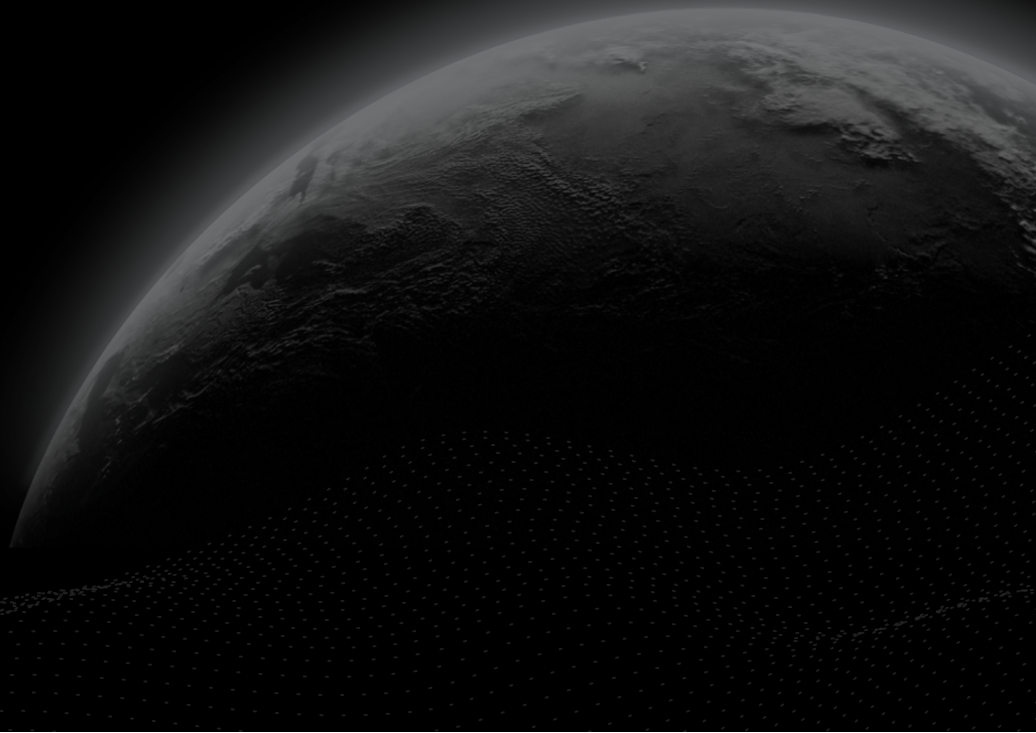




Security Assessment

UBD Network

CertiK Assessed on Jul 14th, 2023





Certik Assessed on Jul 14th, 2023

UBD Network

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

Executive Summary

TYPES

DeFi

ECOSYSTEM

Ethereum (ETH)

METHODS

Formal Verification, Manual Review, Static Analysis

LANGUAGE

Solidity

TIMELINE

Delivered on 07/14/2023

KEY COMPONENTS

N/A

CODEBASE

<https://gitlab.com/devops177/ubdn-token-sale/>[View All in Codebase Page](#)

COMMITTS

- c25a0566ebd4a61a3462beea679218dfb3f0459a
- 6165d245a8fc88de55c697dc84f49008920abf98

[View All in Codebase Page](#)

Vulnerability Summary



7

Total Findings

3

Resolved

1

Mitigated

0

Partially Resolved

3

Acknowledged

0

Declined

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.

4 Major

1 Resolved, 1 Mitigated, 2 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.

2 Minor

2 Resolved



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.

1 Informational

1 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 | UBD NETWORK

I **Summary**

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

I **Review Notes**

[Overview](#)

[External Dependencies](#)

[UBDNToken.sol](#)

[UBDNLockerDistributor.sol](#)

I **Findings**

[UBN-01 : Trust and Transparency Concerns In `UBDNLockerDistributor` Contract](#)

[UBN-02 : Centralization Risks in `UBDNLockerDistributor` Contract](#)

[UBN-03 : Distribution Token Change Vulnerabilities in `UBDNLockerDistributor` Contract](#)

[UBT-02 : Initial Token Distribution](#)

[UBN-04 : Potential Price Slippage in `buyTokensForExactStable\(\)` Function](#)

[UBN-05 : Potential Reentrancy Attack](#)

[UBN-06 : Potential Precision Loss](#)

I **Optimizations**

[UBN-08 : Unnecessary Gas Cost due to Repeated Token Decimal Queries](#)

[UBT-03 : Variables That Could Be Declared as Immutable](#)

I **Formal Verification**

[Considered Functions And Scope](#)

[Verification Results](#)

I **Appendix**

I **Disclaimer**

CODEBASE | UBD NETWORK

Repository





<https://gitlab.com/devops177/ubdn-token-sale/>

Commit

- c25a0566ebd4a61a3462beea679218dfb3f0459a
- 6165d245a8fc88de55c697dc84f49008920abf98

AUDIT SCOPE | UBD NETWORK

4 files audited ● 2 files with Acknowledged findings ● 2 files without findings

ID	Repo	Commit	File	SHA256 Checksum
● UBN	devops177/ubdn-token-sale	c25a056	 contracts/UBDNLockerDistributor.sol	dcbbf68ab4e4bbf687a9b5ba24be88f5902cb0df852d38d6364c24cf1835d0ba
● UBT	devops177/ubdn-token-sale	c25a056	 contracts/UBDNToken.sol	03d2ffaf30c6e8eb9abb5e93819fcc716418d76786b06482299fa20598b8456f
● UBD	devops177/ubdn-token-sale	2fb24d3	 contracts/UBDNLockerDistributor.sol	e20c947be4a46888d626dd83e83c6ea26189c42bb42bea231743ff1ab7606665
● UDN	devops177/ubdn-token-sale	2fb24d3	 contracts/UBDNToken.sol	52391359fa5b9165bd58534aa830cacf572a5c7fa3ba73b3289fb407c265c3cd

APPROACH & METHODS | UBD NETWORK

This report has been prepared for UBD Network to discover issues and vulnerabilities in the source code of the UBD Network 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 | UBD NETWORK

Overview

The **UBD Network** is an asset management project. **It should be noted that the audit's scope was only focused on the UBDN token contract and the sale contract for UBDN token.** Here are the introduction to these contracts.

