CERTIK

Smart Contract Security Audit Report

TheStake.io

The Certik Security Team received the team's application for smart contract security audit of the TheStake(GRND) on 2023.12.08. The following are the details and results of this smart contract security audit:

Project Name :

TheStake(GRND)

The contract address :

TheStake: https://scope.klaytn.com/

Logic: https://scope.klaytn.com/

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability	Passed
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed

NO.	Audit Items	Result
12	Scoping and Declarations Audit	Passed
13	Safety Design Audit	Passed
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002209130001

Audit Date : 2023.12.08 - 2023.12.13

Audit Team : Certik Security Team

Summary conclusion : This is a contract that contains the Upgradeable section and does not contain the dark hidden functions. The total amount of contract functions remains unchangeable. SafeMath security module is used, which is a recommended approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. The admin can't upgrade this contract to a new Implementation address through the UpgradeTo function.

The source code:

TransparentUpgradeableTheStake.sol

```
// File: @TheStake/contracts/proxy/TheStake.sol

// SPDX-License-Identifier: MIT
//Certik// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.0;

/**
 * @dev This abstract contract provides a fallback function that delegates all calls
to another contract using the EVM
 * instruction `delegatecall`. We refer to the second contract as the
_implementation_ behind the proxy, and it has to
 * be specified by overriding the virtual {_implementation} function.
 *
 * Additionally, delegation to the implementation can be triggered manually through
the {_fallback} function, or to a
 * different contract through the {_delegate} function.
 *
 * The success and return data of the delegated call will be returned back to the
caller of the proxy.
 */
abstract contract TheStake {
    /**
     * @dev Delegates the current call to `implementation`.
     *
     * This function does not return to its internall call site, it will return
directly to the external caller.
     */
    function _delegate(address implementation) internal virtual {
        // solhint-disable-next-line no-inline-assembly
        assembly {
            // Copy msg.data. We take full control of memory in this inline assembly
            // block because it will not return to Solidity code. We overwrite the
            // Solidity scratch pad at memory position 0.
            calldatacopy(0, 0, calldatasize())

            // Call the implementation.
            // out and outsize are 0 because we don't know the size yet.
            let result := delegatecall(gas(), implementation, 0, calldatasize(), 0,
0)

            // Copy the returned data.
            returndatacopy(0, 0, returndatasize())

            switch result
            // delegatecall returns 0 on error.
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }

    /**
     * @dev This is a virtual function that should be overriden so it returns the
address to which the fallback function
     * and {_fallback} should delegate.
     */
    function _implementation() internal view virtual returns (address);
    /**
     * @dev Delegates the current call to the address returned by
`_implementation()`.
     *
     * This function does not return to its internall call site, it will return
directly to the external caller.
     */
    function _fallback() internal virtual {
        _beforeFallback();
        _delegate(_implementation());
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by
`_implementation()`. Will run if no other
     * function in the contract matches the call data.
     */
    fallback () external payable virtual {
        _fallback();
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by
`_implementation()`. Will run if call data
     * is empty.
     */
    receive () external payable virtual {
        _fallback();
    }

    /**
     * @dev Hook that is called before falling back to the implementation. Can happen
as part of a manual `_fallback`
     * call, or as part of the Solidity `fallback` or `receive` functions.
     *
     * If overriden should call `super._beforeFallback()`.
     */
    function _beforeFallback() internal virtual {
    }
}

// File: @openzeppelin/contracts/proxy/beacon/IBeacon.sol
```

```
pragma solidity ^0.8.0;

/**
 * @dev This is the interface that {BeaconTheStake} expects of its beacon.
 */
interface IBeacon {
    /**
     * @dev Must return an address that can be used as a delegate call target.
     *
     * {BeaconTheStake} will check that this address is a contract.
     */
    function implementation() external view returns (address);
}

// File: @openzeppelin/contracts/utils/Address.sol



pragma solidity ^0.8.0;

/**
 * @dev Collection of functions related to the address type
 */
library Address {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects-interactions-pattern[checks-effects-interactions pattern].
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     * Returns the raw returned data. To convert to the expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?
highlight=abi.decode#abi-encoding-and-decoding-functions[`abi.decode`].
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes
memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but
with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data, string memory
errorMessage) internal returns (bytes memory) {
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value)
internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call
with value failed");
```

```
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}
[`functionCallWithValue`], but
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value,
string memory errorMessage) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for
call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view
returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call
failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}
[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data, string memory
errorMessage) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.4._
     */
    function functionDelegateCall(address target, bytes memory data) internal returns
(bytes memory) {
        return functionDelegateCall(target, data, "Address: low-level delegate call
failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}
[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.4._
     */
    function functionDelegateCall(address target, bytes memory data, string memory
errorMessage) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory
errorMessage) private pure returns(bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via
assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}

// File: @openzeppelin/contracts/utils/StorageSlot.sol



pragma solidity ^0.8.0;

/**
 * @dev Library for reading and writing primitive types to specific storage slots.
 *
 * Storage slots are often used to avoid storage conflict when dealing with
upgradeable contracts.
 * This library helps with reading and writing to such slots without the need for
inline assembly.
 *
 * The functions in this library return Slot structs that contain a `value` member
that can be used to read or write.
 *
 * Example usage to set ERC1967 implementation slot:
 * ```
 * contract ERC1967 {
 *     bytes32 internal constant _IMPLEMENTATION_SLOT =
0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
 *
 *     function _getImplementation() internal view returns (address) {
 *         return StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value;
 *     }
 *
 *     function _setImplementation(address newImplementation) internal {
 *         require(Address.isContract(newImplementation), "ERC1967: new
implementation is not a contract");
 *         StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value =
newImplementation;
 *     }
 * }
```

