Smart Contract Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the STEP.APP on 2022.04.18. The following are the details and results of this smart contract security audit:

Token Name :

STEP.APP

The contract address :

https://snowtrace.io/address/0x714f020C54cc9D104B6F4f6998C63ce2a31D1888

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
12	Scoping and Declarations Audit	Passed

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

Audit Result : Passed

Audit Number : 0X002204190001

Audit Date : 2022.04.18 - 2022.04.19

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that does not contain the tokenVault section. The contract contains the vote section. The total amount of contract tokens can be changed, users can burn their own tokens through the burn function. The contract does not have the Overflow and the Race Conditions issue.

The source code:

StepAppToken.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity 0.8.7;

import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20VotesComp.sol";

/**
 * @title StepAppToken - FITFI token.
 *
 * @dev step.app governance token.
 */
contract StepAppToken is ERC20Burnable, ERC20VotesComp {
    constructor() ERC20("STEP.APP", "FITFI") ERC20Permit("STEP.APP") {
        _mint(_msgSender(), 5e9 * 1e18);
    }

    function _mint(address account, uint256 amount)
```

```
        internal
        virtual
        override(ERC20, ERC20Votes)
    {
        ERC20Votes._mint(account, amount);
    }

    function _burn(address account, uint256 amount)
        internal
        virtual
        override(ERC20, ERC20Votes)
    {
        ERC20Votes._burn(account, amount);
    }

    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual override(ERC20, ERC20Votes) {
        ERC20Votes._afterTokenTransfer(from, to, amount);
    }
}
```

ERC20Burnable.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (token/ERC20/extensions/ERC20Burnable.sol)

pragma solidity ^0.8.0;

import "../ERC20.sol";
import "../../../utils/Context.sol";

/**
 * @dev Extension of {ERC20} that allows token holders to destroy both their own
 * tokens and those that they have an allowance for, in a way that can be
 * recognized off-chain (via event analysis).
 */
abstract contract ERC20Burnable is Context, ERC20 {
    /**
```

```
     * @dev Destroys `amount` tokens from the caller.
     *
     * See {ERC20-_burn}.
     */
    function burn(uint256 amount) public virtual {
        _burn(_msgSender(), amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`, deducting from the caller's
     * allowance.
     *
     * See {ERC20-_burn} and {ERC20-allowance}.
     *
     * Requirements:
     *
     * - the caller must have allowance for ``accounts``'s tokens of at least
     * `amount`.
     */
    function burnFrom(address account, uint256 amount) public virtual {
        _spendAllowance(account, _msgSender(), amount);
        //SlowMist// Because burnFrom() and transferFrom() share the allowed amount of
approve(), if the agent be evil, there is the possibility of malicious burn
        _burn(account, amount);
    }
}
```

ERC20VotesComp.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (token/ERC20/extensions/ERC20VotesComp.sol)

pragma solidity ^0.8.0;

import "./ERC20Votes.sol";

/**
 * @dev Extension of ERC20 to support Compound's voting and delegation. This version exactly
matches Compound's
 * interface, with the drawback of only supporting supply up to (2^96^ - 1).
 *
 * NOTE: You should use this contract if you need exact compatibility with COMP (for example
in order to use your token
 * with Governor Alpha or Bravo) and if you are sure the supply cap of 2^96^ is enough for
you. Otherwise, use the
 * {ERC20Votes} variant of this module.
 *
 * This extension keeps a history (checkpoints) of each account's vote power. Vote power can
be delegated either
 * by calling the {delegate} function directly, or by providing a signature to be used with
{delegateBySig}. Voting
 * power can be queried through the public accessors {getCurrentVotes} and {getPriorVotes}.
 *
 * By default, token balance does not account for voting power. This makes transfers cheaper.
The downside is that it
 * requires users to delegate to themselves in order to activate checkpoints and have their
voting power tracked.
 *
 * _Available since v4.2._
 */
abstract contract ERC20VotesComp is ERC20Votes {
    /**
     * @dev Comp version of the {getVotes} accessor, with `uint96` return type.
     */
    function getCurrentVotes(address account) external view virtual returns (uint96) {
        return SafeCast.toUint96(getVotes(account));
    }