- `UBDNToken.sol` : An ERC20 token named "UBD Network" with the symbol "UBDN". A specific address, the `minter`, is granted exclusive minting permissions. Initial supply is given to the `_initialkeeper` and additional tokens can be minted by the `minter` using the mint function. The `minter` is intended to be the `UBDNLockerDistributor` contract.
- `UBDNLockerDistributor.sol` : This contract is used for distributing the UBDN tokens with a locking mechanism. The tokens are distributed over time, with a lock period that prevents them from being claimed instantly. Tokens can be bought using stable coins and locked for a period defined during the contract creation. An administrator can set the payment and distribution tokens.

External Dependencies

The following are external contracts referred to in the contracts. The contract mainly uses OpenZeppelin contracts and libraries for the templates and setup of contracts:

UBDNToken.sol

- `minter` : The minter for the UBDN token, which should be the `UBDNLockerDistributor` contract.

UBDNLockerDistributor.sol

- `distributionToken` : The token to be distributed, which should be the `UBDNToken` contract.
- `paymentTokens` : The addresses for stablecoin tokens.

It is assumed that these addresses, contracts, and libraries are valid and are implemented properly within the current project.

FINDINGS | UBD NETWORK

7
Total Findings0
Critical4
Major0
Medium2
Minor1
Informational

This report has been prepared to discover issues and vulnerabilities for UBD Network. Through this audit, we have uncovered 7 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
UBN-01	Trust And Transparency Concerns In <code>UBDNLockerDistributor</code> Contract	Centralization	Major	● Acknowledged
UBN-02	Centralization Risks In <code>UBDNLockerDistributor</code> Contract	Centralization	Major	● Mitigated
UBN-03	Distribution Token Change Vulnerabilities In <code>UBDNLockerDistributor</code> Contract	Logical Issue, Centralization	Major	● Resolved
UBT-02	Initial Token Distribution	Centralization	Major	● Acknowledged
UBN-04	Potential Price Slippage In <code>buyTokensForExactStable()</code> Function	Logical Issue	Minor	● Resolved
UBN-05	Potential Reentrancy Attack	Volatile Code	Minor	● Resolved
UBN-06	Potential Precision Loss	Incorrect Calculation	Informational	● Acknowledged

UBN-01 | TRUST AND TRANSPARENCY CONCERNS IN UBDNLockerDistributor CONTRACT

Category	Severity	Location	Status
Centralization	● Major	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): 59	● Acknowledged

Description

In the `UBDNLockerDistributor` contract, the contract owner is granted significant privileges, including the ability to withdraw all the deposited funds (stablecoins) from the contract by sending them to the `owner()` address. This represents a major centralization risk.

```
function buyTokensForExactStable(address _paymentToken, uint256 _inAmount)
    external
    returns(uint256)
{
    require(address(distributionToken) != address(0), 'Distribution not
Define');
    require(paymentTokens[_paymentToken], 'This payment token not supported');

    // 1. Receive payment
    IERC20Mint(_paymentToken).safeTransferFrom(msg.sender, owner(), _inAmount);
    ...
}
```

In this case, users who deposit their stablecoins into the `UBDNLockerDistributor` contract in order to buy UBDN tokens may lose their assets if the owner account is compromised or the owner chooses to withdraw the deposited funds without properly operating the project. Furthermore, the lock period of 90 days is long. Users who purchase UBDN tokens cannot claim these tokens immediately; instead, they must wait for this lock period to expire. During this time, if the owner account is compromised or the project stops running, users may lose the values of the locked tokens.

Important Note: Certain identification and KYC procedures were attempted to be applied to the project team in order to better understand the centralization situation and potential risks of the project. The project team refused to cooperate with the investigation efforts given the excuse of approaching the timeline for launch. Thus based on the negative signals we concluded that there is potential high risk to the project. We strongly advise end users to conduct further research and exercise due diligence before engaging with the project. It is crucial for end users to independently verify and assess all available information to make informed decisions.

Recommendation

To enhance trust, transparency, and investor confidence, the audit team recommends the following:

1. **Implement KYC (Know Your Customer) procedures:** By implementing KYC procedures, the transparency and accountability of the project can be increased. This will create trust among investors and help protect their interests.
2. **Lock the collected funds in a vault contract:** Instead of transferring the collected funds directly to the owner, they should be locked in a secure vault contract. The release of funds from this contract can be regulated based on project milestones, ensuring that the funds are used for the intended purposes.
3. **Implement a refund mechanism:** A refund mechanism should be provided so that investors can choose to exit the project if they want to. This allows investors to recover their funds if they feel that the project is not meeting their expectations or if they suspect any fraudulent activity.

I Alleviation

[UBD Network Team, 06/19/2023]: The team made the following clarifications:

1. All funds are transferred to multi-sig address for implementing project features.
2. After a 90-day timelock, each user can claim his/her UBDN tokens anyway, even if the owner account is compromised.
3. DEX, such as Uniswap (UBDN token pairs), is exactly the Implementation of the refund mechanism. Implementing the same logic in UBDN contracts seems like overhead.

