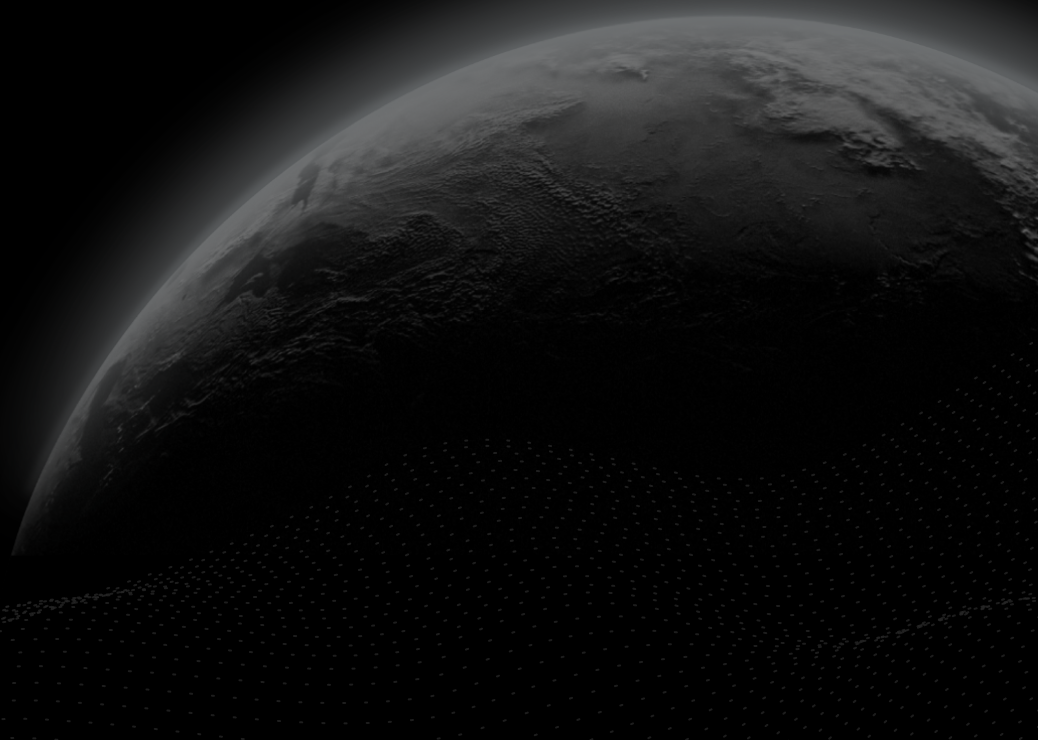
CERTIK

Security Assessment

CleanCarbon

CertiK Assessed on Sept 18th, 2023



CertiK Assessed on Sept 18th, 2023

CleanCarbon

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

Executive Summary

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

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 09/18/2023	N/A

CODEBASE

https://github.com/sotatek-dev/clean-carbon

https://github.com/CleanCarbon

View All in Codebase Page

COMMITS

base: 37268ef0ecfaf3f166707071830b41854b34a5ab

update1: 16de6575939b2afd15afefb41c0bad04daeea426

update2: da29bd77cb376fc098d959cbcd1be65eac077252

View All in Codebase Page

Vulnerability Summary

15	10	0	1	4	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	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.
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.
2	Medium	1 Resolved, 1 Partially Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
3	Minor	2 Resolved, 1 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.
8	Informational	6 Resolved, 2 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 | CLEANCARBON

Summary

Executive Summary

Vulnerability Summary

Codebase

Audit Scope

Approach & Methods

Findings

CON-10 : Centralization Related Risks

CTC-01 : Incorrect `secondsPerMonth`

CTB-01 : Rewards For `teamDev` Can Be Released Early

SCB-01 : `emergencyWithdraw()` Can Transfer Users Staked Tokens

CON-01 : Missing Zero Address Validation

CON-03 : Locked Ether

CTB-02 : Minting To `address(1)`

CON-04 : Typos

CON-05 : Missing Emit Events

CON-07 : Unchecked ERC-20 `transfer()`/`transferFrom()` Call

CON-09 : `changeAdminRole()` Restriction

CTB-04 : Time Units Can Be Used Directly

GIT-01 : Unused Parameters And Variables

GIT-02 : Calling Void Constructor

SCB-02 : `isActive` Discussion

Optimizations

ACB-01 : `ADMIN` Role Not Used In `AirdropCarbonv2`

CON-08 : Variables That Could Be Declared as Immutable

CTB-06 : Unnecessary Check

SCB-04 : Can Use `delete` To Save Gas

SCC-03 : Possibly Inefficient Memory Parameter

Formal Verification

Considered Functions And Scope

Verification Results

Appendix

Disclaimer

CODEBASE | CLEANCARBON

Repository

https://github.com/sotatek-dev/clean-carbon
https://github.com/CleanCarbon

Commit

base: 37268ef0ecfaf3f166707071830b41854b34a5ab
update1: 16de6575939b2afd15afefb41c0bad04daeea426
update2: da29bd77cb376fc098d959cbcd1be65eac077252