    /**
     * @dev Comp version of the {getPastVotes} accessor, with `uint96` return type.
     */
    function getPriorVotes(address account, uint256 blockNumber) external view virtual returns
(uint96) {
        return SafeCast.toUint96(getPastVotes(account, blockNumber));
    }

    /**
     * @dev Maximum token supply. Reduced to `type(uint96).max` (2^96^ - 1) to fit COMP
interface.
     */
    function _maxSupply() internal view virtual override returns (uint224) {
        return type(uint96).max;
    }
}
```

ERC20.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.0;

import "./IERC20.sol";
import "./extensions/IERC20Metadata.sol";
import "../../utils/Context.sol";

/**
 * @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 ERC20 is Context, IERC20, IERC20Metadata {
    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.
 */
constructor(string memory name_, string memory symbol_) {
    _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);
        //SlowMist// The return value conforms to the EIP20 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);
        //SlowMist// The return value conforms to the EIP20 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);
        //SlowMist// The return value conforms to the EIP20 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, _allowances[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 = _allowances[owner][spender];
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
        unchecked {
            _approve(owner, spender, currentAllowance - subtractedValue);
        }

        return true;
    }
    /**
     * @dev Moves `amount` of tokens from `sender` to `recipient`.
     *
     * 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");
        //SlowMist// 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");
        //SlowMist// This kind of check is very good, avoiding user mistake leading to the
loss of token during transfer
        require(spender != address(0), "ERC20: approve to the zero address");

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

    /**
     * @dev Spend `amount` form the allowance of `owner` toward `spender`.
     *
     * 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 {}
}
```

Context.sol

```
// SPDX-License-Identifier: MIT
// 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 Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

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

IERC20.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (token/ERC20/IERC20.sol)

pragma solidity ^0.8.0;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {
    /**
     * @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);

    /**
     * @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);
}
```

IERC20Metadata.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (token/ERC20/extensions/IERC20Metadata.sol)

pragma solidity ^0.8.0;

