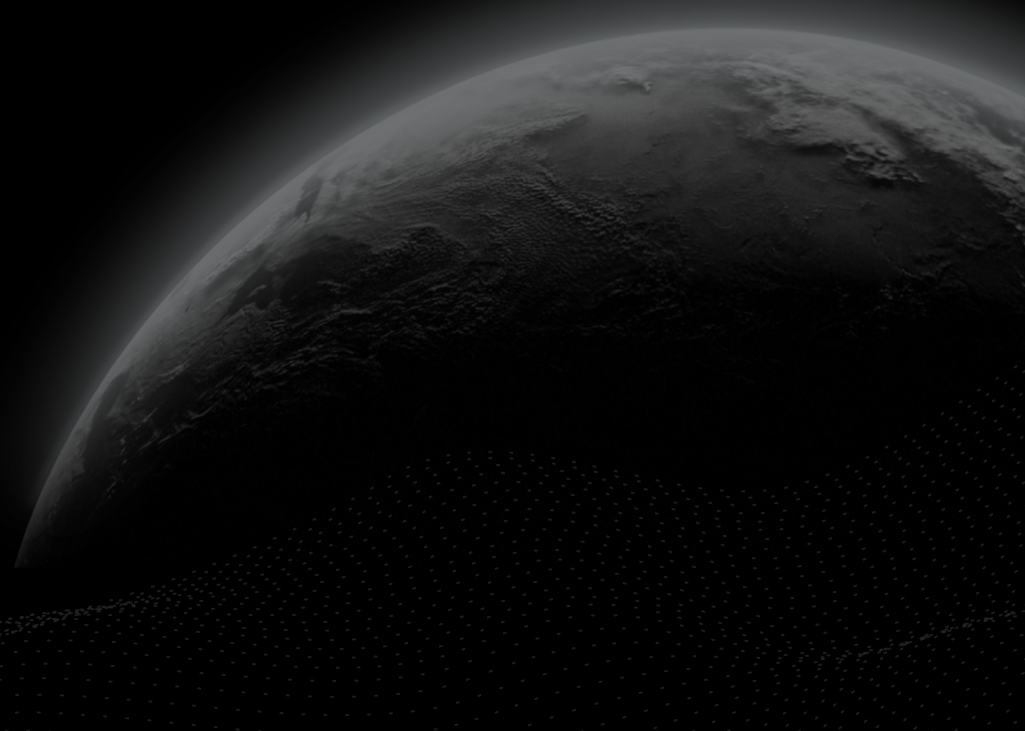
CERTIK

Security Assessment

fightOut - Audit

CertiK Verified on Dec 18th, 2022



CertiK Verified on Dec 18th, 2022

fightOut - Audit

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

Executive Summary

TYPES	ECOSYSTEM	METHODS
Platform	Other	Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 12/18/2022	N/A

CODEBASE

https://github.com/FightoutDev/ERC20Token/tree/30b285198b362664986d102086424a1de682f1b4

...View All

COMMITS

30b285198b362664986d102086424a1de682f1b4

...View All

Vulnerability Summary

2	0	1	1	0	0	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined	Unresolved

0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
1	Major	1 Mitigated	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.
0	Minor		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 Partially Resolved	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 | FIGHTOUT - AUDIT

CODEBASE | FIGHTOUT - AUDIT

Repository

https://github.com/FightoutDev/ERC20Token/tree/30b285198b362664986d102086424a1de682f1b4

Commit

30b285198b362664986d102086424a1de682f1b4

AUDIT SCOPE | FIGHTOUT - AUDIT

1 file audited ● 1 file with Acknowledged findings

ID	File	SHA256 Checksum
● TER	contracts/Token.sol	8c78b12eadeb447d93756b3304d3e219c571d78a7d4f8ca06a0cf4bc47a2f9ab

APPROACH & METHODS | FIGHTOUT - AUDIT

This report has been prepared for fightout to discover issues and vulnerabilities in the source code of the fightOut - Audit project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis 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 | FIGHTOUT - AUDIT

2	0	1	0	0	1
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for fightOut - Audit. Through this audit, we have uncovered 2 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
TER-01	**Initial Token Distribution**	**Centralization / Privilege**	**Major**	**Mitigated**
TER-02	Missing Zero Check On `amount`	Logical Issue	Informational	Partially Resolved

TER-01 | INITIAL TOKEN DISTRIBUTION

Category	Severity	Location	Status
Centralization / Privilege	**Major**	**contracts/Token.sol: 28**	**Mitigated**

Description

10 billion tokens are sent to the address `_to` when deploying the contract. This could be a centralization risk as the `_to` account can distribute tokens without obtaining the consensus of the community.

Recommendation

We recommend the team to be transparent regarding the initial token distribution process, and the team shall make enough efforts to restrict the access of the private key.

Alleviation

The team acknowledged the issue and stated the following:

"All 10 billion tokens need to be sent to a single wallet as, from there, they will be distributed into the appropriate pools.

We will send between 6 and 9 billion of those tokens to the presale smart contract, where they will vest according to the schedule set out in the presale contract to each user who bought.

1 Billion of the tokens will be accessible for exchange listings.

The tokens that were not given away as bonuses in the presale (between 0 and 3 billion) will be put into a Unicrypt wallet and will vest over time to be used for community incentives."

whitepaper

TER-02 | MISSING ZERO CHECK ON `amount`

Category	Severity	Location	Status
Logical Issue	Informational	contracts/Token.sol: 57, 87, 132	Partially Resolved

Description

Functions `transfer()` and `transferFrom()` need to check if `amount > 0` to avoid meaningless transactions.

Recommendation

We recommend the team add zero checks for `amount` in functions `transfer()` and `transferFrom()`.

Alleviation

The team heeded the advice and the changes have been reflected in the commit hash 217b844c0ed565d8ab12da61455f48f6b21b95c1.

OPTIMIZATIONS | FIGHTOUT - AUDIT

ID	Title	Category	Severity	Status
TER-03	Variable Can Be Declared As Immutable	Gas Optimization	Optimization	Acknowledged

TER-03 | VARIABLE CAN BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/Token.sol: 15~16	Acknowledged

Description

The variable **_name**, **_symbol** 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. An advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they are not located in storage.

Recommendation

We recommend declaring state variable **_name**, **_symbol** as immutable.

Alleviation

The team acknowledged the finding and decided to remain unchanged.

FORMAL VERIFICATION | FIGHTOUT - AUDIT

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

Verification Results

For the following contracts, model checking established that each of the 38 properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract Token (contracts/Token.sol)

Verification of ERC-20 compliance

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 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 `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-change-state	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-correct-amount-self	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-change-state	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-fail-recipient-overflow	True	
erc20-transferfrom-never-return-false	True	

Detailed results for function `totalSupply`

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

APPENDIX | FIGHTOUT - AUDIT

Finding Categories

Categories	Description
Centralization / Privilege	Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

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.