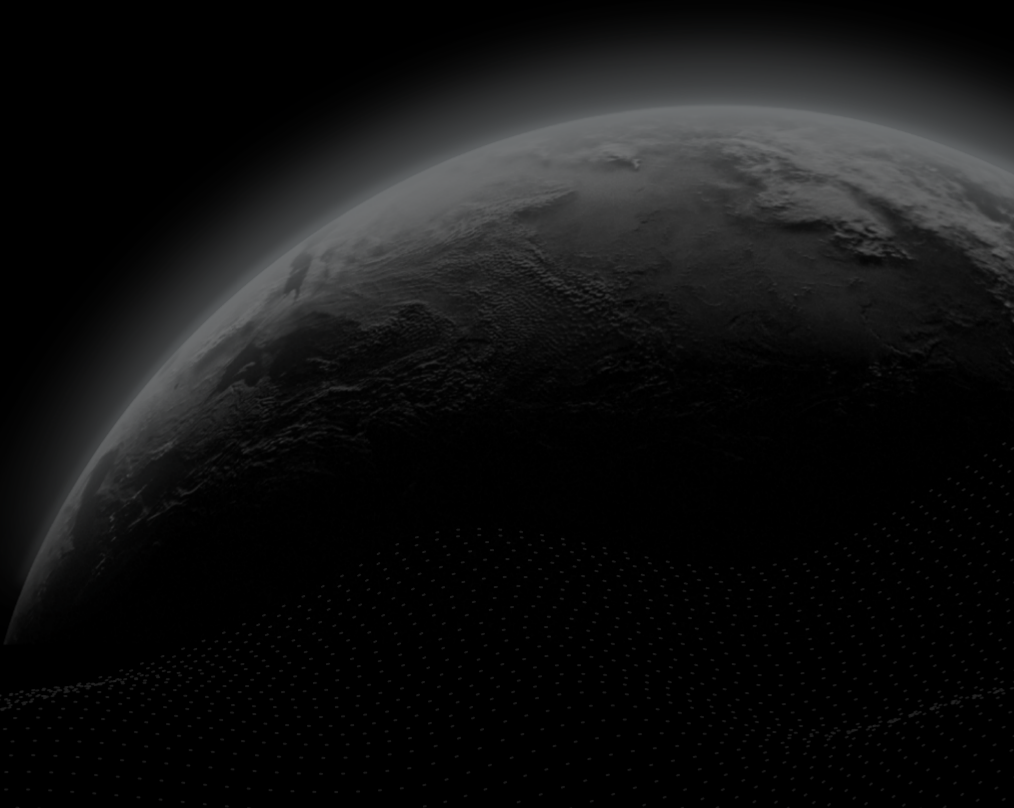
CERTIK

Security Assessment

Numoen

CertiK Verified on Dec 7th, 2022



CertiK Verified on Dec 7th, 2022

Numoen

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

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Ethereum	Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 12/07/2022	N/A

CODEBASE

https://github.com/Numoen/core

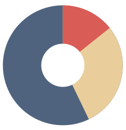
...View All

COMMITS

- ebab0a4730d055492aae40077a448c60bac446d1
- 714d95dce13f92192f77821c8c7ae8598b25f4ad
- 17fcc5629d7cb3b25a9e8ec832b1098a5c7b3424

...View All

Vulnerability Summary



7	3	0	1	3	0	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined	Unresolved

1	Critical	1 Partially Resolved	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.
0	Major		Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
0	Medium		Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
2	Minor	2 Resolved	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
4	Informational	1 Resolved, 3 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 | NUMOEN

CODEBASE | NUMOEN

Repository

https://github.com/Numoen/core

Commit

- ebab0a4730d055492aae40077a448c60bac446d1
- 714d95dce13f92192f77821c8c7ae8598b25f4ad
- 17fcc5629d7cb3b25a9e8ec832b1098a5c7b3424
- 8b40655365cd883140b70a3f16a2f2e916807786
- 9144c06b0d633f4a6e3b4bcf6e085cf1f04ebb2e