AUDIT SCOPE | CLEANCARBON

4 files audited ● 3 files with Acknowledged findings ● 1 file without findings

ID	Repo	Commit	File	SHA256 Checksum
● ACB	sotatek-dev/clean-carbon	37268ef	contracts/AirdropCarbonv2.sol	8230357941b56f9f3256274401f7aec06c8d2690afcb8e716ea4e352bd0596ba
● CTB	sotatek-dev/clean-carbon	37268ef	contracts/CarboTokenv2.sol	32000612fa346af668a369d383918e22e89871c4f64b86c7ab93c7a7e5756b20
● SCB	sotatek-dev/clean-carbon	37268ef	contracts/StakingCarbon.sol	0bef2b7c535980acfe2530adfaffeae13d6dd944f8cb840da90521565b5f84a2
● ICT	sotatek-dev/clean-carbon	37268ef	contracts/v1/interfaces/ICarboToken.sol	4ed789fe1023c550c1a5106a06f819901b5df0c381710cbb0eca6123bc154e17

APPROACH & METHODS | CLEANCARBON

This report has been prepared for CleanCarbon to discover issues and vulnerabilities in the source code of the CleanCarbon 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.

FINDINGS | CLEANCARBON

15	0	2	2	3	8
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for CleanCarbon. Through this audit, we have uncovered 15 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
CON-10	**Centralization Related Risks**	**Centralization**	**Major**	**Acknowledged**
CTC-01	Incorrect `secondsPerMonth`	Logical Issue	Major	Resolved
CTB-01	Rewards For `teamDev` Can Be Released Early	Logical Issue	Medium	Partially Resolved
SCB-01	`emergencyWithdraw()` Can Transfer Users Staked Tokens	Logical Issue	Medium	Resolved
CON-01	Missing Zero Address Validation	Volatile Code	Minor	Resolved
CON-03	Locked Ether	Coding Issue	Minor	Resolved
CTB-02	Minting To `address(1)`	Logical Issue	Minor	Acknowledged
CON-04	Typos	Coding Style	Informational	Resolved
CON-05	Missing Emit Events	Coding Style	Informational	Resolved
CON-07	Unchecked ERC-20 `transfer()` / `transferFrom()` Call	Volatile Code	Informational	Resolved
CON-09	`changeAdminRole()` Restriction	Coding Style	Informational	Acknowledged

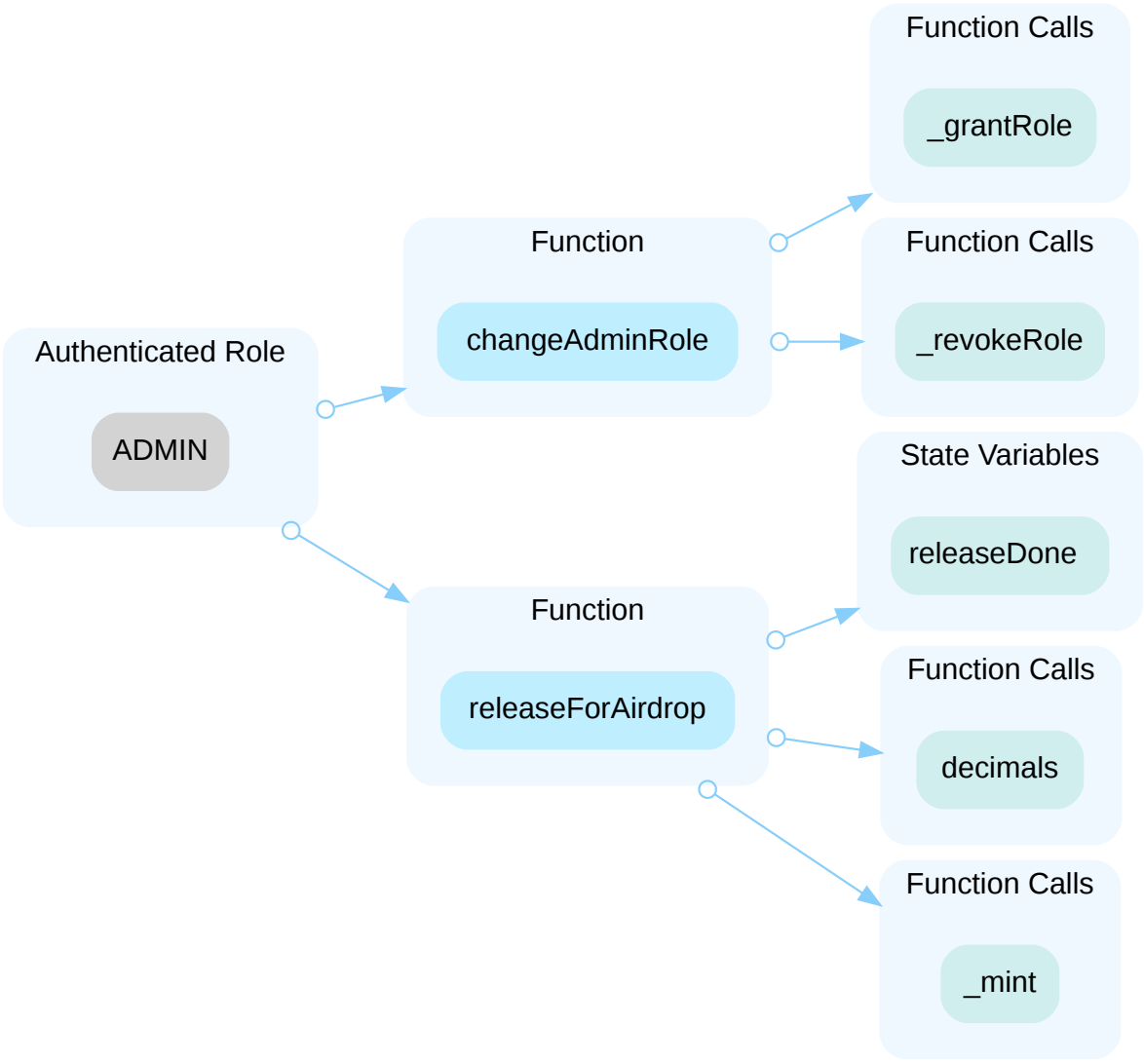
ID	Title	Category	Severity	Status
CTB-04	Time Units Can Be Used Directly	Coding Issue	Informational	Resolved
GIT-01	Unused Parameters And Variables	Coding Style	Informational	Resolved
GIT-02	Calling Void Constructor	Coding Style	Informational	Acknowledged
SCB-02	`isActive` Discussion	Logical Issue	Informational	Resolved