[Certik, 06/20/2023]:

1. **Risk of Centralization:** The UBDNLockerDistributor contract currently transfers funds directly to the owner's account. Although it is a multisig account, this creates a central point of failure. If the multi-sig account keys are compromised, there's a risk of losing the funds. Our recommendation for locking funds in a vault contract is to ensure the safety and controlled release of funds based on project milestones.
2. **90-Day Timelock:** We agree that users could claim their UBDN tokens even if the owner account is compromised after the 90-day lock period. However, the issue is if the project stops running or the multi-signature is compromised for any reason during the lock period, the UBDN tokens may lose their value entirely, leaving users unable to recover their initial investment. This is the risk we wanted to highlight.
3. **Refund Mechanism:** While it's true that users can sell their UBDN tokens on Uniswap or other exchanges, it's not exactly a refund mechanism since the price of the UBDN token is unknown. Our suggestion for a refund mechanism within the contract is to allow users to withdraw their initial investment (in stablecoin) if they feel the project is not meeting their expectations. This provides users with additional protection and could boost their confidence in the project.

UBN-02 CENTRALIZATION RISKS IN UBDNLockerDistributor CONTRACT

Category	Severity	Location	Status
Centralization	● Major	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): <u>92</u> , <u>100</u>	● Mitigated

Description

In the `UBDNLockerDistributor` contract, the `OWNER_ROLE` has authority over the following functions:

- `setPaymentTokenStatus()` : Sets the status of a given payment token with a 48-hour timelock.
- `setDistributionToken()` : Sets the distribution token only once.
- `renounceOwnership()` : Renounces ownership of the contract, leaving it without an owner and thus disabling the functionality of `onlyOwner` restricted functions.
- `transferOwnership()` : Transfers ownership of the contract to a new account. This function can only be called by the current owner.

The `Guardian` has authority over the following function:

- `emergencyPause()` : Emergency pause a payments method for one hour.

Any compromise to the `OWNER_ROLE` could enable an attacker to take advantage of these functionalities, potentially leading to security risks. In particular, a malicious actor gaining control of the `OWNER_ROLE` could transfer ownership to an account under their control or even renounce ownership, rendering the contract ownerless and disabling certain functions. Therefore, the audit team strongly advises clients to mitigate this risk by applying decentralization and promoting transparency in future endeavors.

Important Note: Certain identification and KYC procedures were attempted to be applied to the project team in order to better understand the centralization situation and potential risks of the project. The project team refused to cooperate with the investigation efforts given the excuse of approaching the timeline for launch. Thus based on the negative signals we concluded that there is potential high risk to the project. We strongly advise end users to conduct further research and exercise due diligence before engaging with the project. It is crucial for end users to independently verify and assess all available information to make informed decisions.

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 ($\frac{2}{3}$, $\frac{3}{5}$) 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

[UBD Network Team, 06/16/2023]: The team updated the code by adding 48 hours timelock to the new payment method and the guardians have possibility to pause receiving payment in exact stablecoin for one hour in commit [34977e7a9db8f53aa1784775dac5256eb00047fa](https://github.com/ubd-network/ubd-network/blob/master/contracts/UBDLockerDistributor.sol#L1784775dac5256eb00047fa).

[Certik, 06/16/2023]: After reviewing the new update to the `UBDLockerDistributor` contract, we would like to remind the UBD Network team of a particular scenario. In an instance where a new token is introduced through the `setPaymentTokenStatus` function, and `paymentTokens[_token]` is adjusted to `block.timestamp +`

`ADD_NEW_PAYMENT_TOKEN_TIMELOCK`, it's important to note that a guardian can modify the `paymentTokens[_token]` to `block.timestamp + EMERGENCY_PAYMENT_PAUSE` by utilizing the `emergencyPause()` function. Consequently, the lock time of the new token can be decreased from the standard 48 hours to approximately just 1 hour.

[UBD Network Team, 06/18/2023]: The team resolved this issue in commit [6165d245a8fc88de55c697dc84f49008920abf98](#).

[Certik, 06/19/2023]: The changes were reviewed, and the status of this finding will be updated to 'mitigated' once we have verified the new on-chain address and multi-signature wallet.

[UBD Network Team, 06/27/2023]: The team deployed a new `UBDNLockerDistributor` contract and a new `UBDN` token in the mainnet.

- `UBDNLockerDistributor.sol` : [0x6D8b29c195b9478D678cD9eA7aD870ECfb0A869F](#)
 - On June 27th, 2023, the owner of the contract is the 2-of-3 multi-signature wallet [0xE206f8AC6067d8253C57D86ac96A789Cd90ed4D4](#). The owners of this multi-signature wallet are:
 - The `distributionToken` was set to the new `UBDN` token deployed at [0xd624e5c89466a15812c1d45ce1533be1f16c1702](#).
- `UBDNToken.sol` : [0xd624e5c89466a15812c1d45ce1533be1f16c1702](#)

[Certik, 06/27/2023]: The UBD Network team project has applied the timelock and 2-of-3 multi-signature solution as their short-term solution. The relevant addresses provided by the team on 06/27/2023 are the following:

- 2-of-3 Multi-signature Wallet: [0xE206f8AC6067d8253C57D86ac96A789Cd90ed4D4](#)
- Owners of the 2-of-3 Multi-signature Wallet:
 - [eth:0x81bBB98fFe0844687958f81E15a8803D951547e4](#)
 - [eth:0x229CfeEd196479bbDb13E9E6fCAE4EAd5eD7D215](#)
 - [eth:0x6E367c15Db1c980d4e1869637ACa05F792F328c3](#)

While this strategy has indeed reduced the risk, it's crucial to note that it has not completely eliminated it. CertiK strongly encourages the project team periodically revisit the private key security management of all above-listed addresses.