AUDIT SCOPE | NUMOEN

14 files audited ● 8 files with Acknowledged findings ● 1 file with Partially Resolved findings ● 2 files with Resolved findings ● 3 files without findings

ID	File	SHA256 Checksum
IFN	interfaces/IFactory.sol	4394625cae77627b8a55ccbde686fb5822e473c880c7cb1d1dfd9204e44adff0
IJR	interfaces/IJumpRate.sol	008df9f8e3a00c013a9889add3cc1fc05085e62820372a70e7ae4dbf64a15783
ILN	interfaces/ILendgine.sol	efade1f4f4eadcf01982b054226ad354ab4e1e2aa98a049858da3852b4d3c046
IMC	interfaces/IMintCallback.sol	d7a83bcfef0ebf457fa3eac9fd82f1f49faad4e505e6067b43a6e5ba81223021
IPN	interfaces/IPair.sol	d3c74760392751cc8959d1b3df3653b5c1305d05f6788ca7b4d46e9ed5da1dce
LAN	libraries/LendgineAddress.sol	1f3ce391c3004397fdedc90956c61c090ca6ad367b736446cc2a888b52930e30
LMN	libraries/LiquidityMath.sol	8d1fd7dd1befdc64bb394659e7d55a35cf852245d13e62b92fed077de266b884
LNB	Lendgine.sol	a43c3f1f37c266276c3ec46a684dfd6fccb4c959c3ca0a976106590c184625d8
PNB	Pair.sol	182421be19489c871a5efe5d2626b1838a8c768477dcc306d5b2c728bb4c6f35
ERC	ERC20.sol	e7dc7586c9214b6193ac8895b82eb6d527313fe57687b052328f61fd0e379f25
FNB	Factory.sol	c0bf28e772410f000fc5cb4348ab3edfe4f3f5fac5b4ad40a2eb6e7d4d87079e
PNU	libraries/Position.sol	635ffa30b11d0d0b44fd335a8fd5b1bcca491fb78b16e9fe3f65a7fdf9016d0e
STL	libraries/SafeTransferLib.sol	8127818ee2a30f648089a790a0fa37317bd46e5d98bd6ac2dcc310a42d230e18
JRN	JumpRate.sol	3359b15abcf7770325422bf448e5a562435d9d60b99df5bcd43dd0ddd64cef9d

APPROACH & METHODS | NUMOEN

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

REVIEW NOTES | NUMOEN

Overview

Numoen is a DeFi project that focuses on creating a permissionless options exchange. The platform offers automated liquidity provisioning and exchange of perpetual options directly on the blockchain through smart contracts.

External Dependencies

Contracts

The main external dependency the project relies on is the **PRBMath library** that handles advanced fixed-point math that operates with signed 59.18-decimal fixed-point and unsigned 60.18-decimal fixed-point numbers: https://github.com/paulrberg/prb-math.

The following contracts are referenced in various contracts:

- `PRBMath.sol` & `PRBMathUD60x18.sol`

Addresses

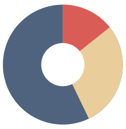
The following are external addresses used within the contracts:

Lendgine.sol

- `IMintCallback` contract address

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

FINDINGS | NUMOEN



7	1	0	0	2	4
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Numoen. Through this audit, we have uncovered 7 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
ERC-01	Function `transfer` Of Contract `ERC20` Does Not Prevent Transfers To The Zero Address	Logical Issue	Minor	Resolved
PNB-01	Curve Model Could Lead To Arbitrage	Logical Issue, Mathematical Operations	Critical	Partially Resolved
SRC-01	Custom Errors Require Solidity Version 0.8.4 And Above	Volatile Code	Minor	Resolved
LNB-01	Potential Front-Running Risk	Volatile Code	Informational	Acknowledged
PNU-01	Potential Loss Of Precision Could Lead To Revert On Invariant Check	Mathematical Operations	Informational	Resolved
SRC-02	Unlocked Compiler Version	Language Specific	Informational	Acknowledged
SRC-03	Payment For Minting Tokens	Logical Issue	Informational	Acknowledged

ERC-01 | FUNCTION `transfer` OF CONTRACT `ERC20` DOES NOT PREVENT TRANSFERS TO THE ZERO ADDRESS

Category	Severity	Location	Status
Logical Issue	Minor	ERC20.sol (Original PR): 69~81	Resolved

Description

It is expected that calls of the form `transfer(recipient, amount)` fail if the recipient address is the zero address.