```
 * ```
 *
 * _Available since v4.1 for `address`, `bool`, `bytes32`, and `uint256`._
 */
library StorageSlot {
    struct AddressSlot {
        address value;
    }

    struct BooleanSlot {
        bool value;
    }

    struct Bytes32Slot {
        bytes32 value;
    }

    struct Uint256Slot {
        uint256 value;
    }

    /**
     * @dev Returns an `AddressSlot` with member `value` located at `slot`.
     */
    function getAddressSlot(bytes32 slot) internal pure returns (AddressSlot storage
r) {
        assembly {
            r.slot := slot
        }
    }

    /**
     * @dev Returns an `BooleanSlot` with member `value` located at `slot`.
     */
    function getBooleanSlot(bytes32 slot) internal pure returns (BooleanSlot storage
r) {
        assembly {
            r.slot := slot
        }
    }

    /**
     * @dev Returns an `Bytes32Slot` with member `value` located at `slot`.
     */
```

```
    function getBytes32Slot(bytes32 slot) internal pure returns (Bytes32Slot storage
r) {
        assembly {
            r.slot := slot
        }
    }

    /**
     * @dev Returns an `Uint256Slot` with member `value` located at `slot`.
     */
    function getUint256Slot(bytes32 slot) internal pure returns (Uint256Slot storage
r) {
        assembly {
            r.slot := slot
        }
    }
}

// File: @openzeppelin/contracts/proxy/ERC1967/ERC1967Upgrade.sol



pragma solidity ^0.8.2;



/**
 * @dev This abstract contract provides getters and event emitting update functions
for
 * https://eips.ethereum.org/EIPS/eip-1967[EIP1967] slots.
 *
 * _Available since v4.1._
 *
 * @custom:oz-upgrades-unsafe-allow delegatecall
 */
abstract contract ERC1967Upgrade {
    // This is the keccak-256 hash of "eip1967.proxy.rollback" subtracted by 1
    bytes32 private constant _ROLLBACK_SLOT =
0x4910fdfa16fed3260ed0e7147f7cc6da11a60208b5b9406d12a635614ffd9143;

    /**
     * @dev Storage slot with the address of the current implementation.
     * This is the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1,
and is
```

```
     * validated in the constructor.
     */
    bytes32 internal constant _IMPLEMENTATION_SLOT =
0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;

    /**
     * @dev Emitted when the implementation is upgraded.
     */
    event Upgraded(address indexed implementation);

    /**
     * @dev Returns the current implementation address.
     */
    function _getImplementation() internal view returns (address) {
        return StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value;
    }

    /**
     * @dev Stores a new address in the EIP1967 implementation slot.
     */
    function _setImplementation(address newImplementation) private {
        require(Address.isContract(newImplementation), "ERC1967: new implementation
is not a contract");
        StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value = newImplementation;
    }

    /**
     * @dev Perform implementation upgrade
     *
     * Emits an {Upgraded} event.
     */
    function _upgradeTo(address newImplementation) internal {
        _setImplementation(newImplementation);
        emit Upgraded(newImplementation);
    }

    /**
     * @dev Perform implementation upgrade with additional setup call.
     *
     * Emits an {Upgraded} event.
     */
    function _upgradeToAndCall(address newImplementation, bytes memory data, bool
forceCall) internal {
        _setImplementation(newImplementation);
        emit Upgraded(newImplementation);
        if (data.length > 0 || forceCall) {
            Address.functionDelegateCall(newImplementation, data);
        }
    }

    /**
     * @dev Perform implementation upgrade with security checks for UUPS proxies, and
additional setup call.
     *
     * Emits an {Upgraded} event.
     */
    function _upgradeToAndCallSecure(address newImplementation, bytes memory data,
bool forceCall) internal {
        address oldImplementation = _getImplementation();

        // Initial upgrade and setup call
        _setImplementation(newImplementation);
        if (data.length > 0 || forceCall) {
            Address.functionDelegateCall(newImplementation, data);
        }

        // Perform rollback test if not already in progress
        StorageSlot.BooleanSlot storage rollbackTesting =
StorageSlot.getBooleanSlot(_ROLLBACK_SLOT);
        if (!rollbackTesting.value) {
            // Trigger rollback using upgradeTo from the new implementation
            rollbackTesting.value = true;
            Address.functionDelegateCall(
                newImplementation,
                abi.encodeWithSignature(
                    "upgradeTo(address)",
                    oldImplementation
                )
            );
            rollbackTesting.value = false;
            // Check rollback was effective
            require(oldImplementation == _getImplementation(), "ERC1967Upgrade:
