ABDK

Marble Smart Contracts

Mikhail Vladimirov and Dmitry Khovratovich

4th 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. We also consulted the documentation on the Marble web site.

2. ISecuredLendingProtocol

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

> Line 32: usage of the variable `balances` is incorrect because interfaces usually do not have storage variables defined.

3. IDeployer

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

> Line 6: it is unclear from the function name `deploy()` what it actually does. Documentation comment would be helpful.

4. IBorrower.sol

In this section we describe issues related to the smart contract defined in the IBorrower.col.

> Line 8: interfaces usually do not implement any functions.

5. IWeth

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

Line 4: the contract `IWeth` looks very similar to ERC20 token, but adds method deposit and withdraw. Probably, it should inherit from IERC20.

6. IUniSwapExchange

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

Lines 38, 39, 40: interfaces usually do not declare any state variables.

7. IUniSwapFactory

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

Line 11: there should be `IUniswapExchange` instead of `address`.

7. ISLPGovernance

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

Line 11: interfaces usually do not declare any state variables.

8. GlobalDeployer

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

8.1 Suboptimal Code

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

1. Line 13: the class `GlobalDeployer` class looks too complicated for the problem it solves. Actually, deployment for multiple interlinked contracts may be performed without auxiliary contracts at all by establishing links in the right order. Note that the contract addresses can be computed before deployment as they are deterministic functions of the deployer's address and the number of already deployed contracts from this address. This can be used to set all

external addresses in constructors even though some contracts have not been deployed yet.

8.2 Major Issues

1. Line 31,42: method `deployGovernance, deploySLP` may be called several times, probably producing unexpected results. Consider adding protection from calling another time.
2. Line 36,38,39: the returned value is ignored, unsuccessful call will be treated as successful and transaction will not be reverted.

8.3 Other Issues

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

1. Line 32: the method is named `deployGovernance`, but actually it deploys Marble smart contract as well. This is confusing. Consider renaming method or moving Marble deployment into separate method.
2. Line 36, 38, 39, 49, 50, 51, 52, 53, 56: usage of simple method call instead of abilities would make code easier to read and less error prone.
3. Line 49: the method `slpGovernance` assumes that SLP governance is already deployed, consider adding explicit check for this.

9. SLPInsuranceManageDeployer

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

Line 9: perhaps, contract `SLPInsuranceManagerDeployer` complicates the deployment schema..

10. SLPGovernanceDeployer

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

10.1 Suboptimal Code

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

1. Line 18: the command `assets = _assets;` adds the arbitrary number of addresses into storage. Writing large number of addresses could exceed block gas limit and make contract impossible to deploy. Consider implementing a split of list of assets into smart contract among several transactions.

11. SLPDeployer

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

Line 9: perhaps, the contract `SLPDeployer` complicates the deployment schema.

12. MarbleDeployer

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

Line 9: perhaps, the contract `MarbleDeployer` complicates the deployment schema.

13. MakerOracle

In this section we describe issues related to the token defined in the MakerOracle.com.

13.1 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.

1. Line 10,15: functions signature `compute` and `MakerOracle` do not explain what they actually do.
2. Line 30: the second part of returned value is ignored. Is it OK?

13.2 Suboptimal Code

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

1. Line 9: most interfaces in this project are declared in separate files whose names are the same as the names of corresponding interfaces, but this interface shares file with another smart contract. This looks inconsistent and makes code harder to read.

13.3 Major Issues

1. Line 34: there should be `div(decimals)` instead `mul(decimals)`.
2. Line 34: the div `srcAmount` perhaps a bug. Bigger source amount leads to smaller destination amount.

13.4 Other Issues

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

3. Line 18,23: there should be `IMakerOracle`, `IMakerOracle` instead of `address`.
4. Line 21: in ERC20 “decimals” refers to the number of decimals, but here it refers to `10 ^` number of decimals. Consider renaming.

14. Marble

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

14.1 Major Flaws

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

1. Line 60: malicious actor may set his own address as governance quickly after smart contract is deployed and then take ether, deposited by contract creator.
2. Line 65: the value of the `k` may drift due to rounding errors and probably may be manipulated.

14.2 Suboptimal Code

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

1. Line 17: the `TOKEN` looks like synonym for `this`, but is more expensive to access.
2. Line 23: the function will be called not only when there is no data in transaction, but also when method signature is not recognized. If purpose of this function is to receive plain payments, then it would probably be better to check that `msg.data.length` is zero.
3. Line 31: minting tokens at token contract's address and then immediately transferring them to contract owner's address is effectively the same as minting tokens at contract owner's address, but is more expensive.
4. Line 31: add in this line a check `premine<totalSupply` (for clarity).
5. Line 36: the variable `governance` is used without explicit assignment.
6. Line 37: once zero governance address means that governance was not set yet, it should probably be forbidden to set it to zero address.
7. Line 78: `ETHER` and `TOKEN` should be enum constants, not addresses.
8. Line 78: The equality ‘`src == address(weth)`’ is always false.