import "../IERC20.sol";
```

```
/**
 * @dev Interface for the optional metadata functions from the ERC20 standard.
 *
 * _Available since v4.1._
 */
interface IERC20Metadata is IERC20 {
    /**
     * @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);
}
```

ERC20Votes.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (token/ERC20/extensions/ERC20Votes.sol)

pragma solidity ^0.8.0;

import "./draft-ERC20Permit.sol";
import "../../../utils/math/Math.sol";
import "../../../governance/utils/IVotes.sol";
import "../../../utils/math/SafeCast.sol";
import "../../../utils/cryptography/ECDSA.sol";

/**
 * @dev Extension of ERC20 to support Compound-like voting and delegation. This version is
more generic than Compound's,
 * and supports token supply up to 2^224^ - 1, while COMP is limited to 2^96^ - 1.
 *
 * NOTE: If exact COMP compatibility is required, use the {ERC20VotesComp} variant of this
```

```
module.
 *
 * This extension keeps a history (checkpoints) of each account's vote power. Vote power can
be delegated either
 * by calling the {delegate} function directly, or by providing a signature to be used with
{delegateBySig}. Voting
 * power can be queried through the public accessors {getVotes} and {getPastVotes}.
 *
 * By default, token balance does not account for voting power. This makes transfers cheaper.
The downside is that it
 * requires users to delegate to themselves in order to activate checkpoints and have their
voting power tracked.
 *
 * _Available since v4.2._
 */
abstract contract ERC20Votes is IVotes, ERC20Permit {
    struct Checkpoint {
        uint32 fromBlock;
        uint224 votes;
    }

    bytes32 private constant _DELEGATION_TYPEHASH =
        keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");

    mapping(address => address) private _delegates;
    mapping(address => Checkpoint[]) private _checkpoints;
    Checkpoint[] private _totalSupplyCheckpoints;

    /**
     * @dev Get the `pos`-th checkpoint for `account`.
     */
    function checkpoints(address account, uint32 pos) public view virtual returns (Checkpoint
memory) {
        return _checkpoints[account][pos];
    }

    /**
     * @dev Get number of checkpoints for `account`.
     */
    function numCheckpoints(address account) public view virtual returns (uint32) {
        return SafeCast.toUint32(_checkpoints[account].length);
    }

    /**
```

```
     * @dev Get the address `account` is currently delegating to.
     */
    function delegates(address account) public view virtual override returns (address) {
        return _delegates[account];
    }

    /**
     * @dev Gets the current votes balance for `account`
     */
    function getVotes(address account) public view virtual override returns (uint256) {
        uint256 pos = _checkpoints[account].length;
        return pos == 0 ? 0 : _checkpoints[account][pos - 1].votes;
    }

    /**
     * @dev Retrieve the number of votes for `account` at the end of `blockNumber`.
     *
     * Requirements:
     *
     * - `blockNumber` must have been already mined
     */
    function getPastVotes(address account, uint256 blockNumber) public view virtual override
returns (uint256) {
        require(blockNumber < block.number, "ERC20Votes: block not yet mined");
        return _checkpointsLookup(_checkpoints[account], blockNumber);
    }

    /**
     * @dev Retrieve the `totalSupply` at the end of `blockNumber`. Note, this value is the
sum of all balances.
     * It is but NOT the sum of all the delegated votes!
     *
     * Requirements:
     *
     * - `blockNumber` must have been already mined
     */
    function getPastTotalSupply(uint256 blockNumber) public view virtual override returns
(uint256) {
        require(blockNumber < block.number, "ERC20Votes: block not yet mined");
        return _checkpointsLookup(_totalSupplyCheckpoints, blockNumber);
    }

    /**
     * @dev Lookup a value in a list of (sorted) checkpoints.
     */
    function _checkpointsLookup(Checkpoint[] storage ckpts, uint256 blockNumber) private view
returns (uint256) {
        // We run a binary search to look for the earliest checkpoint taken after
`blockNumber`.
        //
        // During the loop, the index of the wanted checkpoint remains in the range [low-1,
high).
        // With each iteration, either `low` or `high` is moved towards the middle of the
range to maintain the invariant.
        // - If the middle checkpoint is after `blockNumber`, we look in [low, mid)
        // - If the middle checkpoint is before or equal to `blockNumber`, we look in [mid+1,
high)