upgrade breaks further upgrades");
            // Finally reset to the new implementation and log the upgrade
            _setImplementation(newImplementation);
            emit Upgraded(newImplementation);
        }
    }
    /**
     * @dev Perform beacon upgrade with additional setup call. Note: This upgrades the address of the beacon, it does
     * not upgrade the implementation contained in the beacon (see {UpgradeableBeacon-_setImplementation} for that).
     *
     * Emits a {BeaconUpgraded} event.
     */
    function _upgradeBeaconToAndCall(address newBeacon, bytes memory data, bool forceCall) internal {
        _setBeacon(newBeacon);
        emit BeaconUpgraded(newBeacon);
        if (data.length > 0 || forceCall) {
            Address.functionDelegateCall(IBeacon(newBeacon).implementation(), data);
        }
    }

    /**
     * @dev Storage slot with the admin of the contract.
     * This is the keccak-256 hash of "eip1967.proxy.admin" subtracted by 1, and is
     * validated in the constructor.
     */
    bytes32 internal constant _ADMIN_SLOT = 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103;

    /**
     * @dev Emitted when the admin account has changed.
     */
    event AdminChanged(address previousAdmin, address newAdmin);

    /**
     * @dev Returns the current admin.
     */
    function _getAdmin() internal view returns (address) {
        return StorageSlot.getAddressSlot(_ADMIN_SLOT).value;
    }

    /**
     * @dev Stores a new address in the EIP1967 admin slot.
     */
    function _setAdmin(address newAdmin) private {
        require(newAdmin != address(0), "ERC1967: new admin is the zero address");
        StorageSlot.getAddressSlot(_ADMIN_SLOT).value = newAdmin;
    }

    /**
     * @dev Changes the admin of the proxy.
     *
     * Emits an {AdminChanged} event.
     */
    function _changeAdmin(address newAdmin) internal {
        emit AdminChanged(_getAdmin(), newAdmin);
        _setAdmin(newAdmin);
    }

    /**
     * @dev The storage slot of the UpgradeableBeacon contract which defines the implementation for this proxy.
     * This is bytes32(uint256(keccak256('eip1967.proxy.beacon')) - 1)) and is validated in the constructor.
     */
    bytes32 internal constant _BEACON_SLOT = 0xa3f0ad74e5423aebfd80d3ef4346578335a9a72aeaee59ff6cb3582b35133d50;

    /**
     * @dev Emitted when the beacon is upgraded.
     */
    event BeaconUpgraded(address indexed beacon);

    /**
     * @dev Returns the current beacon.
     */
    function _getBeacon() internal view returns (address) {
        return StorageSlot.getAddressSlot(_BEACON_SLOT).value;
    }

    /**
     * @dev Stores a new beacon in the EIP1967 beacon slot.
     */
    function _setBeacon(address newBeacon) private {
        require(
            Address.isContract(newBeacon),
            "ERC1967: new beacon is not a contract"
        );
        require(
            Address.isContract(IBeacon(newBeacon).implementation()),
            "ERC1967: beacon implementation is not a contract"
        );
        StorageSlot.getAddressSlot(_BEACON_SLOT).value = newBeacon;
    }
}

// File: @openzeppelin/contracts/proxy/ERC1967/ERC1967TheStake.sol



pragma solidity ^0.8.0;


/**
 * @dev This contract implements an upgradeable proxy. It is upgradeable because
calls are delegated to an
 * implementation address that can be changed. This address is stored in storage in
the location specified by
 * https://eips.ethereum.org/EIPS/eip-1967[EIP1967], so that it doesn't conflict with
the storage layout of the
 * implementation behind the proxy.
 */
contract ERC1967TheStake is TheStake, ERC1967Upgrade {
    /**
     * @dev Initializes the upgradeable proxy with an initial implementation
specified by `_logic`.
     *
     * If `_data` is nonempty, it's used as data in a delegate call to `_logic`. This
will typically be an encoded
     * function call, and allows initializating the storage of the proxy like a
Solidity constructor.
     */
    constructor(address _logic, bytes memory _data) payable {
        assert(_IMPLEMENTATION_SLOT ==
bytes32(uint256(keccak256("eip1967.proxy.implementation")) - 1));
        _upgradeToAndCall(_logic, _data, false);
    }

    /**
     * @dev Returns the current implementation address.
     */
    function _implementation() internal view virtual override returns (address impl)
{
        return ERC1967Upgrade._getImplementation();
    }
```

```
}

// File: @openzeppelin/contracts/proxy/transparent/TransparentUpgradeableTheStake.sol



pragma solidity ^0.8.0;

/**
 * @dev This contract implements a proxy that is upgradeable by an admin.
 *
 * To avoid https://medium.com/nomic-labs-blog/malicious-backdoors-in-ethereum-
proxies-62629adf3357[proxy selector
 * clashing], which can potentially be used in an attack, this contract uses the