UBN-03 | DISTRIBUTION TOKEN CHANGE VULNERABILITIES IN UBDNLockerDistributor CONTRACT

Category	Severity	Location	Status
Logical Issue, Centralization	● Major	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): <u>100~106</u>	● Resolved

Description

```
100     function setDistributionToken(address _token)
101         external
102         onlyOwner
103     {
104         distributionToken = IERC20Mint(_token);
105         emit DistributionTokenSet(_token);
106     }
```

The `UBDNLockerDistributor` contract allows the owner to change the distribution token at any point in time. This creates a couple of potential issues:

Changing Distribution Token to an Unwanted Token or Empty Address: The owner of the contract can change the distribution token to a useless token or even an empty address (0x0). In such a case, users who previously purchased the distribution token would end up with worthless tokens or no tokens at all, resulting in the loss of their stablecoins.

Insufficient Balance for New Distribution Token: The owner could change the distribution token during the distribution period to a new token that the contract does not have a sufficient balance of. And there are no checks in place to ensure that the contract has a sufficient balance of the new token. If the contract does not have a sufficient balance and a user tries to claim their tokens, the `claimTokens()` function will fail, potentially causing a loss of funds for the users.

Recommendation

The audit team recommends UBN Network team restrict the change of the distribution token to certain periods (For example, only allowing it to be changed before the distribution starts or after it ends), or make the distribution token unchangeable after it's been set initially.

Alleviation

[UBD Network Team, 06/10/2023]: The team updated the code by adding a check `require(distributedAmount == 0)` to the `setDistributionToken()` function to ensure the `distributionToken` cannot be changed after distribution start in commit [57a8da2818b83025e82e0e77f3b9091df2977786](#).

[Certik, 06/12/2023]: The audit team would like to remind the UBD Network team that the `setDistributionToken()` function is vulnerable to a front-running attack that some malicious users can prevent the update of the distribution token. The malicious user only needs to purchase one previous distribution token before the execution of `setDistributionToken()` function.

[UBD Network Team, 06/16/2023]: The team resolved this issue by ensuring that the distribution token can only be initialized once in commit [30e17749ce88a0c7407949af2098c0c7023d57c8](#).

```
function setDistributionToken(address _token)
    external
    onlyOwner
{
    require(address(distributionToken) == address(0), "Can call only once");
    distributionToken = IERC20Mint(_token);
    emit DistributionTokenSet(_token);
}
```

UBT-02 | INITIAL TOKEN DISTRIBUTION

Category	Severity	Location	Status
Centralization	● Major	contracts/UBDNToken.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): 19	● Acknowledged

Description

`INITIAL_SUPPLY` UBD Network (UBDN) tokens are sent to `_initialKeeper` when deploying the contract. This could be a centralization risk as the `_initialKeeper` can distribute tokens without obtaining the consensus of the community.

```
15     constructor(address _initialKeeper, address _minter, uint256 _premintAmount)
16     ERC20("UBD Network", "UBDN")
17     {
18         INITIAL_SUPPLY = _premintAmount;
19         _mint(_initialKeeper, INITIAL_SUPPLY);
20         minter = _minter;
21     }
22 }
```

On-Chain Analysis (06/27/2023)

- The address of the finalized `UBDN` token is [0xd624e5c89466a15812c1d45ce1533be1f16c1702](#)
- During the [creation](#) of UBDN token contract, 5,000,000 UBDN tokens were sent to the multi-signature wallet [0xE206f8AC6067d8253C57D86ac96A789Cd90ed4D4](#).

The audit team would like to remind users that two outdated `UBDN` tokens exist in the mainnet.

- [0x3c388f96d5c698f980c49d31fc48ef88f90d0d8b](#)
- [0xda7d1ca5019d4ca46fa9e70035a0764c7547cf2c](#)

The audit team would like to know the token distribution plan for the initial supply hold by `_initialKeeper`.

Important Note: Certain identification and KYC procedures were attempted to be applied to the project team in order to better understand the centralization situation and potential risks of the project. The project team refused to cooperate with the investigation efforts given the excuse of approaching the timeline for launch. Thus based on the negative signals we concluded that there is potential high risk to the project. We strongly advise end users to conduct further research and exercise due diligence before engaging with the project. It is crucial for end users to independently verify and assess all available information to make informed decisions.

Recommendation

It is recommended that the team be transparent regarding the initial token distribution process. The token distribution plan should be published in a public location that the community can access. The team should make efforts to restrict access to the private keys of the deployer account or EOAs. A multi-signature ($\frac{2}{3}$, $\frac{3}{5}$) wallet can be used to prevent a single point of failure due to a private key compromise. Additionally, the team can lock up a portion of tokens, release them with a vesting schedule for long-term success, and deanonymize the project team with a third-party KYC provider to create greater accountability.

■ Alleviation

[UBD Network Team, 06/20/2023]: The team acknowledged the issue and decided not to make any changes to the current design.

[UBD Network Team, 06/27/2023]: The team deployed a new UBDN token to [0xd624e5c89466a15812c1d45ce1533be1f16c1702](https://www.benzinga.com/markets/cryptocurrency/23/07/33185637/UBDNetwork-Transforming-Crypto-Industry-with-Decentralization-and-innovation). The issue of initial distribution token still exists, and the details have been updated in the Description section.

[UBD Network Team, 07/11/2023]: The team has published an article that outlines the plan for distributing tokens. The article can be accessed through this link: <https://www.benzinga.com/markets/cryptocurrency/23/07/33185637/UBDNetwork-Transforming-Crypto-Industry-with-Decentralization-and-innovation>.