14.3 Arithmetic Overflow Issues

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

1. Line 74: overflow is possible in multiplication and addition.
2. Line 74: in this line division by zero is possible.

14.4 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 30: the `TOKEN` gives all minted tokens to token contract itself. Does it make sense for token contract to own its own tokens, as long as token contract may anyway issue more tokens at any time?
2. Line 35: the function `setGovernance` may be called by anyone. It is unclear to say should it be like this or not.
3. Line 59: perhaps, there is no need to declare the function `payout` as external.

14.5 Other Issues

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

1. Line 27: there should be `IWeth` instead of `address`.
2. Line 35: there should be `ISLPGovernance` instead of `address`.
3. Line 80: the condition `else` may never happen.
4. Line 24: the `buy()` does not fit into 2300 gas. The fallback function does not follow recommendation in Solidity documentation: “In the worst case, the fallback function can only rely on 2300 gas being available (for example when send or transfer is used), leaving little room to perform other operations except basic logging”.

15. SLPPosition

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

1. Line 10: the SPLPosition smart contract looks almost identical to SLPLoan smart contract. The only different is an error message in method `mint`. Probably, it would be better to put all common parts into base contract and inherit both, SLPLoan and SLPPosition from it, or even use separate instances of the same contract to represent SLP loan and position tokens.

16. SLPScriptCaller

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

1. Line 15: the function execute always return true, so returned value does not make any sense. Probably it should just return nothing.

17. SLPGovernance

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

17.1 Moderate Issues

This section lists moderate flaws, which were found in the smart contract.

1. Line 51,59, 67,79,91: methods `addAsset`, `removeAsset`,`setCollateralRatio`, `setInsuranceFee` and `setLiquidationDiscount` may be called by everyone, but actual behavior depends on who calls it. This looks odd and confusing. Consider splitting into two methods: one callable by everyone that proposes adding asset, and another callable only by this contract, that actually adds asset.

17.2 Readability Issues

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

1. Line 98: notation `1e18` (instead `10**18`) would probably be more readable and consistent with other similar constants in this smart contract.

17.3 Suboptimal Code

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

1. Line 17: having in the same contract governance voting logic and functionality being governed makes this contract overcomplicated and less flexible. Having both, multi-owner vote-base Governance and single-owner Ownable logic in the same contract looks redundant. Consider implementing simple Ownable contract implementing SLP-specific governance logic, and separate Governance contract serving as an owner of the first contract. This way it would be possible to change voting logic in the future by deploying another version of multi-owner Governance contract and transfer owner of the first smart contract to it.

2. Line 21: as long as version IDs are assigned sequentially, it would probably be more effective to use array instead of mapping in this line.
3. Line 26: the using decimal fixed point math is less effective than binary fixed point math. WAD uses 10^18 as fixed denominator which is more expensive to multiply and divide by than, say, 2^64.
4. Line 37: the number of assets is effectively limited by block gas limit. Consider implementing an ability to split uploading list of assets among several transactions.
5. Line 42: the state variable is used before assignment. In that case, it will be always zero here.
6. Line 105,110: as long as zero SLP address means that SLP address is not set yet, it would be better to forbid setting it to zero.
7. Line 130: state variable `slpInsuranceManager` already has type `IInsuranceManager`, so these type casts are redundant.
8. Line 130,138,163,172: in case, when SLP insurance manager, token and SLP are not set, it would be better to handle this case explicitly, e.g. to throw.

17.4 Other Issues

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

1. Line 19: the meaning of the mapping `assets` is unclear. Documentation comment would be helpful.
2. Line 37: there should be `IPriceOracle` instead of `uint` and `IConverter` instead of `address`.
3. Line 45, 46, 47: probably, the `1.5e18`, `.025e18` and `.9e18` should be constructor parameters.
4. Line 46: there should be `2.5%` instead of `0.25%`.
5. Line 60: the owner can’t add assets. Perhaps, it should be fixed.
6. Line 74: perhaps, instead > in this line should be >=. It makes possible to set collateral ration to `MIN_COLLATERAL_RATION`.
7. Line 94: perhaps, the `liquidationDiscount` shouldn’t be required to be non-zero.
8. Line 103: there should be `ISecuredLendingProtocol` instead of `address`.