Recommendation

Ensure that the `transfer` function in contract `ERC20` reverts if invoked with a recipient address of zero.

PNB-01 | CURVE MODEL COULD LEAD TO ARBITRAGE

Category	Severity	Location	Status
Logical Issue, Mathematical Operations	Critical	Pair.sol (Original PR)	Partially Resolved

Description

In the pair contract, swapping tokens/mint(add liquidity)/burn(remove liquidity) operations need to check the invariant to be consistent. In the following is a code snippet that goes through the current implementation of the `verifyInvariant()` function.

```
if (!verifyInvariant(balance0, balance1, liquidity + totalSupply)) revert
InvariantError();
```

```
function verifyInvariant(
        uint256 r0,
        uint256 r1,
        uint256 shares
    ) public view returns (bool valid) {
        uint256 scale0 = PRBMathUD60x18.div(PRBMathUD60x18.div(r0, shares),
10**baseScaleFactor);
        uint256 scale1 = PRBMathUD60x18.div(PRBMathUD60x18.div(r1, shares),
10**speculativeScaleFactor);

        uint256 a = scale0;
        uint256 b = PRBMathUD60x18.mul(scale1*, upperBound);
        uint256 c = PRBMathUD60x18.powu(scale1, 2) / 4;
        uint256 d = PRBMathUD60x18.powu(upperBound, 2);

        if (scale1 > 2 * upperBound) revert SpeculativeInvariantError();

        return a + b == c + d;
}
```

The above code implements the following the following equation: $a + b = c + d$

The above equation can be simplified as following steps:

Step 1: $scale0 + scale1 * upperBound = \frac{scale1^2}{4} + upperBound^2$

Step 2: $scale0 = \frac{scale1^2}{4} - scale1 * upperBound + upperBound^2$

Step 3: $scale0 = (\frac{scale1}{2} - upperBound)^2$

The above formula will be part of the curve between [0, 2*upperBound]. However, this model could lead to potential economic attacks. This is because when adding liquidity or removing liquidity, users can adjust the expected output amount of base tokens or speculative tokens to gain more profit. In the following example, an attacker can mint 1 ETH of liquidity and burn 1 ETH of liquidity to gain profits because he can withdraw more tokens from the liquidity than he would normally be allowed.

Proof of concept

1. Initially, 1 ETH of liquidity is added to the 16 ETH base tokens and 2 speculative tokens. The assumed price of speculative tokens 8.
2. The attacker can add another 1 ETH of liquidity to the collection of 16 ETH base tokens and 2 speculative tokens.
3. Attacker burns his 1 ETH of liquidity along with 9 ETH base tokens and 4 speculative tokens.
4. Since the price of speculative tokens is 8, the attacker actually gains a profit of 7 additional base tokens.

```
    function testAttackOnCurveModel() public {
        // Initially, there is 1 liquidity with 16 base tokens and 2 speculative
tokens
        // Suppose current market price is 1:8
        _pairMint(16 ether, 2 ether, 1 ether, cuh);

        //The attacker added 1 liquidity another 16 base tokens and 2 speculative
tokens
        _pairMint(16 ether, 2 ether, 1 ether, cuh);
        assertEq(pair.totalSupply(), 2 ether);
        uint baseTokenBalanceBefore = base.balanceOf(address(cuh));
        uint speculativeTokenBalanceBefore = speculative.balanceOf(address(cuh));

        //The attacker can actually withdraw much more tokens than he added as
liquidity
        pair.burn(cuh, 9 ether , 4 ether , 1 ether);
        assertEq(pair.totalSupply(), 1 ether);
        uint baseTokenBalanceAfter = base.balanceOf(address(cuh));
        uint speculativeTokenBalanceAfter = speculative.balanceOf(address(cuh));

        assertEq(baseTokenBalanceAfter - baseTokenBalanceBefore, 9 ether);
        assertEq(speculativeTokenBalanceAfter - speculativeTokenBalanceBefore, 4
ether);
    }
```