[UBD Network Team, 07/13/2023]: The team confirmed that the token distribution plan is for the initial supply (5,000,000 UBDN tokens). These preminted tokens are for private investors and the community who wants to buy tokens and support development, marketing, listing, and liquidity.

[CertiK, 07/13/2023]: As of 07/13/2023, the 5,000,000 UBDN tokens were held by the [multi-signature wallet](#) and the distribution has not yet begun. The status of this issue will be updated after the distribution plan has been completed.

UBN-04 | POTENTIAL PRICE SLIPPAGE IN `buyTokensForExactStable()` FUNCTION

Category	Severity	Location	Status
Logical Issue	Minor	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): 51	Resolved

Description

The `buyTokensForExactStable()` function allows users to purchase distribution tokens by paying with a certain amount of stablecoins. However, because Ethereum transactions don't execute immediately and are added to a pool of pending transactions, other users (or bots) could potentially manipulate the outcome of the function call by watching the transaction pool and placing their own transaction with a higher gas price.

The code uses the `_calcTokensForExactStable()` function to determine the amount of distribution tokens that will be minted and locked for the user based on the amount of stablecoins they're spending. This calculation is based on the current state of the contract at the time of execution. If another user executes the same function just before the original transaction gets mined, the `distributedAmount` would have increased and the calculated tokens to be received by the original user will decrease. This means the original user may receive fewer tokens than they originally anticipated.

Recommendation

The audit team recommends UBN Network team provide an optional argument that specifies the user's acceptable slippage limit. If the actual amount of tokens to be minted falls below this limit, the transaction could be reverted to protect the user from price slippage.

Alleviation

[UBD Network Team, 06/10/2023]: The team resolved the issue by providing a function called `buyTokensForExactStableWithSlippage()` with an argument called `_outNotLess` to prevent slippage in commit [d304b1b20e489d4b3183c04cc5dcc381be5cc402](#).

UBN-05 | POTENTIAL REENTRANCY ATTACK

Category	Severity	Location	Status
Volatile Code	Minor	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): 59 , 66 , 67 , 182~184	Resolved

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.

External call(s)

```
59      IERC20Mint(_paymentToken).safeTransferFrom(msg.sender, owner(), _inAmount);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,
 - `returndata = address(token).functionCall(data, "SafeERC20: low-level call failed")`
- In `Address.functionCallWithValue`,
 - `(success, returndata) = target.call{value: value}(data)`

State variables written after the call(s)

```
67      distributedAmount += outAmount;
```

```
66      _newLock(msg.sender, outAmount);
```

- This function call executes the following assignment(s).
- In `UBDNLockerDistributor._newLock`,
 - `userLocks[_user].push(Lock(_lockAmount, block.timestamp + LOCK_PERIOD))`

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

[UBD Network Team, 06/10/2023]: The team resolved the issue by using Checks-Effects-Interactions Pattern in commit [ebf314977e84a080900bbd413a8d7a2acb45ba91](#).

UBN-06 | POTENTIAL PRECISION LOSS

Category	Severity	Location	Status
Incorrect Calculation	● Informational	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): <u>115</u>	● Acknowledged

Description

The `UBDNLockerDistributor` contract contains precision loss issues in its `calcTokensForExactStable()` and `calcStableForExactTokens()` functions. The issue is rooted in the way Solidity handles integer division, rounding down to the nearest integer, which can cause minor discrepancies. Over repeated operations, these minor discrepancies can compound, leading to more significant discrepancies.

Proof of Concept

The fuzzing test was written by the foundry to show the a potential discrepancy between `calcTokensForExactStable` and `calcStableForExactTokens` due to precision loss issues in the computations.

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

import "forge-std/Test.sol";

contract PriceCalculationTest is Test {
    uint256 constant public START_PRICE = 1;          // 1 stable coin unit, not
    decimal.
    uint256 constant public PRICE_INCREASE_STEP = 1; // 1 stable coin unit, not
    decimal.
    uint256 constant public INCREASE_FROM_ROUND = 1;
    uint256 constant public ROUND_VOLUME = 1_000_000e18; // in wei

    function setUp() public {

    }

    function testPriceCalculation(uint256 _paymentToken_decimals, uint256 _inAmount,
    uint256 distributedAmount) public {
        // limit _paymentToken_decimals between 6 and 18
        vm.assume(_paymentToken_decimals >= 6);
        vm.assume(_paymentToken_decimals <= 18);
        vm.assume(distributedAmount < 100_000_000 ether);
        vm.assume(_inAmount < 100_000 ether);
        // calculate outAmount
        uint256 outAmount = _calcTokensForExactStable(_paymentToken_decimals,
        _inAmount, distributedAmount);
        vm.assume(outAmount > 0);
        // calculate inAmount
        uint256 inAmount = _calcStableForExactTokens(_paymentToken_decimals,
        outAmount, distributedAmount);
        // check inAmount
        uint256 diff = _inAmount - inAmount;
        assertTrue(diff <= 1);
    }

    function testPriceCalculationWithExample() public {
        uint256 _paymentToken_decimals = 18;
        uint256 _inAmount = 334;
        uint256 distributedAmount = 3604557350015191483654609;
        uint256 outAmount = _calcTokensForExactStable(_paymentToken_decimals,
        _inAmount, distributedAmount);
        uint256 inAmount = _calcStableForExactTokens(_paymentToken_decimals,
        outAmount, distributedAmount);

        uint256 curR = distributedAmount / ROUND_VOLUME + 1;
        (uint256 curPrice, uint256 curRest) = _priceInUnitsAndRemainByRound(curR,
        distributedAmount);
    }
}
```

```
        console.log("price: ", curPrice);
        console.log("_inAmount: ", _inAmount);
        console.log("outAmount: ", outAmount);
        console.log("inAmount: ", inAmount);
        uint256 diff = _inAmount - inAmount;
        console.log("diff: ", diff);
    }

    function _calcStableForExactTokens(uint256 _paymentToken_decimals, uint256
_outAmount, uint256 distributedAmount)
        internal
        view
        returns(uint256 inAmount)
    {
        uint256 outA = _outAmount;
        uint256 curR = distributedAmount / ROUND_VOLUME + 1;
        uint256 curPrice;
        uint256 curRest;
        while (outA > 0) {
            (curPrice, curRest) = _priceInUnitsAndRemainByRound(curR,
distributedAmount);
            if (outA > curRest) {
                inAmount += curRest
                    * curPrice * 10**_paymentToken_decimals
                    / (10**18);
                outA -= curRest;
                ++ curR;
            } else {
                inAmount += outA
                    * curPrice * 10**_paymentToken_decimals
                    / (10**18);
                return inAmount;
            }
        }
    }

    function _calcTokensForExactStable(uint256 _paymentToken_decimals, uint256
_inAmount, uint256 distributedAmount)
        internal
        view
        returns(uint256 outAmount)
    {
        uint256 inA = _inAmount;
        uint256 curR = distributedAmount / ROUND_VOLUME + 1;
        uint256 curPrice;
        uint256 curRest;
        while (inA > 0) {
            (curPrice, curRest) = _priceInUnitsAndRemainByRound(curR,
distributedAmount);
            if (
```

```
        // calc out amount
        inA
        * (10**18)
        / (curPrice * 10**_paymentToken_decimals)
        > curRest
    )
}
// Case when inAmount more then price of all tokens
// in current round
outAmount += curRest;
inA -= curRest
    * curPrice * 10**_paymentToken_decimals
    / (10**18);
++ curR;
} else {
    // Case when inAmount less or equal then price of all tokens
    // in current round
    outAmount += inA
        * 10**18
        / (curPrice * 10**_paymentToken_decimals);
    return outAmount;
}
}
}

function _priceInUnitsAndRemainByRound(uint256 _round, uint256
distributedAmount)
    internal
    view
    virtual
    returns(uint256 price, uint256 rest)
{
    uint256 _currentRound = distributedAmount / ROUND_VOLUME + 1;
    if (_round < INCREASE_FROM_ROUND){
        price = START_PRICE;
    } else {
        price = START_PRICE + PRICE_INCREASE_STEP * (_round -
INCREASE_FROM_ROUND + 1);
    }

    // in finished rounds rest always zero
    if (_round < _currentRound){
        rest = 0;

    // in current round need calc
    } else if (_round == _currentRound){
        if (_round == 1){
            // first round
            rest = ROUND_VOLUME - distributedAmount;
        }
    }
}
```



```
    } else {  
        rest = ROUND_VOLUME - (distributedAmount % ROUND_VOLUME);  
    }  
  
    // in future rounds rest always ROUND_VOLUME  
    } else {  
        rest = ROUND_VOLUME;  
    }  
}  
}
```