18. SLPLoan

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

18.1 Readability Issues

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

1. Line 14: the name of the parameter `_slp` is prefixed with underscore ('_') in contrast with all other parameter names in the same smart contract. This inconsistency makes code harder to read.

18.2 Suboptimal Code

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

1. Line 20,33: explicit reference to `super` is redundant, because this smart contract does not override `_mint` and `_checkOnERC721Receiver` method.
2. Line 33: the returned value is ignored, while documentation suggest that calling contract should revert in case false was returned. The documentation for `_checkOnERC721Receiver` (https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC721/ERC721.sol#L258) says: "Internal function to invoke `onERC721Received` on a target address ... return bool whether the call correctly returned the expected magic value". And for `onERC721Received` (https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC721/IERC721Receiver.sol#L11): The ERC721 smart contract calls this function on the recipient ... This function MUST return the function selector, otherwise the caller will revert the transaction". So calling contract is supposed to revert in case `_checkOnERC721Receiver` returned false.

18.3 Other Issues

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

1. Line 14: there should be `address _slp` instead of `ISecuredLendingProtocol`.

19. SLPInsuranceManager.sol

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

19.1 Major Flaws

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

1. Line 36: the function `collect` may be called by anyone, calls smart contract whose address is passed as argument, does not prevent recursive calls, updates state variables after calling contract passed as argument, and does transfer real assets. Altogether this is a very risky combination.

19.2 Critical Flaws

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

1. Line 42: the condition not supposed that returned value is ignored, so unsuccessful transfer will be counted as successful one.
2. Line 55: the returned value is ignored, so unsuccessful transfer will be counted as successful one.

19.3 Suboptimal Code

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

1. Line 32: as long as zero governance address means that governance is not set yet, it would be better to forbid setting governance to zero.
2. Line 62: it seems that `governance.getSLP()` and `governance.getInsurancePool()` always returns the same value. It would be more effective to cache it it smart contract's storage. Also situation when governance is not set yet is not covered. It would be better to add check for zero governance address.

19.4 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 30:perhaps, there is no need to set the function `setGovernance` as public.
2. Line 81: this line checking conversion rate from borrow asset to weth, while actually the main process is convert from weth to borrow asset, and spread may be arbitrary large. Perhaps, there is no need to initiate this check? Probably we should convert `mediaEthAmont` from weth to borrow asset and then compare result with `requestedAmount` taking margin into account.
3. Line 87: there is no check for conversion rate prior to conversion, so any offered rate converter is going to be accepted. Is this OK?

19.5 Other Issues

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

1. Line 26,30: there should be `IWeth` and `ISLPGovernance` instead of `address`.
2. Line 33: `address(governance)` is equal to `_governance`.
3. Line 33: the `uint(-1)` is looks confusing. Consider extracting named constant. e.g. MAX_UINT256.
4. Line 36: the `address` should be aligned to IERC20.
5. Line 52: if the `spotEthAmount` is equal to `minEthToCollect`, it will not be collected. It would to be better to use `<=`.

20. SLPConverter.sol

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

20.1 Critical Flaw

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

1. Line 44,55: the returned value `transferFrom` is not checked, so failed transfers may be processed as successful.

20.2. Moderate Flaws

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

1. Line 28: the condition in this line doesn't cover case when `src==dest`. It would be better to handle is explicitly, i.e. return `srcAmount` or throw.

20.3 Suboptimal Code

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

1. Line 35-39, 62-63: the code should be refactored. It mostly duplicates code in `(src == address(weth))` branch.
2. Line 58-60: this code should be refactored too. It duplicates `(dest == address(weth))` branch.

20.4 Other Issues

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

1. Line 17,29: instead `address` there should be `IUniswapFactory`.
2. Line 29: instead second `address` there should be `IWeth`.
3. Line 43: the `address` should be aligned to IERC20.

21. SecuredLendingProtocol.sol

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

21.1 Critical Flaws

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

1. Line 62,114,125, 135, 155, 167, 219, 220: returned value is ignored, so an unsuccessful transfer may reduce the sender balance.
2. Line 110: it seems that lender, once approved by borrower, may add arbitrary amount to position forcing borrower to pay interest for it, in case amount of collateral deposited by borrower permits this.
3. Line 114: it seems that lender, once approved by borrower, may send borrowed assets to arbitrary address (e.g. to himself), still forcing borrower to repay.

21.2 Moderate Flaws

This section lists moderate flaws, which were found in the smart contract.