        // Once we reach a single value (when low == high), we've found the right checkpoint
at the index high-1, if not
        // out of bounds (in which case we're looking too far in the past and the result is
0).
        // Note that if the latest checkpoint available is exactly for `blockNumber`, we end
up with an index that is
        // past the end of the array, so we technically don't find a checkpoint after
`blockNumber`, but it works out
        // the same.
        uint256 high = ckpts.length;
        uint256 low = 0;
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (ckpts[mid].fromBlock > blockNumber) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }

        return high == 0 ? 0 : ckpts[high - 1].votes;
    }

    /**
     * @dev Delegate votes from the sender to `delegatee`.
     */
    function delegate(address delegatee) public virtual override {
        _delegate(_msgSender(), delegatee);
    }

    /**
```

```
     * @dev Delegates votes from signer to `delegatee`
     */
    function delegateBySig(
        address delegatee,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= expiry, "ERC20Votes: signature expired");
        address signer = ECDSA.recover(
            _hashTypedDataV4(keccak256(abi.encode(_DELEGATION_TYPEHASH, delegatee, nonce,
expiry))),
            v,
            r,
            s
        );
        require(nonce == _useNonce(signer), "ERC20Votes: invalid nonce");
        _delegate(signer, delegatee);
    }

    /**
     * @dev Maximum token supply. Defaults to `type(uint224).max` (2^224^ - 1).
     */
    function _maxSupply() internal view virtual returns (uint224) {
        return type(uint224).max;
    }

    /**
     * @dev Snapshots the totalSupply after it has been increased.
     */
    function _mint(address account, uint256 amount) internal virtual override {
        super._mint(account, amount);
        require(totalSupply() <= _maxSupply(), "ERC20Votes: total supply risks overflowing
votes");

        _writeCheckpoint(_totalSupplyCheckpoints, _add, amount);
    }

    /**
     * @dev Snapshots the totalSupply after it has been decreased.
     */
    function _burn(address account, uint256 amount) internal virtual override {
        super._burn(account, amount);

        _writeCheckpoint(_totalSupplyCheckpoints, _subtract, amount);
    }

    /**
     * @dev Move voting power when tokens are transferred.
     *
     * Emits a {DelegateVotesChanged} event.
     */
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual override {
        super._afterTokenTransfer(from, to, amount);

        _moveVotingPower(delegates(from), delegates(to), amount);
    }

    /**
     * @dev Change delegation for `delegator` to `delegatee`.
     *
     * Emits events {DelegateChanged} and {DelegateVotesChanged}.
     */
    function _delegate(address delegator, address delegatee) internal virtual {
        address currentDelegate = delegates(delegator);
        uint256 delegatorBalance = balanceOf(delegator);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveVotingPower(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveVotingPower(
        address src,
        address dst,
        uint256 amount
    ) private {
        if (src != dst && amount > 0) {
            if (src != address(0)) {
                (uint256 oldWeight, uint256 newWeight) = _writeCheckpoint(_checkpoints[src],
_subtract, amount);
```

```
                emit DelegateVotesChanged(src, oldWeight, newWeight);
            }

            if (dst != address(0)) {
                (uint256 oldWeight, uint256 newWeight) = _writeCheckpoint(_checkpoints[dst],
_add, amount);
                emit DelegateVotesChanged(dst, oldWeight, newWeight);
            }
        }
    }

    function _writeCheckpoint(
        Checkpoint[] storage ckpts,
        function(uint256, uint256) view returns (uint256) op,
        uint256 delta
    ) private returns (uint256 oldWeight, uint256 newWeight) {
        uint256 pos = ckpts.length;
        oldWeight = pos == 0 ? 0 : ckpts[pos - 1].votes;
        newWeight = op(oldWeight, delta);

        if (pos > 0 && ckpts[pos - 1].fromBlock == block.number) {
            ckpts[pos - 1].votes = SafeCast.toUint224(newWeight);
        } else {
            ckpts.push(Checkpoint({fromBlock: SafeCast.toUint32(block.number), votes:
SafeCast.toUint224(newWeight)}));
        }
    }

    function _add(uint256 a, uint256 b) private pure returns (uint256) {
        return a + b;
    }

    function _subtract(uint256 a, uint256 b) private pure returns (uint256) {
        return a - b;
    }
}
```