Output Logs

```
No files changed, compilation skipped
```

```
[FAIL. Reason: Assertion failed. Counterexample:
```

Logs:

Traces:

```
├ [0] VM::assume(true) [staticcall]
```

$$| \quad \leftarrow ()$$

```
├ [0] VM::assume(true) [staticcall]
```

$$| \quad \leftarrow ()$$

```
├ [0] VM::assume(true) [staticcall]
```

$$| \quad \leftarrow ()$$

```
└─ [0] VM::assume(true) [staticcall]
```

$$| \quad \leftarrow ()$$

```
├ [0] VM::assume(true) [staticcall]
```

$$| \quad \leftarrow ()$$

```
| emit log(: Error: Assertion Failed)
```

```
└─ [0] VM::store(VM: [0x7109709ECfa91a80626fF3989D68f67F5b1DD12D],
```

[illegible]
$$| \quad \vdash \quad \leftarrow \quad ()$$
$$L \leftarrow ()$$

Logs:

```
price: 5
```

```
_inAmount: 334
```

```
outAmount: 66
```

```
inAmount: 330
```

```
diff: 4
```

Failing tests:

Encountered 1 failing test in test/PriceCalculationTest.t.sol:PriceCalculationTest

Encountered a total of 1 failing tests, 1 tests succeeded

[Certik, 06/15/2023]: The audit team agrees the precision loss is insignificant with high digits, but there is no restriction on the input value for `_inAmount` . As a result, a user could potentially input any value into this function. The precision loss is significant only in lower digit values, so the audit team has classified the severity of this finding as informational.

OPTIMIZATIONS | UBD NETWORK

ID	Title	Category	Severity	Status
UBN-08	Unnecessary Gas Cost Due To Repeated Token Decimal Queries	Gas Optimization	Optimization	● Resolved
UBT-03	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	● Resolved

UBN-08 | UNNECESSARY GAS COST DUE TO REPEATED TOKEN DECIMAL QUERIES

Category	Severity	Location	Status
Gas Optimization	● Optimization	contracts/UBDNLockerDistributor.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): 197 , 224	● Resolved

Description

The linked Solidity code in the smart contract unnecessarily queries the number of decimals for both the payment token and the distribution token within a loop. These queries lead to extra gas costs, as they are repeatedly fetching data that is not changing throughout the loop execution.

```
while (inA > 0) {
    (curPrice, curRest) = _priceInUnitsAndRemainByRound(curR);
    if (
        // calc out amount
        inA
        * (10**distributionToken.decimals())
        / (curPrice * 10**IERC20Mint(_paymentToken).decimals())
        > curRest
    )
    {
        // Case when inAmount more then price of all tokens
        // in current round
        outAmount += curRest;
        inA -= curRest
            * curPrice * 10**IERC20Mint(_paymentToken).decimals()
            / (10**distributionToken.decimals());
        ++ curR;
    } else {
        // Case when inAmount less or equal then price of all tokens
        // in current round
        outAmount += inA
            * 10**distributionToken.decimals()
            / (curPrice * 10**IERC20Mint(_paymentToken).decimals());
        return outAmount;
    }
}
```

Recommendation

Recommend querying the number of decimals for both tokens outside of the loop, storing these values in local variables and using these variables within the loop.