 * https://blog.openzeppelin.com/the-transparent-proxy-pattern/[transparent proxy
pattern]. This pattern implies two
 * things that go hand in hand:
 *
 * 1. If any account other than the admin calls the proxy, the call will be forwarded
to the implementation, even if
 * that call matches one of the admin functions exposed by the proxy itself.
 * 2. If the admin calls the proxy, it can access the admin functions, but its calls
will never be forwarded to the
 * implementation. If the admin tries to call a function on the implementation it
will fail with an error that says
 * "admin cannot fallback to proxy target".
 *
 * These properties mean that the admin account can only be used for admin actions
like upgrading the proxy or changing
 * the admin, so it's best if it's a dedicated account that is not used for anything
else. This will avoid headaches due
 * to sudden errors when trying to call a function from the proxy implementation.
 *
 * Our recommendation is for the dedicated account to be an instance of the
{TheStakeAdmin} contract. If set up this way,
 * you should think of the `TheStakeAdmin` instance as the real administrative interface
of your proxy.
 */
contract TransparentUpgradeableTheStake is ERC1967TheStake {
    /**
     * @dev Initializes an upgradeable proxy managed by `_admin`, backed by the
implementation at `_logic`, and
     * optionally initialized with `_data` as explained in {ERC1967TheStake-
constructor}.
```

```
     */
    constructor(address _logic, address admin_, bytes memory _data) payable
ERC1967TheStake(_logic, _data) {
        assert(_ADMIN_SLOT == bytes32(uint256(keccak256("eip1967.proxy.admin")) -
1));
        _changeAdmin(admin_);
    }

    /**
     * @dev Modifier used internally that will delegate the call to the
implementation unless the sender is the admin.
     */
    modifier ifAdmin() {
        if (msg.sender == _getAdmin()) {
            _;
        } else {
            _fallback();
        }
    }

    /**
     * @dev Returns the current admin.
     *
     * NOTE: Only the admin can call this function. See {TheStakeAdmin-getTheStakeAdmin}.
     *
     * TIP: To get this value clients can read directly from the storage slot shown
below (specified by EIP1967) using the
     * https://eth.wiki/json-rpc/API#eth_getstorageat[`eth_getStorageAt`] RPC call.
     * `0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103`
     */
    function admin() external ifAdmin returns (address admin_) {
        admin_ = _getAdmin();
    }

    /**
     * @dev Returns the current implementation.
     *
     * NOTE: Only the admin can call this function. See {TheStakeAdmin-
getTheStakeImplementation}.
     *
     * TIP: To get this value clients can read directly from the storage slot shown
below (specified by EIP1967) using the
     * https://eth.wiki/json-rpc/API#eth_getstorageat[`eth_getStorageAt`] RPC call.
     * `0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc`
     */
    function implementation() external ifAdmin returns (address implementation_) {
        implementation_ = _implementation();
    }

    /**
     * @dev Changes the admin of the proxy.
     *
     * Emits an {AdminChanged} event.
     *
     * NOTE: Only the admin can call this function. See {TheStakeAdmin-
changeTheStakeAdmin}.
     */
    function changeAdmin(address newAdmin) external virtual ifAdmin {
        _changeAdmin(newAdmin);
    }

    /**
     * @dev Upgrade the implementation of the proxy.
     *
     * NOTE: Only the admin can call this function. See {TheStakeAdmin-upgrade}.
     */
    //Certik// The admin role can upgrade this contract to a new Implementation
address through the UpgradeTo function
    function upgradeTo(address newImplementation) external ifAdmin {
        _upgradeToAndCall(newImplementation, bytes(""), false);
    }

    /**
     * @dev Upgrade the implementation of the proxy, and then call a function from
the new implementation as specified
     * by `data`, which should be an encoded function call. This is useful to
initialize new storage variables in the
     * proxied contract.
     *
     * NOTE: Only the admin can call this function. See {TheStakeAdmin-upgradeAndCall}.
     */
    function upgradeToAndCall(address newImplementation, bytes calldata data)
external payable ifAdmin {
        _upgradeToAndCall(newImplementation, data, true);
    }

    /**
     * @dev Returns the current admin.
     */
    function _admin() internal view virtual returns (address) {
        return _getAdmin();
    }

    /**
     * @dev Makes sure the admin cannot access the fallback function. See {TheStake-
_beforeFallback}.
     */
    function _beforeFallback() internal virtual override {
        require(msg.sender != _getAdmin(), "TransparentUpgradeableTheStake: admin cannot
fallback to proxy target");
        super._beforeFallback();
    }
}
```