CON-10 | CENTRALIZATION RELATED RISKS

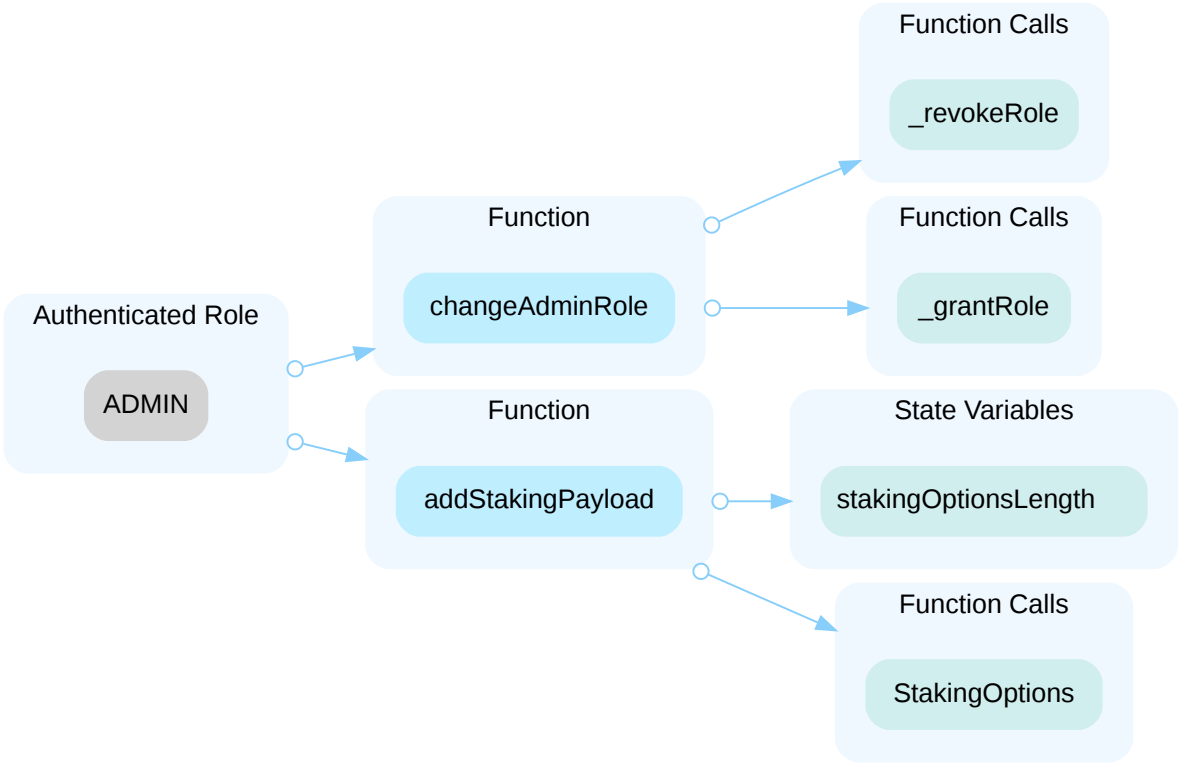
Category	Severity	Location	Status
Centralization	Major	contracts/AirdropCarbonv2.sol (base): 30, 48; contracts/CarboTokenv2.sol (base): 91, 128, 133; contracts/StakingCarbon.sol (base): 49, 54, 134	Acknowledged