I Alleviation

[UBD Network Team, 06/10/2023]: The team resolved the issue by querying the number of decimals for both tokens outside of the loop in commit [2bf5e9f6dbce4d91cfb9547ea70fa280b9a0814f](#).

UBT-03 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	● Optimization	contracts/UBDNToken.sol (c25a0566ebd4a61a3462beea679218dfb3f0459a): <u>13</u>	● Resolved

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

[UBD Network Team, 06/10/2023]: The team resolved the issue by declaring the variable `minter` as immutable in commit [da0e67947fee0b8361f2035a73cdc1e6e2f9602b](#).

FORMAL VERIFICATION | UBD NETWORK

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	<code>transfer</code> Prevents Transfers to the Zero Address
erc20-transfer-succeed-normal	<code>transfer</code> Succeeds on Admissible Non-self Transfers
erc20-transfer-succeed-self	<code>transfer</code> Succeeds on Admissible Self Transfers
erc20-transfer-correct-amount	<code>transfer</code> Transfers the Correct Amount in Non-self Transfers
erc20-transfer-correct-amount-self	<code>transfer</code> Transfers the Correct Amount in Self Transfers
erc20-transfer-change-state	<code>transfer</code> Has No Unexpected State Changes
erc20-transfer-exceed-balance	<code>transfer</code> Fails if Requested Amount Exceeds Available Balance
erc20-transfer-recipient-overflow	<code>transfer</code> Prevents Overflows in the Recipient's Balance
erc20-transfer-false	If <code>transfer</code> Returns <code>false</code> , the Contract State Is Not Changed
erc20-transfer-never-return-false	<code>transfer</code> Never Returns <code>false</code>

Property Name	Title
erc20-transferfrom-revert-from-zero	<code>transferFrom</code> Fails for Transfers From the Zero Address
erc20-transferfrom-revert-to-zero	<code>transferFrom</code> Fails for Transfers To the Zero Address
erc20-transferfrom-correct-amount	<code>transferFrom</code> Transfers the Correct Amount in Non-self Transfers
erc20-transferfrom-succeed-normal	<code>transferFrom</code> Succeeds on Admissible Non-self Transfers
erc20-transferfrom-succeed-self	<code>transferFrom</code> Succeeds on Admissible Self Transfers
erc20-transferfrom-correct-amount-self	<code>transferFrom</code> Performs Self Transfers Correctly
erc20-transferfrom-fail-exceed-balance	<code>transferFrom</code> Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-correct-allowance	<code>transferFrom</code> Updated the Allowance Correctly
erc20-transferfrom-change-state	<code>transferFrom</code> Has No Unexpected State Changes
erc20-transferfrom-fail-exceed-allowance	<code>transferFrom</code> Fails if the Requested Amount Exceeds the Available Allowance
erc20-totalsupply-succeed-always	<code>totalSupply</code> Always Succeeds
erc20-transferfrom-fail-recipient-overflow	<code>transferFrom</code> Prevents Overflows in the Recipient's Balance
erc20-transferfrom-false	If <code>transferFrom</code> Returns <code>false</code> , the Contract's State Is Unchanged
erc20-transferfrom-never-return-false	<code>transferFrom</code> Never Returns <code>false</code>
erc20-totalsupply-correct-value	<code>totalSupply</code> Returns the Value of the Corresponding State Variable
erc20-totalsupply-change-state	<code>totalSupply</code> Does Not Change the Contract's State
erc20-balanceof-succeed-always	<code>balanceOf</code> Always Succeeds
erc20-balanceof-correct-value	<code>balanceOf</code> Returns the Correct Value
erc20-balanceof-change-state	<code>balanceOf</code> Does Not Change the Contract's State
erc20-allowance-succeed-always	<code>allowance</code> Always Succeeds
erc20-allowance-correct-value	<code>allowance</code> Returns Correct Value
erc20-allowance-change-state	<code>allowance</code> Does Not Change the Contract's State

Property Name	Title	
erc20-approve-revert-zero	<code>approve</code>	Prevents Approvals For the Zero Address
erc20-approve-succeed-normal	<code>approve</code>	Succeeds for Admissible Inputs
erc20-approve-correct-amount	<code>approve</code>	Updates the Approval Mapping Correctly
erc20-approve-change-state	<code>approve</code>	Has No Unexpected State Changes
erc20-approve-false	If <code>approve</code>	Returns <code>false</code> , the Contract's State Is Unchanged
erc20-approve-never-return-false	<code>approve</code>	Never Returns <code>false</code>

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 UBDNToken (contracts/UBDNToken.sol) In Commit 2fb24d3d8d6be744a24c239b59bf37bbcfda140c

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	● True	
erc20-transfer-succeed-normal	● True	
erc20-transfer-succeed-self	● 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-succeed-normal	● True	
erc20-transferfrom-succeed-self	● 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-fail-recipient-overflow	● True	
erc20-transferfrom-false	● 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	

**Detailed Results For Contract UBDNToken (contracts/UBDNToken.sol) In Commit
c25a0566ebd4a61a3462beea679218dfb3f0459a**

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	● True	
erc20-transfer-succeed-normal	● True	
erc20-transfer-succeed-self	● 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-succeed-normal	● True	
erc20-transferfrom-succeed-self	● True	
erc20-transferfrom-correct-amount	● True	
erc20-transferfrom-correct-amount-self	● True	
erc20-transferfrom-correct-allowance	● True	
erc20-transferfrom-fail-exceed-balance	● True	
erc20-transferfrom-change-state	● True	
erc20-transferfrom-fail-exceed-allowance	● True	
erc20-transferfrom-fail-recipient-overflow	● True	
erc20-transferfrom-false	● 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 | UBD NETWORK

Finding Categories

Categories	Description
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.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.