TheStake.sol

```
// File: @openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol

// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.6.0) (token/ERC20/IERC20.sol)
//Certik// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.0;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20Upgradeable {
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `to`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address to, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns
(uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `from` to `to` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);
}

// File: @openzeppelin/contracts-upgradeable/token/ERC20/extensions/IERC20MetadataUpgradeable.sol


// OpenZeppelin Contracts v4.4.1 (token/ERC20/extensions/IERC20Metadata.sol)

pragma solidity ^0.8.0;

/**
 * @dev Interface for the optional metadata functions from the ERC20 standard.
 *
 * _Available since v4.1._
 */
interface IERC20MetadataUpgradeable is IERC20Upgradeable {
    /**
     * @dev Returns the name of the token.
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the symbol of the token.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the decimals places of the token.
     */
    function decimals() external view returns (uint8);
}

// File: @openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol


// OpenZeppelin Contracts (last updated v4.7.0) (utils/Address.sol)

pragma solidity ^0.8.1;

/**
 * @dev Collection of functions related to the address type
 */
library AddressUpgradeable {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     *
     * [IMPORTANT]
     * ====
     * You shouldn't rely on `isContract` to protect against flash loan attacks!
     *
     * Preventing calls from contracts is highly discouraged. It breaks
composability, breaks support for smart wallets
     * like Gnosis Safe, and does not provide security since it can be circumvented
by calling from a contract
     * constructor.
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize/address.code.length, which returns 0
        // for contracts in construction, since the code is only stored at the end
        // of the constructor execution.

        return account.code.length > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-
now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-
the-checks-effects-interactions-pattern[checks-effects-interactions pattern].
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value, recipient may have
reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain `call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     * Returns the raw returned data. To convert to the expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?
highlight=abi.decode#abi-encoding-and-decoding-functions[`abi.decode`].
```

```
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes
memory) {
        return functionCall(target, data, "Address: low-level call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but
with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value
    ) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call
with value failed");
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}
[`functionCallWithValue`], but
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for
call");
        require(isContract(target), "Address: call to non-contract");

        (bool success, bytes memory returndata) = target.call{value: value}(data);
        return verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view
returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call
failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}
[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        (bool success, bytes memory returndata) = target.staticcall(data);
        return verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Tool to verifies that a low level call was successful, and revert if it
wasn't, either by bubbling the
     * revert reason using the provided one.
     *
     * _Available since v4.3._
     */
    function verifyCallResult(
        bool success,
        bytes memory returndata,
        string memory errorMessage
    ) internal pure returns (bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via
assembly
                /// @solidity memory-safe-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}

// File: @openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol
// OpenZeppelin Contracts (last updated v4.7.0) (proxy/utils/Initializable.sol)

pragma solidity ^0.8.2;

/**
 * @dev This is a base contract to aid in writing upgradeable contracts, or any kind
of contract that will be deployed
 * behind a proxy. Since proxied contracts do not make use of a constructor, it's
common to move constructor logic to an
 * external initializer function, usually called `initialize`. It then becomes
necessary to protect this initializer
 * function so it can only be called once. The {initializer} modifier provided by
this contract will have this effect.
 *
 * The initialization functions use a version number. Once a version number is used,
it is consumed and cannot be
 * reused. This mechanism prevents re-execution of each "step" but allows the
creation of new initialization steps in
 * case an upgrade adds a module that needs to be initialized.
 *
 * For example:
 *
 * [.hljs-theme-light.nopadding]
 * ```
 * contract MyToken is ERC20Upgradeable {
 *     function initialize() initializer public {
 *         __ERC20_init("MyToken", "MTK");
 *     }
 * }
 * contract MyTokenV2 is MyToken, ERC20PermitUpgradeable {
 *     function initializeV2() reinitializer(2) public {
 *         __ERC20Permit_init("MyToken");
 *     }
 * }
 * ```
 *