Description

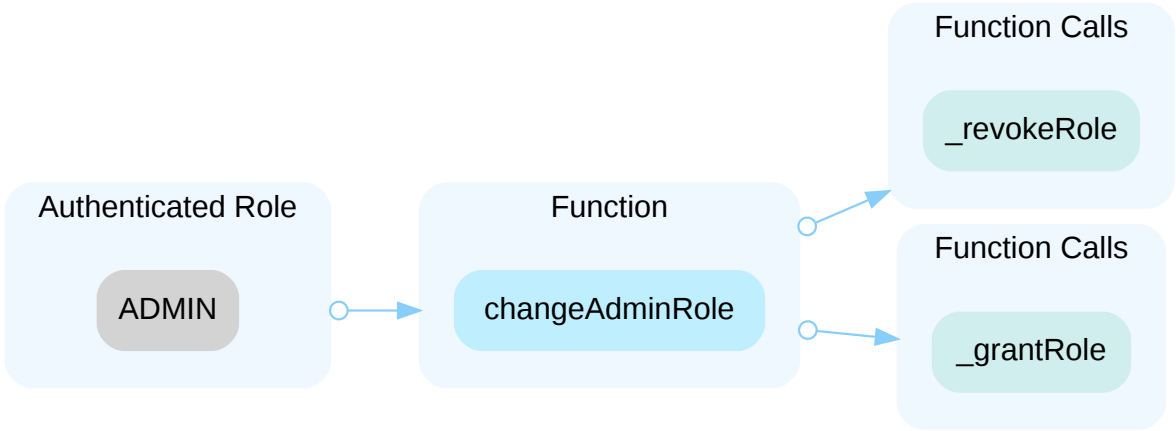
In the contract `CarboTokenv2` the role `ADMIN` has authority over the functions shown in the diagram below. Any compromise to the `ADMIN` account may allow the hacker to take advantage of this authority and change the address of the `ADMIN` role or send the tokens allocated for airdrop to any address that they wish.



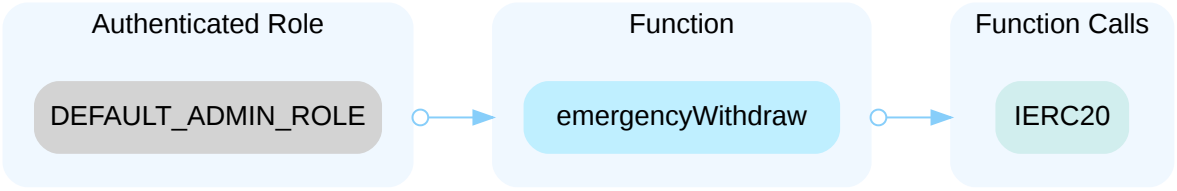
In the contract `StakingCarbon` the role `ADMIN` has authority over the functions shown in the diagram below. Any compromise to the `ADMIN` account may allow the hacker to take advantage of this authority and change the address with the `ADMIN` role or add any staking option they wish.



In the contract `AirdropCarbonv2` the role `ADMIN` has authority over the functions shown in the diagram below. Any compromise to the `ADMIN` account may allow the hacker to take advantage of this authority and change the address of the `ADMIN` role.



In the contract `CarboTokenv2`, `StakingCarbon`, and `AirdropCarbonv2` the role `DEFAULT_ADMIN_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow the hacker to take advantage of this authority and withdraw any ERC20 token held by the contract.



In addition, the DEFAULT_ADMIN_ROLE can also grant or revoke the ADMIN role.

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 recommend carefully managing 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., multi-signature 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

`[CleanCarbon, 09/07/2023]` : Acknowledged. We are aware that if the Admin wallet was ever compromised, there would be very serious issues. We take all necessary precautions to make sure our private keys stay safe.

CTC-01 | INCORRECT `secondsPerMonth`

Category	Severity	Location	Status
Logical Issue	Major	contracts/CarboTokenv2.sol (update1): 16	Resolved

Description

There are `86_400` seconds in a day, so there are `86_400*30 = 2_592_000` seconds per month (assuming the convention that a month always counts as 30 days). However, the constant `secondPerMonth` is set to `259_200`, which is the amount of seconds per 3 days. This will cause the amount that is to be rewarded each month for the team developers to be rewarded every 3 days.

