenReserve / (srcTokenReserve * destETHReserve)`, so the division will happen only once at the very end of calculation.
4. Line 98: the actual condition compares block timestamps assuming two blocks may not have the same timestamp. While this assumption is probably true for now, more straightforward way would be to compare block numbers.
5. Line 166: the function `isHiman` prohibits usage of multisig wallets, which are common for handling big amounts of asset.
6. Line 186: the expression `exchange != address(0)` is always true.

7. Line 239: if `side1 == side2==0`, this operation will throw and the checkpoint won't be added. This may affect the price availability and freshness.

5.5 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

1. Line 25: according to instruction, this should be 14 blocks, not 3.5 minutes.
2. Line 81,136: consider align the `address` to IERC20 requirements.
3. Line 116: it would be more secure to use `<` instead `of + 1 !=`.
4. Line 124: according to instruction poke becomes checkpoint only after 14 blocks, but according to the code, current incomplete checkpoint is also taken into account.
5. Line 127: according to instruction, checkpoint is created when there were more than 1 poke within 14 blocks, but according to the code, it is enough to have at least one poke.

6. Our Recommendations

Based on our findings, we recommend the following:

1. Fix critical issues which could lead to asset loss.
2. Pay attention to major issues.
3. Fix arithmetic overflow issues.
4. Check issues marked “unclear behavior” against functional requirements.
5. Refactor the code to remove suboptimal parts.
6. Fix the readability and other (minor) issues.