 * TIP: To avoid leaving the proxy in an uninitialized state, the initializer
function should be called as early as
 * possible by providing the encoded function call as the `_data` argument to
{ERC1967TheStake-constructor}.
 *
 * CAUTION: When used with inheritance, manual care must be taken to not invoke a
parent initializer twice, or to ensure
```

```
 * that all initializers are idempotent. This is not verified automatically as constructors are by Solidity.
 *
 * [CAUTION]
 * ====
 * Avoid leaving a contract uninitialized.
 *
 * An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation
 * contract, which may impact the proxy. To prevent the implementation contract from being used, you should invoke
 * the {_disableInitializers} function in the constructor to automatically lock it when it is deployed:
 *
 * [.hljs-theme-light.nopadding]
 * ```
 * /// @custom:oz-upgrades-unsafe-allow constructor
 * constructor() {
 *     _disableInitializers();
 * }
 * ```
 * ====
 */
abstract contract Initializable {
    /**
     * @dev Indicates that the contract has been initialized.
     * @custom:oz-retyped-from bool
     */
    uint8 private _initialized;

    /**
     * @dev Indicates that the contract is in the process of being initialized.
     */
    bool private _initializing;

    /**
     * @dev Triggered when the contract has been initialized or reinitialized.
     */
    event Initialized(uint8 version);

    /**
     * @dev A modifier that defines a protected initializer function that can be invoked at most once. In its scope,
     * `onlyInitializing` functions can be used to initialize parent contracts.
```

```
Equivalent to `reinitializer(1)`.
     */
    modifier initializer() {
        bool isTopLevelCall = !_initializing;
        require(
            (isTopLevelCall && _initialized < 1) ||
(!AddressUpgradeable.isContract(address(this)) && _initialized == 1),
            "Initializable: contract is already initialized"
        );
        _initialized = 1;
        if (isTopLevelCall) {
            _initializing = true;
        }
        _;
        if (isTopLevelCall) {
            _initializing = false;
            emit Initialized(1);
        }
    }

    /**
     * @dev A modifier that defines a protected reinitializer function that can be
invoked at most once, and only if the
     * contract hasn't been initialized to a greater version before. In its scope,
`onlyInitializing` functions can be
     * used to initialize parent contracts.
     *
     * `initializer` is equivalent to `reinitializer(1)`, so a reinitializer may be
used after the original
     * initialization step. This is essential to configure modules that are added
through upgrades and that require
     * initialization.
     *
     * Note that versions can jump in increments greater than 1; this implies that if
multiple reinitializers coexist in
     * a contract, executing them in the right order is up to the developer or
operator.
     */
    modifier reinitializer(uint8 version) {
        require(!_initializing && _initialized < version, "Initializable: contract is
already initialized");
        _initialized = version;
        _initializing = true;
        _;
        _initializing = false;
        emit Initialized(version);
    }

    /**
     * @dev Modifier to protect an initialization function so that it can only be
invoked by functions with the
     * {initializer} and {reinitializer} modifiers, directly or indirectly.
     */
    modifier onlyInitializing() {
        require(_initializing, "Initializable: contract is not initializing");
        _;
    }

    /**
     * @dev Locks the contract, preventing any future reinitialization. This cannot
be part of an initializer call.
     * Calling this in the constructor of a contract will prevent that contract from
being initialized or reinitialized
     * to any version. It is recommended to use this to lock implementation contracts
that are designed to be called
     * through proxies.
     */
    function _disableInitializers() internal virtual {
        require(!_initializing, "Initializable: contract is initializing");
        if (_initialized < type(uint8).max) {
            _initialized = type(uint8).max;
            emit Initialized(type(uint8).max);
        }
    }
}

// File: @openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol


// OpenZeppelin Contracts v4.4.1 (utils/Context.sol)

pragma solidity ^0.8.0;

/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract ContextUpgradeable is Initializable {
    function __Context_init() internal onlyInitializing {
    }

    function __Context_init_unchained() internal onlyInitializing {
    }
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }

    /**
     * @dev This empty reserved space is put in place to allow future versions to add
new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */
    uint256[50] private __gap;
}

// File: @openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol


// OpenZeppelin Contracts (last updated v4.7.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.0;




/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-
mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin Contracts guidelines: functions revert
 * instead returning `false` on failure. This behavior is nonetheless
 * conventional and does not conflict with the expectations of ERC20
 * applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20Upgradeable is Initializable, ContextUpgradeable, IERC20Upgradeable,
IERC20MetadataUpgradeable {
    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;

    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * The default value of {decimals} is 18. To select a different value for