Recommendation

We recommend changing the value of `secondsPerMonth` to be `2_592_000`.

Alleviation

`[CertiK, 09/17/2023]`: The client made the recommended changes in commit: 04d99e3523957ccf47b736f3addcc4abea9b02c2.

CTB-01 | REWARDS FOR `teamDev` CAN BE RELEASED EARLY

Category	Severity	Location	Status
Logical Issue	Medium	contracts/CarboTokenv2.sol (base): 97~118	Partially Resolved

Description

The `latestUpdateForTeamDev` in the `constructor()` can be set to any initial value. In particular, it can be set to a value that is not a multiple of `secondsPerMonth`, which allows for rewards to be released earlier than expected.

In addition, `latestUpdateForTeamDev` is set in the `constructor()` with no upper or lower bounds. If this value is set to a low value accidentally, then it is possible for the `teamDev` to be rewarded the `maxTokenForDev` as soon as the contract is deployed. If this value is set to a value much higher than the `block.timestamp`, then the `teamDev` may not be eligible for rewards when they should.

Scenario

Scenario 1:

- For simplicity assume that the contract is deployed with `latestUpdateForTeamDev` set to `2_591_999` in the `constructor()` and is deployed when the current `block.timestamp` is less than `2_591_999`.
- `rewardForTeamDev()` is then called when the `block.timestamp = 2_592_000`, so that only 1 second has passed since the initial `latestUpdateForTeamDev`.
- However, in the calculation `tillTime = (block.timestamp / secondsPerMonth) = 1` while `fromTime = (latestUpdateForTeamDev / secondsPerMonth) = 0` as it will be rounded down. Thus the multiplier will be `1` causing the reward that should only be given after a month to be given out after 1 second.

Scenario 2:

- Assume that the contract is deployed with `latestUpdateForTeamDev` set to `0`.
- `rewardForTeamDev()` is then called with the current `block.timestamp`, which will cause the `maxTokenForDev` to be minted to the `teamDev`.

Recommendation

We recommend setting reasonable upper and lower bounds for the `latestUpdateForTeamDev` in the constructor and also checking it is a multiple of `secondsPerMonth`.

Alleviation

`[CertiK, 09/17/2023]`: The client added logic to ensure that `latestUpdateForTeamDev` is a multiple of seconds per month in commit: 4062b1f635a30d20b9fed77a4e90f708bde291fd.

However, no bounds were set so we mark this finding as *partially resolved* considering scenario 2 is still possible.

SCB-01 `emergencyWithdraw()` CAN TRANSFER USERS STAKED TOKENS

Category	Severity	Location	Status
Logical Issue	Medium	contracts/StakingCarbon.sol (base): 134~143	Resolved

Description

In the contract `stakingCarbon`, users stake `mainToken` in the contract. The `emergencyWithdraw()` function allows the `DEFAULT_ADMIN_ROLE` to withdraw any ERC20 token from the contract, including the staked `mainToken` of users.

Scenario

The address that has the `DEFAULT_ADMIN_ROLE` calls `emergencyWithdraw()` with the input address of `mainToken`. This then transfers the contracts balance to the `msg.sender` including all tokens that have been staked by users.

Recommendation

We recommend ensuring the `emergencyWithdraw()` function cannot withdraw tokens that have been staked by users.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in commit: ab604f54afa79ddc987bb5b31fd2afa8fb07a928.

CON-01 | MISSING ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	contracts/AirdropCarbonv2.sol (base): 26, 27; contracts/StakingCarbon.sol (base): 46	Resolved

Description

Addresses should be checked before assignment or external call to make sure they are not zero addresses.

```
26          carboV1Addr = _tokenV1;
```

- `_tokenV1` is not zero-checked before being used.

```
27          carboV2Addr = _tokenV2;
```

- `_tokenV2` is not zero-checked before being used.

```
46          mainToken = _mainToken;
```

- `_mainToken` is not zero-checked before being used.

Recommendation

We recommend adding a zero-check for the passed-in address value to prevent unexpected errors.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in the following commits:

- 4386676ce82a272de1d702262ad419b26a25fb94;
- 6ef876690b2821253db2237a13637987496650b2.

CON-03 | LOCKED ETHER

Category	Severity	Location	Status
Coding Issue	Minor	contracts/CarboTokenv2.sol (base): 33; contracts/StakingCarbon.sol (base): 40	Resolved

Description

The contracts `StakingCarbon` and `CarboTokenv2` have `payable` constructors allowing native tokens to accidentaly be sent when deploying the contract that will be locked in the contract.

Recommendation

We recommend removing the `payable` attribute.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in commit: 4386676ce82a272de1d702262ad419b26a25fb94.