1. Line 335: the function `function time()` accesses block metadata, so it is probably supposed to run inside mined transaction. Calling this function off-chain may produce unexpected result.
2. Line 132: in this case insurance fee is greater than repay amount (such situation does not look impossible), and lender's balance is not enough to pay insurance fee, the borrower will not be able to repay the loan.

21.3 Suboptimal Code

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

1. Line 28: as long as positions are assigned sequentially, it would be more effective to use dynamically-sized array instead of `mapping` here. Also, this would make “nonce” statue variable unnecessary, because current length of array could be used instead.
2. Line 36,41: variables `governance, scriptCaller, slpLoan, slpPosition` are uninitialized.
3. Line 37: the type case is redundant, `_governance` already has type `ISLPGovernance`. And as long as zero governance address means that governance is not set yet, setting `governance` address to zero should probably be forbidden.

4. Line 42: as long as zero governance address means that governance is not set yet, setting Script Caller, Slp Loan, Slp Position address to zero should probably be forbidden.
5. Line 55,60: the usage of deposit is inconsistent across the smart contract: in some cases, e.g. when money borrowed assets are transferred from lender to borrower, they are taken from lender's deposit; in other cases, e.g. when repaid assets are transferred from borrower to lender, they are taken directly from borrower via `transferFrom` method.
6. Line 57,61: the operation with `amount` is correct only for token contracts that do not charge fees in tokens.
7. Line 66: the case when `governance` is not set yet should be checked.
8. Line 72: the same method is used for checking both borrow asset and collateral asset. What if some assets are allowed only for borrowing, or only as collateral? There should probably be two separate methods.
9. Line 78,79: it seems that `lender` and `borrower` is stored twice: one time in this lines and another time in `slpLoan` and `slpPosition` token contract as the owner of corresponding token. It should be enough to store only once in either place.
10. Line 118,150: the approval in one token contract permits transferring tokens managed by another token contract should be refactored.
11. Line 134, 203: the case when the `governance` is not set yet is not provided. There should be explicit check for zero governance address.
12. Line 226: the `priveOracle` said that the price oracle is related to an external trading contract not under the control of Marble. As a result, an adversary who can manipulate the token price over some period of time, can also manipulate the liquidate values.
13. Line 243: other methods of this smart contract that need to check that position is not expired, require the following: `time() < toPosition.expirationTimestamp`, so, expiration timestamp seems to be inclusive, and check for expired position should look like this: `position.expirationTimestamp <= time()`.
14. Line 294: the code in this line is confusing, because it looks like the assignment of new value to state variable while actually the value is returned from the method. The `return` would make code more readable.
15. Line 294: using annual interest rate is subeffectve, because it requires raising numbers to fractional powers, which is expensive. Consider using 1 second interest rate, that needs raising numbers to integer powers only.
16. Line 316: the event `emit Open(positionId)` is emitted when user calls `openAndBorrow()`, but it is not emitted when user calls `open()`, then `add()`, and then `borrow()`, while effectively this leads to the same result.
17. Line 323: the case when the `governance` is not set yet is not provided. There should be explicit check for script caller address.

18. Line 331: mapping `positions` is already declared public. So Solidity compiler will generate getter methods for it. No need to declare separate public getter explicitly.
19. Line 335: perhaps, there is no need to declare the function `function time()` as public. It does not seems useful outside smart contract.

21.4 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 35,40: methods `setGovernance`,`setScriptCaller`,`setSLPLoan`, `setSLPPosition` can be called by anybody and declared as external. Is this OK?
2. Line 57: the one may deposit to other people's accounts. Is this OK?
3. Line 130: division by zero throws exception, so the function does not support zero interest rates.

21.5 Arithmetic Overflow Issues

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

1. Line 127: potential overflow here may make it impossible for borrower to repay loan, at least in one transaction. Probably borrow amount should be limited to guarantee that calculation will not lead to overflow. Another option is to implement method that multiplies number by simple fraction and does not throw in case final result fits into 235 bits, even when intermediate result does not fit.
2. Line 130: overflow in this line may make it impossible for borrower to repay load, at least in one transaction.

21.6 Other Issues

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

1. Line 20: the `positionId` should be indexed.
2. Line 57: the method `add` is probably defined in `DSMath` contract whose sources are missing.
3. Line 130: the calculation of `insuranceFee` is different in the documentation on the website.
4. Line 286: perhaps, in this line there should be `wmul` and `wdiv` instead of `mul` and `div`.

22. Our Recommendations

Based on our findings, we recommend the following:

1. Check issues marked “unclear behavior” against functional requirements.
2. Refactor the code to remove suboptimal parts.
3. Fix the readability and other (minor) issues.