     * {decimals} you should overload it.
     *
     * All two of these values are immutable: they can only be set once during
     * construction.
     */
    function __ERC20_init(string memory name_, string memory symbol_) internal
onlyInitializing {
        __ERC20_init_unchained(name_, symbol_);
    }
    function __ERC20_init_unchained(string memory name_, string memory symbol_)
internal onlyInitializing {
        _name = name_;
        _symbol = symbol_;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual override returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view virtual override returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5.05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the value {ERC20} uses, unless this function is
     * overridden;
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view virtual override returns (uint8) {
        return 18;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view virtual override returns
(uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `to` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address to, uint256 amount) public virtual override returns
(bool) {
        address owner = _msgSender();
        _transfer(owner, to, amount);
        //Certik// The return value conforms to the KIP7 specification
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual override
returns (uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, amount);
        //Certik// The return value conforms to the KIP7 specification
        return true;
    }

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20}.
     *
     * NOTE: Does not update the allowance if the current allowance
     * is the maximum `uint256`.
     *
     * Requirements:
     *
     * - `from` and `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     * - the caller must have allowance for ``from``'s tokens of at least
     * `amount`.
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        //Certik// The return value conforms to the KIP7 specification
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual
returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, allowance(owner, spender) + addedValue);
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public
virtual returns (bool) {
        address owner = _msgSender();
        uint256 currentAllowance = allowance(owner, spender);
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance
below zero");
        unchecked {
            _approve(owner, spender, currentAllowance - subtractedValue);
        }

        return true;
    }

    /**
     * @dev Moves `amount` of tokens from `from` to `to`.
     *
     * This internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     */
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
        require(from != address(0), "ERC20: transfer from the zero address");
        //Certik// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer
        require(to != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from];
        require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
        unchecked {
            _balances[from] = fromBalance - amount;
        }
        _balances[to] += amount;

        emit Transfer(from, to, amount);

        _afterTokenTransfer(from, to, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements:
     *
     * - `account` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");
        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);

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

    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements:
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

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

        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);

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

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
```

```
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        //Certik// This kind of check is very good, avoiding user mistake leading
to approve errors
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Updates `owner` s allowance for `spender` based on spent `amount`.
     *
     * Does not update the allowance amount in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Might emit an {Approval} event.
     */
    function _spendAllowance(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            require(currentAllowance >= amount, "ERC20: insufficient allowance");
            unchecked {
                _approve(owner, spender, currentAllowance - amount);
            }
        }
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-
hooks[Using Hooks].
     */
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

    /**
     * @dev Hook that is called after any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * has been transferred to `to`.
     * - when `from` is zero, `amount` tokens have been minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens have been burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-
hooks[Using Hooks].
     */
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

    /**
     * @dev This empty reserved space is put in place to allow future versions to add
new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */
    uint256[45] private __gap;
}

// File: @openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol


// OpenZeppelin Contracts (last updated v4.7.0) (security/Pausable.sol)

pragma solidity ^0.8.0;


/**
 * @dev Contract module which allows children to implement an emergency stop
 * mechanism that can be triggered by an authorized account.
 *
 * This module is used through inheritance. It will make available the