Result

The tests pass which means that the attack is successful based on the current model.
[PASS] testAttackOnCurveModel() (gas: 272404)

[CertiK, 11/14/2022]:

This model does have fancy features and can restrict the boundary of the price. However, due to the decentralized feature of blockchain, those tokens trading/swapping with the pair contract may lead to inconsistency issues and thus cause arbitrage risk.

Here are two main concerns regarding the curve model:

1. When the token is also available in AMM like UniSwap/PancakeSwap using xy=k as their curve model, the inconsistency with other AMMs could lead to arbitrage due to different models used (see the above example).
2. The curve itself is not consistent regarding the swapping process (see the following two testcases).

```
    // case 1: swap 2 ether speculative tokens with 9 ether base tokens
    function testSwapOnCurveModel() public {
        // Initially, there is 1 liquidity with 16 base tokens and 2 speculative
tokens
        _pairMint(16 ether, 2 ether, 1 ether, cuh);

        // To swap 2 ether speculative tokens out, 9 ether base token are required
        base.mint(cuh, 9 ether);
        vm.prank(cuh);
        base.transfer(address(pair),9 ether);

        uint speculativeBalanceBefore = speculative.balanceOf(address(cuh));
        pair.swap(cuh, 0, 2 ether);
        uint speculativeBalanceAfter = speculative.balanceOf(address(cuh));

        // In normal cases,
        assertEq(speculativeBalanceAfter - speculativeBalanceBefore, 2 ether);
    }

    // case 2: swap 2 ether speculative tokens with 7 ether base tokens
    function testAttackOnCurveModel() public {
        // Initially, there is 1 liquidity with 16 base tokens and 2 speculative
tokens
        _pairMint(16 ether, 2 ether, 1 ether, cuh);

        //The attacker added 1 liquidity another 16 base tokens and 2 speculative
tokens
        _pairMint(16 ether, 2 ether, 1 ether, cuh);
        assertEq(pair.totalSupply(), 2 ether);
        uint baseTokenBalanceBefore = base.balanceOf(address(cuh));
        uint speculativeTokenBalanceBefore = speculative.balanceOf(address(cuh));

        //The attacker can withdraw 4 ether speculative tokens and 9 base tokens out
        //This means the attacker uses 7 ether base tokens swapping for 2 ether
speculative tokens
        pair.burn(cuh, 9 ether , 4 ether , 1 ether);
        assertEq(pair.totalSupply(), 1 ether);
        uint baseTokenBalanceAfter = base.balanceOf(address(cuh));
        uint speculativeTokenBalanceAfter = speculative.balanceOf(address(cuh));

        // 16 ether base tokens and 2 speculative tokens are added as liquidity
        assertEq(baseTokenBalanceAfter - baseTokenBalanceBefore, 9 ether);
        assertEq(speculativeTokenBalanceAfter - speculativeTokenBalanceBefore, 4
ether);
    }
```

In the first test case, when the pool has 16 base tokens, 2 speculative tokens, and 1 ether liquidity. By using the normal swap function, it needs 9 ether base tokens to swap all the 2 ether speculative tokens from the pool.

However, in the second test case, by using the `burn()` function, the attacker can use only 7 ether tokens (16-9=7) to "swap" 2 speculative tokens from the pool.

This is due to the curve model which ensures the invariant is the same. However, due to the nature of the curve, users can actually choose different strategies to "swap"/"remove liquidity"/"add liquidity", but those strategies are not equivalent in terms of the movement on the curve.

