Ambisafe

DeXe Platform Contracts Code Audit and Verification by Ambisafe Inc.

July 2023

Oleksii Matiiasevych

1. **Introduction.** DeXe Network requested Ambisafe to perform a code audit of the DeXe Platform contracts. The contracts in question can be identified by the following git commit hash:

   ```
   e0240228947c8664ebc01095fe9c8e8c41d6d6be
   ```

   All contracts in the repo are in scope.

   After the initial code audit, DeXe Network team applied a number of updates which can be identified by the following git commit hash:

   ```
   36657e92e562381597ed87258b75e744d89ddf18
   ```

   Additional verification was performed after that.

2. **Disclaimer.** The code audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bugfree status.

3. **Executive Summary.** There are **no** known compiler bugs for the specified compiler version (0.8.9), that might affect the contracts' logic. There were 3 critical, 3 major, 14 minor, 32 informational and optimizational findings identified in the initial version of the contracts. All the minor and above severity findings were addressed and were not found in the final version of the code, while a few of the optimizations and notes remain acknowledged in favor of code readability and design choices.

4. **Critical Bugs and Vulnerabilities.** Three critical issues were fixed over the course of the engagement. First one would allow users to drain the TokenSaleProposal of all the

funds (5.3). Other two would let users drain the TraderPool and TraderPoolRiskyProposal by repeatedly calling invest/divest functions (5.28, 5.39).

5. Initial Line By Line Review. Fixed Findings.

5.1. ERC721Power, line 102. Note, the **addCollateral()** function allows an addition of zero amount.

5.2. DistributionProposal, line 66. Optimization, the **claim()** function could excessively check own balance in case of native currency payout.

5.3. TokenSaleProposal, line 71. **Critical**, the **vestingWithdraw()** function lets users drain the contract by providing duplicate entries in the tierIds array.

5.4. TokenSaleProposal, line 118. Minor, the **buy()** function first transfers the rounded down amount of tokens to the user, then rounds down the **vestingTotalAmount** which could result in **vested + unvested < amountBought**. Consider storing **vestingTotalAmount** as **saleTokenAmount - unvested**. Or do not store **vestingTotalAmount** at all, and calculate it on the fly each time.

5.5. TokenSaleProposal, line 134. **Major**, the **recover()** function lets users corrupt the **tier.tierInfo.tierInfoView.totalSold** by providing duplicate entries in the tierIds array if more than one tier sells the same token.

5.6. TokenSaleProposal, line 173. Minor, the **getSaleTokenAmount()** function could return 0 if **amount*exchangeRate < PRECISION**. Consider validating the **saleTokenAmount** to be greater than zero instead of **exchangeRate**.

5.7. TokenSaleProposal, line 348. Optimization, the **_addToWhitelist()** function should read the **request.tierId** into a local variable once in the beginning of the function and then use the var every time.

5.8. TokenSaleProposal, line 387. Note, the **_countPrefixVestingAmount()** function could return an **amount < vestingTotalAmount** in some cases, restricting users from withdrawing part of the vested tokens after full vesting period.

5.9. GovSettings, line 29. Note, the **__GovSettings_init()** function does not verify the length of the **proposalSettings[]** which could result in not all the **ExecutorTypes** being set up.

5.10. GovSettings, line 32. Optimization, the **__GovSettings_init()** function could use **settingsId** as an increment in the for loop instead of **i**.

5.11. GovSettings, line 37. Note, the **__GovSettings_init()** function could use **_setSettings()** function instead of filling up the settings mapping directly, for consistency of **SettingsChanged** events.

5.12. GovSettings, line 45. Note, the **__GovSettings_init()** function has a special condition for the **ExecutorType.DISTRIBUTION** settings, which is not enforced in the **editSettings()** function.

5.13. GovValidators, line 118. Minor, the **createExternalProposal()** function could create a proposal with **quorum > 100%**.

5.14. GovValidators, line 240. Note, the **getProposalRequiredQuorum()** function duplicates the **_proposalExists()** code in place.

5.15. GovPool, line 173. Optimization, the **createProposal()** function reads the **latestProposalId** variable from storage multiple times.

5.16. GovPool, line 198. Optimization, the **vote()** function executes the **onlyBABTHolder()** modifier twice.

5.17. GovPool, line 318. Optimization, the **unlock()** function executes the **onlyBABTHolder()** modifier twice.

5.18. ShrinkableArray, line 31. Note, if the **crop()** function is used to increase the length of the array, then it could expose previously shrunk entries.

5.19. GovPoolCreate, line 192. Optimization, the **_handleDataForValidatorBalanceProposal()** function excessively iterates over the **data** array which always has a single entry.

5.20. GovPoolStaking, line 128. **Major**, the **_recalculateStakingState()** function allows a reentrancy if the **rewardToken** is a native currency. The reentrancy could be used to receive the same reward multiple times.