 * modifiers `whenNotPaused` and `whenPaused`, which can be applied to
 * the functions of your contract. Note that they will not be pausable by
 * simply including this module, only once the modifiers are put in place.
 */
abstract contract PausableUpgradeable is Initializable, ContextUpgradeable {
    /**
     * @dev Emitted when the pause is triggered by `account`.
     */
    event Paused(address account);

    /**
     * @dev Emitted when the pause is lifted by `account`.
     */
    event Unpaused(address account);

    bool private _paused;

    /**
     * @dev Initializes the contract in unpaused state.
     */
    function __Pausable_init() internal onlyInitializing {
        __Pausable_init_unchained();
    }

    function __Pausable_init_unchained() internal onlyInitializing {
        _paused = false;
    }

    /**
     * @dev Modifier to make a function callable only when the contract is not
paused.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    modifier whenNotPaused() {
        _requireNotPaused();
        _;
    }

    /**
     * @dev Modifier to make a function callable only when the contract is paused.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    modifier whenPaused() {
        _requirePaused();
        _;
    }

    /**
     * @dev Returns true if the contract is paused, and false otherwise.
     */
    function paused() public view virtual returns (bool) {
        return _paused;
    }

    /**
     * @dev Throws if the contract is paused.
     */
    function _requireNotPaused() internal view virtual {
        require(!paused(), "Pausable: paused");
    }

    /**
     * @dev Throws if the contract is not paused.
     */
    function _requirePaused() internal view virtual {
        require(paused(), "Pausable: not paused");
    }

    /**
     * @dev Triggers stopped state.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    function _pause() internal virtual whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }

    /**
     * @dev Returns to normal state.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    function _unpause() internal virtual whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }

    /**
     * @dev This empty reserved space is put in place to allow future versions to add
new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */
    uint256[49] private __gap;
}

// File: @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol


// OpenZeppelin Contracts (last updated v4.7.0) (access/Ownable.sol)

pragma solidity ^0.8.0;
```

```
/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract OwnableUpgradeable is Initializable, ContextUpgradeable {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    function __Ownable_init() internal onlyInitializing {
        __Ownable_init_unchained();
    }

    function __Ownable_init_unchained() internal onlyInitializing {
        _transferOwnership(_msgSender());
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        _checkOwner();
        _;
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view virtual returns (address) {
        return _owner;
    }
    /**
     * @dev Throws if the sender is not the owner.
     */
    function _checkOwner() internal view virtual {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        //Certik// This check is quite good in avoiding losing control of the
contract caused by user mistakes
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }

    /**
     * @dev This empty reserved space is put in place to allow future versions to add
new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */
    uint256[49] private __gap;
}

// File: contracts/TheStake.sol


pragma solidity ^0.8.4;




contract SuperWalk is Initializable, ERC20Upgradeable, PausableUpgradeable,
OwnableUpgradeable {
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize() initializer public {
        __ERC20_init("SuperWalk", "GRND");
        __Pausable_init();
        __Ownable_init();

        _mint(msg.sender, 1000000000 * 10 ** decimals());
    }
    //Certik// Suspending all transactions upon major abnormalities is a
recommended approach
    function pause() public onlyOwner {
        _pause();
    }

    function unpause() public onlyOwner {
        _unpause();
    }

    function _beforeTokenTransfer(address from, address to, uint256 amount)
        internal
        whenNotPaused
        override
    {
        super._beforeTokenTransfer(from, to, amount);
    }
}
```

Statement

Certik issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, Certik is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to Certik by the information provider till the date of the insurance report (referred to as "provided information"). Certik assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the Certik shall not be liable for any loss or adverse effect resulting therefrom. Certik only conducts the agreed security audit on the security situation of the project and issues this report. Certik is not responsible for the background and other conditions of the project.

Statement

Certik issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, Certik is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to Certik by the information provider till the date of the insurance report (referred to as "provided information"). Certik assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the Certik shall not be liable for any loss or adverse effect resulting therefrom. Certik only conducts the agreed security audit on the security situation of the project and issues this report. Certik is not responsible for the background and other conditions of the project.

CERTIK

Official Website
www.certik.com