draft-ERC20Permit.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (token/ERC20/extensions/draft-ERC20Permit.sol)
```

```
pragma solidity ^0.8.0;

import "./draft-IERC20Permit.sol";
import "../ERC20.sol";
import "../../../utils/cryptography/draft-EIP712.sol";
import "../../../utils/cryptography/ECDSA.sol";
import "../../../utils/Counters.sol";

/**
 * @dev Implementation of the ERC20 Permit extension allowing approvals to be made via
signatures, as defined in
 * https://eips.ethereum.org/EIPS/eip-2612[EIP-2612].
 *
 * Adds the {permit} method, which can be used to change an account's ERC20 allowance (see
{IERC20-allowance}) by
 * presenting a message signed by the account. By not relying on `{IERC20-approve}`, the token
holder account doesn't
 * need to send a transaction, and thus is not required to hold Ether at all.
 *
 * _Available since v3.4._
 */
abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712 {
    using Counters for Counters.Counter;

    mapping(address => Counters.Counter) private _nonces;

    // solhint-disable-next-line var-name-mixedcase
    bytes32 private immutable _PERMIT_TYPEHASH =
        keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256
deadline)");

    /**
     * @dev Initializes the {EIP712} domain separator using the `name` parameter, and setting
`version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined as the ERC20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}

    /**
     * @dev See {IERC20Permit-permit}.
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= deadline, "ERC20Permit: expired deadline");

        bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner, spender, value,
_useNonce(owner), deadline));

        bytes32 hash = _hashTypedDataV4(structHash);

        address signer = ECDSA.recover(hash, v, r, s);
        require(signer == owner, "ERC20Permit: invalid signature");

        _approve(owner, spender, value);
    }

    /**
     * @dev See {IERC20Permit-nonces}.
     */
    function nonces(address owner) public view virtual override returns (uint256) {
        return _nonces[owner].current();
    }

    /**
     * @dev See {IERC20Permit-DOMAIN_SEPARATOR}.
     */
    // solhint-disable-next-line func-name-mixedcase
    function DOMAIN_SEPARATOR() external view override returns (bytes32) {
        return _domainSeparatorV4();
    }

    /**
     * @dev "Consume a nonce": return the current value and increment.
     *
     * _Available since v4.1._
     */
    function _useNonce(address owner) internal virtual returns (uint256 current) {
        Counters.Counter storage nonce = _nonces[owner];
        current = nonce.current();
        nonce.increment();
    }
}
```

Math.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (utils/math/Math.sol)

pragma solidity ^0.8.0;

/**
 * @dev Standard math utilities missing in the Solidity language.
 */
library Math {
    /**
     * @dev Returns the largest of two numbers.
     */
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    /**
     * @dev Returns the smallest of two numbers.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    /**
     * @dev Returns the average of two numbers. The result is rounded towards
     * zero.
     */
    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b) / 2 can overflow.
        return (a & b) + (a ^ b) / 2;
    }

    /**
     * @dev Returns the ceiling of the division of two numbers.
     *
     * This differs from standard division with `/` in that it rounds up instead
     * of rounding down.
     */
    function ceilDiv(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b - 1) / b can overflow on addition, so we distribute.
        return a / b + (a % b == 0 ? 0 : 1);
    }
}
```

IVotes.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (governance/utils/IVotes.sol)
pragma solidity ^0.8.0;

/**
 * @dev Common interface for {ERC20Votes}, {ERC721Votes}, and other {Votes}-enabled contracts.
 *
 * _Available since v4.5._
 */
interface IVotes {
    /**
     * @dev Emitted when an account changes their delegate.
     */
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address
indexed toDelegate);

    /**
     * @dev Emitted when a token transfer or delegate change results in changes to a
delegate's number of votes.
     */
    event DelegateVotesChanged(address indexed delegate, uint256 previousBalance, uint256
newBalance);

    /**
     * @dev Returns the current amount of votes that `account` has.
     */
    function getVotes(address account) external view returns (uint256);

    /**
     * @dev Returns the amount of votes that `account` had at the end of a past block
(`blockNumber`).
     */
    function getPastVotes(address account, uint256 blockNumber) external view returns
(uint256);

    /**
     * @dev Returns the total supply of votes available at the end of a past block
(`blockNumber`).
     *
     * NOTE: This value is the sum of all available votes, which is not necessarily the sum of
all delegated votes.
     * Votes that have not been delegated are still part of total supply, even though they
would not participate in a
     * vote.
     */
    function getPastTotalSupply(uint256 blockNumber) external view returns (uint256);

    /**
     * @dev Returns the delegate that `account` has chosen.
     */
    function delegates(address account) external view returns (address);

    /**
     * @dev Delegates votes from the sender to `delegatee`.
     */
    function delegate(address delegatee) external;