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

Technical description

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.

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 of those functions. 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 as operations on the congruence classes arising from the bit-width of the underlying numeric 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 an ERC-20 token 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 definitions

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 steps. 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 \Box) and "eventually" (written \Diamond), we use the following predicates to reason about the validity of 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 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 for ERC-20 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)))
```

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 <= type(uint256).max && _balances[to] >= 0
  && _balances[msg.sender] <= type(uint256).max)
  ==> <>(finished(contract.transfer(to, value), return)))
```

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] <= type(uint256).max)
    ==> <>(finished(contract.transfer(to, value), return)))

```

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 <= type(uint256).max
    && _balances[msg.sender] >= 0 && _balances[msg.sender] <= type(uint256).max)
    ==> <>(finished(contract.transfer(to, value), return
        ==> _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] <= type(uint256).max)
    ==> <>(finished(contract.transfer(to, value), return
        ==> _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
    ==> (_totalSupply == old(_totalSupply) && _allowances == old(_allowances)
      && _balances[p1] == old(_balances[p1]) )))

```

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 <= type(uint256).max)
  ==> <> (reverted(contract.transfer) || finished(contract.transfer(to, value),
    !return)))

```

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 > type(uint256).max
  && _balances[to] >= 0 && _balances[to] <= type(uint256).max
  && _balances[msg.sender] <= type(uint256).max
  && value > 0 && value <= _balances[msg.sender])
  ==> <> (reverted(contract.transfer) || finished(contract.transfer(to, value),
    !return) || finished(contract.transfer(to, value), _balances[to]
      > old(_balances[to]) + value - type(uint256).max - 1)))

```

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)
  ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
      && _allowances == old(_allowances) ))))

```

erc20-transfer-never-return-false

Function `transfe` Never Returns `false` .

The transfer function must never return `false` to signal a failure.

Specification:

```

[] (! (finished(contract.transfer, !return)))

```

Properties for ERC-20 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)))

```

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)))

```

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 <= type(uint256).max
    && value >= 0 && _balances[to] >= 0 && _balances[from] >= 0
    && _balances[from] <= type(uint256).max
    && _allowances[from][msg.sender] >= 0
    && _allowances[from][msg.sender] <= type(uint256).max)
    ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

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] <= type(uint256).max
    && _allowances[from][msg.sender] <= type(uint256).max)
    ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

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] <= type(uint256).max
&& _balances[to] >= 0 && _balances[to] + value <= type(uint256).max)
==> <>(finished(contract.transferFrom(from, to, value), return
    ==> _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 <= type(uint256).max && _balances[from] >= 0
&& _balances[from] <= type(uint256).max)
==> <>(finished(contract.transferFrom(from, to, value), return
    ==> _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 <= type(uint256).max && _balances[from] >= 0
&& _balances[from] <= type(uint256).max && _balances[to] >= 0
&& _balances[to] <= type(uint256).max && _allowances[from][msg.sender] >= 0
&& _allowances[from][msg.sender] <= type(uint256).max)
==> <>(finished(contract.transferFrom(from, to, value), return
    ==> ((_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])
                == type(uint256).max))))))
```

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
    ==> (_totalSupply == old(_totalSupply) && _balances[p1] == old(_balances[p1])
      && _allowances[p2][p3] == old(_allowances[p2][p3])))))

```

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] <= type(uint256).max)
  ==> <>(reverted(contract.transferFrom)
    || finished(contract.transferFrom, !return)))

```

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 <= type(uint256).max)
  ==> <>(reverted(contract.transferFrom)
    || finished(contract.transferFrom(from, to, value), !return)
    || finished(contract.transferFrom(from, to, value), return
      && (msg.sender == from
        || _allowances[from][msg.sender] == type(uint256).max))))

```


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 > type(uint256).max && value <= type(uint256).max
  && _balances[to] >= 0 && _balances[to] <= type(uint256).max)
  ==> <>(reverted(contract.transferFrom)
    || finished(contract.transferFrom(from, to, value), !return)
    || finished(contract.transferFrom(from, to, value), _balances[to]
      > old(_balances[to]) + value - type(uint256).max - 1)))

```

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.transfer(to, value))
  ==> <>(finished(contract.transfer(to, value), !return)
  ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
    && _allowances == old(_allowances) )))

```

erc20-transferfrom-never-return-false

Function `transferFrom` Never Returns `false`.

The `transferFrom` function must never return `false`.

Specification:

```

[](!(finished(contract.transferFrom, !return)))

```

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) )))
```

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) )))

```

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) )))

```

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)))

```

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)))

```

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 <= type(uint256).max)
  ==> <>(finished(contract.approve(spender, value), return
    ==> _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
    ==> _totalSupply == old(_totalSupply) && _balances == old(_balances)
    && _allowances[p1][p2] == old(_allowances[p1][p2]) )))

```

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
    ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
    && _allowances == old(_allowances) ))))

```

erc20-approve-never-return-false

Function `approve` Never Returns `false`.

The function `approve` must never returns `false`.

Specification:

```

[](!(finished(contract.approve, !return)))

```

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.