CTB-02 | MINTING TO `address(1)`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/CarboTokenv2.sol (base): 39~43	Acknowledged

Description

On contract deployment `80_000_000` tokens are minted to `address(1)`, which is a null address and will cause those tokens to be unusable. As the tokens are not available, they should not be accounted for in the total supply. However, as they are minted to `address(1)`, they will be included in the total supply of the token.

Proof of Concept

The function `_mint()` from OpenZeppelin's ERC20 contract:

```
function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount;
        unchecked {
            // Overflow not possible: balance + amount is at most totalSupply +
amount, which is checked above.
            _balances[account] += amount;
        }
        emit Transfer(address(0), account, amount);

        _afterTokenTransfer(address(0), account, amount);
    }
```

Increases the total supply by the input `amount`.

Recommendation

We recommend removing this portion of code to ensure the total supply is reflective of the tokens in circulation.

Alleviation

`[CleanCarbon, 09/07/2023]`: Acknowledged. Minting to the null address was done on purpose, and we don't mind the changes not being accurately reflected in the token's total supply.

CON-04 | TYPOS

Category	Severity	Location	Status
Coding Style	Informational	contracts/AirdropCarbonv2.sol (base): 52; contracts/CarboTokenv2.sol (base): 18, 136; contracts/StakingCarbon.sol (base): 138	Resolved

Description

In the contract `CarboTokenv2`:

- The comment below `secondsPerMonth` is unnecessary and can be deleted.
- The comment in the function `emergencyWithdraw()` is unnecessary and can be deleted.

In the contract `AirdropCarbonv2`:

- The comment in the function `emergencyWithdraw()` is unnecessary and can be deleted.

In the contract `StakingCarbon`:

- The comment in the function `emergencyWithdraw()` is unnecessary and can be deleted.

Recommendation

We recommend fixing the typos mentioned above.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in commit: 4386676ce82a272de1d702262ad419b26a25fb94.

CON-05 | MISSING EMIT EVENTS

Category	Severity	Location	Status
Coding Style	Informational	contracts/AirdropCarbonv2.sol (base): 30, 48; contracts/CarboTokenv2.sol (base): 91, 128, 133; contracts/StakingCarbon.sol (base): 49, 134	Resolved

Description

There should always be events emitted in the sensitive functions that are controlled by centralization roles. The functions linked above do not emit events.

Recommendation

We recommend emitting events for the sensitive functions mentioned above.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in commit: 4386676ce82a272de1d702262ad419b26a25fb94.

CON-07 | UNCHECKED ERC-20 `transfer()` / `transferFrom()` CALL

Category	Severity	Location	Status
Volatile Code	Informational	contracts/AirdropCarbonv2.sol (base): 53~56; contracts/CarboTokenv2.sol (base): 137~140; contracts/StakingCarbon.sol (base): 139~142	Resolved

Description

The `emergencyWithdraw()` interacts with any possible ERC20 tokens. Since some ERC20 tokens return no values and others return a `bool` value, they should be handled with care.

Recommendation

We recommend using the OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if `false` is returned, making it compatible with all ERC-20 token implementations.

Alleviation

`[CertiK, 09/17/2023]`: The client made the recommended changes in commit: 59a6c5fbfd37a3609655944385f0649846e87e02.

CON-09 | `changeAdminRole()` RESTRICTION

Category	Severity	Location	Status
Coding Style	Informational	contracts/AirdropCarbonv2.sol (base): 30~33; contracts/CarboTokenv2.sol (base): 128~131; contracts/StakingCarbon.sol (base): 49~52	Acknowledged

Description

The function `changeAdminRole()` can only be called by the `ADMIN`. However, the `DEFAULT_ADMIN_ROLE` can still grant and revoke the `ADMIN` role through the `grantRole()` and `revokeRole()` functions as it is the admin of all roles by default. This in particular allows there to be multiple addresses with the `ADMIN` role.

Recommendation

We recommend considering the use of the `DEFAULT_ADMIN_ROLE` and the `grantRole()` and `revokeRole()` functions instead of the `changeAdminRole()` function.

Alleviation

`[CleanCarbon, 09/07/2023]`: Acknowledged. We have a super admin role to grant and revoke any roles, as it should make our internal workflow easier. In particular, some admins may want to transfer their roles to other wallets owned by them.

CTB-04 | TIME UNITS CAN BE USED DIRECTLY

Category	Severity	Location	Status
Coding Issue	Informational	contracts/CarboTokenv2.sol (base): 17	Resolved

Description

Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

- 1 == 1 seconds;
- 1 minutes == 60 seconds;
- 1 hours == 60 minutes;
- 1 days == 24 hours;
- 1 weeks == 7 days;

Recommendation

We recommend using `30 days` for `secondsPerMonth` to increase readability.

Alleviation

`[CleanCarbon, 09/07/2023]` : The team is used to work with specific data formats where time is defined in seconds, as it makes it easier to change values while testing. Changing to secondsPerMonth is not necessary.