5.21. GovPoolStaking, line 148. Note, the **_getMicropoolPendingRewards()** function calculates **rewardsDeviation** with very poor precision. If it is intended then consider adding an explanation.

5.22. GovUserKeeperView, line 34. Optimization, the **votingPower()** function calls **userKeeper.tokenAddress()** on every iteration of the loop.

5.23. GovUserKeeperView, line 42. Optimization, the **nftAddress()** function calls **userKeeper.tokenAddress()** on every iteration of the loop.

5.24. UniswapV2PathFinder, line 61. Note, the **_getPathWithPrice()** function should return **withProvidedPath false** in case of zero **amount**.

5.25. TraderPoolCommission, line 100. **Major**, the **distributeCommission()** function could be self-sandwiched to force it to buy DEXE at an inflated price, resulting in a stolen commission.

5.26. TraderPoolInvest, line 118. Minor, the **investInitial()** function could mint zero LP in case there is **totalBase** already present in the pool before the invest.

5.27. TraderPoolPrice, line 45. Optimization, the **getNormalizedPoolPriceAndPositions()** function excessively copies **openPositions** into **positionTokens**. Could just use the **openPositions** everywhere.

5.28. TraderPool, line 173. **Critical**, the **divest()** function if called in the next block after the **invest()** will produce profit to the user letting them drain the pool.

5.29. TraderPoolInvestProposal, line 56. Minor, the **changeProposalRestrictions()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.30. TraderPoolInvestProposal, line 101. Minor, the **invest()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.31. TraderPoolInvestProposal, line 128. Minor, the **divest()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.32. TraderPoolInvestProposal, line 160. Minor, the **withdraw()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.33. TraderPoolInvestProposal, line 181. Minor, the **supply()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.34. TraderPoolInvestProposal, line 201. Minor, the **convertInvestedBaseToDividends()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.35. TraderPoolRiskyProposal, line 60. Minor, the **changeProposalRestrictions()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.36. TraderPoolRiskyProposal, line 123. Minor, the **invest()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.37. TraderPoolRiskyProposal, line 183. Minor, the **divest()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.38. TraderPoolRiskyProposal, line 209. Minor, the **exchange()** function allows modification of a zero proposal. The requirement should also check that **proposalId > 0**.

5.39. TraderPoolRiskyProposal, line 423. **Critical**, the **_divestActivePortfolio()** function will produce profit if called right after **invest()**.

5.40. UserRegistry, line 32. Optimization, the **agreeToPrivacyPolicy()** function reads **documentHash** value from storage multiple times.

5.41. UserRegistry, line 53. Note, the **agreed()** function will return true if the user agrees but the document hash is changed afterwards.

6. Verification Line By Line Review. Acknowledged Findings.

6.1. ERC721Multiplier, line 38. Optimization, the **lock()** function reads the **_tokens[tokenId]** value from storage multiple times.

6.2. ERC721Multiplier, line 67. Optimization, the **getExtraRewards()** function reads the **_tokens[latestLockedTokenId]** value from storage multiple times.

6.3. ERC721Multiplier, line 80. Optimization, the **getCurrentMultiplier()** function reads the **_tokens[latestLockedTokenId]** value from storage multiple times.

6.4. ERC721Power, line 39. Note, the **totalPower** will always be outdated as long as there are not fully collateralized tokens that didn't perform **recalculateNftPower()** in the current block.

6.5. ERC721Power, line 126. Optimization, the **removeCollateral()** function reads the **nftInfos[tokenId]** value from storage multiple times.

6.6. TokenSaleProposal, line 82. Optimization, the **vestingWithdraw()** function updates the **purchase.latestVestingWithdraw** variable value, which is only used for information purposes.

6.7. GovValidators, line 126. Note, proposals could be created with **quorum** set to zero, which will allow any validator to pass the proposal.

6.8. GovPool, line 75. Note, the **deployerBABTid** variable is only set and never changes, also not used in other contracts. Consider removing or making it immutable.

6.9. IGovSettings, line 36. Note, the **quorum** is used in an unconventional way. Usually **quorum** indicates the minimum number of participants to make voting possible, then a decision is made on the majority of the votes. Here the quorum meaning is the number (part) of votes required for the passing of the proposal, so if quorum is set to 10% then if 90% of participants are against, they would not be able to defeat the vote, which will be passed by another 10%.

6.10. GovUserKeeperView, line 86. Note, the **nftVotingPower()** function could produce **nftPower > sum(perNftPower[])**.

6.11. TraderPoolDivest, line 136. Optimization, the **_checkUserBalance()** function denies users from doing invest and divest in the same block, but that is not needed if divest does not produce immediate profit. It doesn't protect from the sandwich attacks, because instead of wrapping the victim's **invest()** into invest-divest, the attacker would just wrap the swap itself. User is already providing a minimum expected return as a means of sandwich protection.

Oleksii Matiiasevych