Recommendation

Considering many unknown circumstance on chain (such as the presence of MEV bots), this issue might be even more severe and lead to loss of client funds. Therefore, it is recommended to use more battle-tested curve models like Uniswap (i.e., `x*y=k`) to avoid such attacks on the curve model.

Alleviation

[Numoen]: The team heeded the advice and partially resolved this issue in commits 1b6e2e42eda549e837d816417ac6e6353b9eda9a. The `burn` function was updated to remove input parameters for amounts which removes the ability of users to choose arbitrary points on the curve to withdraw at. Additionally, the team stated that it is intended to create arbitrage opportunities with external platforms like Uniswap.

[Certik]: There are still issues when the prices might not match with those of external prices such as Uniswap, which still creates an environment for arbitrage.

[Numoen, 11/18/2022]: The team is aware of the arbitrage and stated it is the **designed feature** of the curve.

[Certik, 11/18/2022]: Auditors understand and respect the design of the project. It is worth mentioning that the arbitrage might lead to a potential loss for liquidity providers.

SRC-01 | CUSTOM ERRORS REQUIRE SOLIDITY VERSION 0.8.4 AND ABOVE

Category	Severity	Location	Status
Volatile Code	Minor	Factory.sol (Original PR): 2; Lendgine.sol (Original PR): 2; Pair.sol (Original PR): 2	Resolved

Description

An issue relating to custom error messages used in contracts can occur with the current solidity compiler settings.

The solidity version is currently set to `pragma solidity ^0.8.0` , however the solidity version needs to be at least 0.8.4 or above to support custom errors.

Recommendation

Recommend setting the solidity version to at least 0.8.4 or above to prevent unexpected compiler errors related to the use of custom error messages.

Alleviation

[Numoen]: The team heeded the advice and resolved this issue in commit ebab0a4730d055492aae40077a448c60bac446d1.

LNB-01 | POTENTIAL FRONT-RUNNING RISK

Category	Severity	Location	Status
Volatile Code	Informational	Lendgine.sol (Original PR)	Acknowledged

Description

To deposit into the contract, users are supposed to call `mint()` to add liquidity. However, if the `mint()` and `deposit()` calls are invoked in separate transactions, it might lead to front-run attack. In this situation, an attacker can front run the original `deposit()` with his own address. Therefore, the original user might lose his liquidity forever.

Recommendation

Recommend that the team review the design and ensure that checks and controls are put in place to prevent front running attacks from occurring.

Alleviation

[Numoen]: The team acknowledged the finding and did not make any changes related to this finding. The team clarified that `numoen-core` is intended to be called through another smart contract that has front-running checks in place. These contracts are not under the scope of the current audit.

PNU-01 | POTENTIAL LOSS OF PRECISION COULD LEAD TO REVERT ON INVARIANT CHECK

Category	Severity	Location	Status
Mathematical Operations	Informational	Pair.sol (11/23/2022): 161~163	Resolved

Description

The loss of precision on the calculation in L161~162 could cause the invariant check revert.

```
161         uint256 amount0 = PRBMath.mulDiv(r0, liquidity, _totalSupply);
162         uint256 amount1 = PRBMath.mulDiv(r1, liquidity, _totalSupply);
163         if (!verifyInvariant(amount0, amount1, liquidity)) revert
InvariantError();
```

The `verifyInvariant()` requires the exact number of two tokens to be matched according to the formula. However, the loss of precision could cause the two token's amount to be unmatched, thus causing the revert on the invariant check.

Recommendation

If the economy model allows, allowing imprecision when checking the invariant and ensure the invariant are within certain range.

Alleviation

[Numoen]: The team resolved this issue by updating the invariant check from `a + b == c + d` to `a + b >= c + d` in the commit 9144c06b0d633f4a6e3b4bcf6e085cf1f04ebb2e.