GIT-01 | UNUSED PARAMETERS AND VARIABLES

Category	Severity	Location	Status
Coding Style	Informational	contracts/StakingCarbon.sol (update1): 32; contracts/CarboTokenv2.sol (base): 12, 30~31	Resolved

Description

In the contract `CarboTokenv2` there are paramaters and variables that are never used:

- In the `constructor()`, the parameters `buybacks` and `treasury` are never used.
- The variable `CONTRACT_MANAGER` is defined and never used.

Recommendation

We recommend either implementing or removing these parameters and variables.

Alleviation

`[CertiK, 09/17/2023]`: The client made the recommended changes in commits:

- 4386676ce82a272de1d702262ad419b26a25fb94;
- da29bd77cb376fc098d959cbcd1be65eac077252.

GIT-02 | CALLING VOID CONSTRUCTOR

Category	Severity	Location	Status
Coding Style	Informational	contracts/AirdropCarbonv2.sol (update2): 11; contracts/StakingCarbon.sol (update2): 10; contracts/AirdropCarbonv2.sol (base): 22; contracts/CarboTokenv2.sol (base): 33; contracts/StakingCarbon.sol (base): 41	Acknowledged

Description

Calling an undefined parent constructor has no effect. The `constructor()` of the contracts `AirdropCarbonv2`, `CarboTokenv2`, and `StakingCarbon` all call `AccessControl()` which does not have a defined constructor.

Recommendation

We recommend removing the constructor call.

Alleviation

`[CertiK, 09/17/2023]`: The client made the recommended changes in commit: e4cf33ea2feac7356a9a960e653f547db58ae237.

However, in doing so the constructor call to `ReentrancyGuard` was removed. While the constructor sets the default value, we still recommend calling all constructors that are not null. Similarly as `Pausable` was added, we recommend calling its constructor as well.

`[CleanVarbon, 09/18/2023]`: Issue acknowledged. We've decided to keep this as it is.

SCB-02 | `isActive` DISCUSSION

Category	Severity	Location	Status
Logical Issue	Informational	contracts/StakingCarbon.sol (base): 62	Resolved

Description

The `ADMIN` calls `addStakingPayload()`, to add additional staking options. This function would not be called unless the new staking option was intended to be active, as there is no functionality to change if a specific staking option is active. Thus, the `payload` does not need a parameter for `isActive` as it should always be true.

Recommendation

However, we believe it is possible that some staking options may want to be deprecated in the future. If this is the case we recommend instead adding functionality for the `ADMIN` to change if a staking option `isActive`. Note that if this functionality is added it should be considered if a user should be allowed to `unstake()` from a pool if it is inactive and their lock duration has not yet passed.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in commit: 4386676ce82a272de1d702262ad419b26a25fb94.

OPTIMIZATIONS | CLEANCARBON

ID	Title	Category	Severity	Status
ACB-01	`ADMIN` Role Not Used In `AirdropCarbonv2`	Logical Issue	Optimization	Resolved
CON-08	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	Acknowledged
CTB-06	Unnecessary Check	Gas Optimization	Optimization	Resolved
SCB-04	Can Use `delete` To Save Gas	Coding Style	Optimization	Acknowledged
SCC-03	Possibly Inefficient Memory Parameter	Gas Optimization	Optimization	Acknowledged

ACB-01 | `ADMIN` ROLE NOT USED IN `AirdropCarbonv2`

Category	Severity	Location	Status
Logical Issue	Optimization	contracts/AirdropCarbonv2.sol (base): 14	Resolved

Description

The `ADMIN` role is only used in the `AirdropCarbonv2` contract to restrict `changeAdminRole`. Since the contract does not use the `ADMIN` role to restrict access to any functions not directly related to the role itself, it can be removed.

Recommendation

We recommend removing the `ADMIN` role from this contract.

Alleviation

`[CertiK, 09/12/2023]`: The client updated the code to use the `ADMIN` role in commit: 62534fc5432b97dcf69c37ebae4d4e25bf094ac1.

CON-08 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/AirdropCarbonv2.sol (base): 10, 12; contracts/CarboTokenv2.sol (base): 8, 14, 20, 21; contracts/StakingCarbon.sol (base): 28	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.

Alleviation

`[CleanCarbon, 08/07/2023]`: Acknowledged. We don't want to change to immutable variable, in case we decide to change the contract to proxy in the future. Saving on gas is less important in this case.

CTB-06 | UNNECESSARY CHECK

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/CarboTokenv2.sol (base): 121~124	Resolved

Description

The check in the function `_mint()`, that `capSupply >= amount + totalSupply()` is unnecessary as the current implementation only allows a maximum of `500_000_000` tokens to be minted.

Proof of Concept

The function `_mint()` is only used in the `constructor()`, `releaseForAirdrop()`, and `rewardForTeamDev()`:

1. In the `constructor()`, `_mint()` is used to mint a total of `380_000_000` tokens.
2. In `releaseForAirdrop()`, this function can only be called once and mints a total of `90_000_000` tokens.
3. In `rewardForTeamDev()`, this function mints up to the `maxTokenForDev`, which is `30_000_000` tokens.

Thus the total amount of tokens that can be minted is `380_000_000 + 90_000_000 + 30_000_000 = 500_000_000` tokens.

Recommendation

We recommend removing this unnecessary check to save gas.

Alleviation

`[CertiK, 09/12/2023]`: The client made the recommended changes in commit: 4386676ce82a272de1d702262ad419b26a25fb94.

SCB-04 | CAN USE `delete` TO SAVE GAS

Category	Severity	Location	Status
Coding Style	Optimization	contracts/StakingCarbon.sol (base): 129	Acknowledged

Description

When a user calls `unstake()`, the `userStateStorage[msg.sender]` is set back to the default values of `0` by hand. This can instead be done using the `delete` operator saving around 4007 gas on deployment and 63 gas on each function call.

See the documentation on `delete` here: Solidity Delete Documentation.

Recommendation

We recommend using `delete` instead of setting the values to `0` by hand.

Alleviation

`[CleanCarbon, 09/07/2023]`: Issue acknowledged. I won't make any changes for the current version.

SCC-03 | POSSIBLY INEFFICIENT MEMORY PARAMETER

Category	Severity	Location	Status
Gas Optimization	Optimization	StakingCarbon.sol (0xabc60): 54	Acknowledged

Description

One or more parameters with `memory` data location are never modified in their functions and those functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from calldata to memory.

```
54      function addStakingPayload(StakingOptions memory payload)
```

`addStakingPayload` has memory location parameters: `payload`.

This change will increase the deployment cost by around `7813` gas, while saving around `61` gas on each function call.

Recommendation

We recommend changing the parameter's data location to `calldata` to save gas if the `addStakingPayload()` is expected to be called more than 128 times.

Alleviation

`[CleanCarbon, 09/07/2023]`: Acknowledged, but we decided this optimization is not needed.

FORMAL VERIFICATION | CLEANCARBON

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	`transfer` Prevents Transfers to the Zero Address
erc20-transfer-correct-amount	`transfer` Transfers the Correct Amount in Non-self Transfers
erc20-transfer-succeed-self	`transfer` Succeeds on Admissible Self Transfers
erc20-transfer-succeed-normal	`transfer` Succeeds on Admissible Non-self Transfers
erc20-transfer-correct-amount-self	`transfer` Transfers the Correct Amount in Self Transfers
erc20-transfer-change-state	`transfer` Has No Unexpected State Changes
erc20-transfer-exceed-balance	`transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transfer-false	If `transfer` Returns `false`, the Contract State Is Not Changed
erc20-transferfrom-revert-from-zero	`transferFrom` Fails for Transfers From the Zero Address
erc20-transfer-never-return-false	`transfer` Never Returns `false`

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

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

Verification Results

In the remainder of this section, we list all contracts where model checking of at least one property was not successful. There are several reasons why this could happen:

- Model checking reports a counterexample that violates the property. Depending on the counterexample,this occurs if
  - The specification of the property is too generic and does not accurately capture the intended behavior of the smart contract. In that case, the counterexample does not indicate a problem in the underlying smart contract. We report such instances as being "inapplicable".
  - The property is applicable to the smart contract. In that case, the counterexample showcases a problem in the smart contract and a correspond finding is reported separately in the Findings section of this report. In the following tables, we report such instances as "invalid". The distinction between spurious and actual counterexamples is done manually by the auditors.
- The model checking result is inconclusive. Such a result does not indicate a problem in the underlying smart contract. An inconclusive result may occur if
  - The model checking engine fails to construct a proof. This can happen if the logical deductions necessary are beyond the capabilities of the automated reasoning tool. It is a technical limitation of all proof engines and cannot be avoided in general.
  - The model checking engine runs out of time or memory and did not produce a result. This can happen if automatic abstraction techniques are ineffective or of the state space is too big.

Detailed Results For Contract CarboTokenv2 (contracts/CarboTokenv2.sol) In Commit 37268ef0ecfaf3f166707071830b41854b34a5ab

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-succeed-self	True	
erc20-transfer-succeed-normal	True	
erc20-transfer-correct-amount-self	True	
erc20-transfer-change-state	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	
erc20-transfer-recipient-overflow	False	Context not considered

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-self	True	
erc20-transferfrom-succeed-normal	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-never-return-false	True	
erc20-transferfrom-fail-recipient-overflow	False	Context not considered

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 | CLEANCARBON

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.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
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 `[]`) and "eventually" (written `<>`), 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.