[CertiK]: It is worth mentioning that attackers can exploit imprecise invariant checks to manipulate the price, which could have potential attack vectors to project's components that are out of scope.

[Numoen]: The team confirmed it is the intended design, and other components will not be affected.

SRC-02 | UNLOCKED COMPILER VERSION

Category	Severity	Location	Status
Language Specific	● Informational	interfaces/IFactory.sol (Original PR): 2; interfaces/IJumpRate.sol (Original PR): 2; interfaces/ILendgine.sol (Original PR): 2; interfaces/IMintCallback.sol (Original PR): 2; interfaces/IPair.sol (Original PR): 2; libraries/LendgineAddress.sol (Original PR): 2; libraries/LiquidityMath.sol (Original PR): 2	● Acknowledged

Description

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to different compiler versions. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation

It is advised to use a compiler version that is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.8.4` the contract should contain the following line:

```
pragma solidity 0.8.4;
```

Alleviation

[Numoen]: The team acknowledged the finding and decided not to make any changes related to this issue.

SRC-03 | PAYMENT FOR MINTING TOKENS

Category	Severity	Location	Status
Logical Issue	Informational	Lendgine.sol (Original PR): 136; interfaces/IMintCallback.sol (Original PR): 9	Acknowledged

Description

From the analysis of the contracts, the minting and burning of NRD tokens within the `Lendgine.sol` contract seem to be open to any user to access. No centralized roles have been put in place to control this part of the process. There seems to be a mechanism in place that controls payment in exchange for the minting of tokens.

Within the `mint()` function, there is a call to the `msg.sender` which is supposed to be a contract address that contains the function `MintCallback()` .

```
136    IMintCallback(msg.sender).MintCallback(amountS, data);
```

Upon further investigation, the `mint()` function is actually calling an implementation of the interface `IMintCallback` .

```
9    function MintCallback(uint256 amount0, bytes calldata data) external;
```

Recommendation

Recommend that the team review the design and implementation of the `IMintCallback` interface to ensure that any external contracts calling the `MintCallback()` function are working properly.

Alleviation

[Numoen]: The team mentioned that a managing contract is called to control the calling of the `MintCallback()` function.

OPTIMIZATIONS | NUMOEN

ID	Title	Category	Severity	Status
ERC-02	Function Should Be Declared External	Gas Optimization	Optimization	Resolved

ERC-02 | FUNCTION SHOULD BE DECLARED EXTERNAL

Category	Severity	Location	Status
Gas Optimization	Optimization	ERC20.sol (Original PR): 61, 69, 83, 109	Resolved

Description

The functions which are never called internally within the contract should have external visibility for gas optimization.

ERC20.sol

```
61      function approve(address spender, uint256 amount) public virtual returns
(bool) {
```

```
69      function transfer(address to, uint256 amount) public virtual returns (bool)
{
```

```
83      function transferFrom(
```

```
109      function permit(
```

Recommendation

Recommend reviewing the functions to change the visibility of any functions not used internally to visibility `external` .

Alleviation

[Numoen]: The team heeded the advice and resolved this issue in commit ebab0a4730d055492aae40077a448c60bac446d1.

FORMAL VERIFICATION | NUMOEN

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

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-exceed-balance	Function `transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transfer-recipient-overflow	Function `transfer` Prevents Overflows in the Recipient's Balance
erc20-transferfrom-fail-exceed-balance	Function `transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-fail-recipient-overflow	Function `transferFrom` Prevents Overflows in the Recipient's Balance

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.

Contract ERC20 (Source File src/ERC20.sol)

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-exceed-balance	● Inapplicable	Incorrect finding
erc20-transfer-recipient-overflow	● Inapplicable	Incorrect finding

Detailed results for function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-fail-exceed-balance	● Inapplicable	Incorrect finding
erc20-transferfrom-fail-recipient-overflow	● Inapplicable	Incorrect finding

APPENDIX | NUMOEN

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.
Mathematical Operations	Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.
Language Specific	Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

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-deterministicly (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.