    /**
     * @dev Delegates votes from signer to `delegatee`.
     */
    function delegateBySig(
        address delegatee,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external;
}
```

SafeCast.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/math/SafeCast.sol)

pragma solidity ^0.8.0;

/**
 * @dev Wrappers over Solidity's uintXX/intXX casting operators with added overflow
 * checks.
 *
 * Downcasting from uint256/int256 in Solidity does not revert on overflow. This can
 * easily result in undesired exploitation or bugs, since developers usually
 * assume that overflows raise errors. `SafeCast` restores this intuition by
 * reverting the transaction when such an operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 *
 * Can be combined with {SafeMath} and {SignedSafeMath} to extend it to smaller types, by
performing
 * all math on `uint256` and `int256` and then downcasting.
 */
library SafeCast {
    /**
     * @dev Returns the downcasted uint224 from uint256, reverting on
     * overflow (when the input is greater than largest uint224).
     *
     * Counterpart to Solidity's `uint224` operator.
     *
     * Requirements:
     *
     * - input must fit into 224 bits
     */
    function toUint224(uint256 value) internal pure returns (uint224) {
        require(value <= type(uint224).max, "SafeCast: value doesn't fit in 224 bits");
        return uint224(value);
    }

    /**
     * @dev Returns the downcasted uint128 from uint256, reverting on
     * overflow (when the input is greater than largest uint128).
     *
     * Counterpart to Solidity's `uint128` operator.
     *
     * Requirements:
     *
     * - input must fit into 128 bits
     */
    function toUint128(uint256 value) internal pure returns (uint128) {
        require(value <= type(uint128).max, "SafeCast: value doesn't fit in 128 bits");
        return uint128(value);
    }

    /**
     * @dev Returns the downcasted uint96 from uint256, reverting on
     * overflow (when the input is greater than largest uint96).
     *
     * Counterpart to Solidity's `uint96` operator.
     *
     * Requirements:
     *
     * - input must fit into 96 bits
     */
    function toUint96(uint256 value) internal pure returns (uint96) {
        require(value <= type(uint96).max, "SafeCast: value doesn't fit in 96 bits");
        return uint96(value);
    }

    /**
     * @dev Returns the downcasted uint64 from uint256, reverting on
     * overflow (when the input is greater than largest uint64).
     *
     * Counterpart to Solidity's `uint64` operator.
     *
     * Requirements:
     *
     * - input must fit into 64 bits
     */
    function toUint64(uint256 value) internal pure returns (uint64) {
        require(value <= type(uint64).max, "SafeCast: value doesn't fit in 64 bits");
        return uint64(value);
    }

    /**
     * @dev Returns the downcasted uint32 from uint256, reverting on
     * overflow (when the input is greater than largest uint32).
     *
     * Counterpart to Solidity's `uint32` operator.
     *
     * Requirements:
     *
     * - input must fit into 32 bits
     */
    function toUint32(uint256 value) internal pure returns (uint32) {
        require(value <= type(uint32).max, "SafeCast: value doesn't fit in 32 bits");
        return uint32(value);
    }

    /**
     * @dev Returns the downcasted uint16 from uint256, reverting on
     * overflow (when the input is greater than largest uint16).
     *
     * Counterpart to Solidity's `uint16` operator.
     *
     * Requirements:
     *
     * - input must fit into 16 bits
     */
    function toUint16(uint256 value) internal pure returns (uint16) {
        require(value <= type(uint16).max, "SafeCast: value doesn't fit in 16 bits");
        return uint16(value);
    }

    /**
     * @dev Returns the downcasted uint8 from uint256, reverting on
     * overflow (when the input is greater than largest uint8).
     *
     * Counterpart to Solidity's `uint8` operator.
     *
     * Requirements:
     *
     * - input must fit into 8 bits.
     */
    function toUint8(uint256 value) internal pure returns (uint8) {
        require(value <= type(uint8).max, "SafeCast: value doesn't fit in 8 bits");
        return uint8(value);
    }

    /**
     * @dev Converts a signed int256 into an unsigned uint256.
     *
     * Requirements:
```

```
     *
     * - input must be greater than or equal to 0.
     */
    function toUint256(int256 value) internal pure returns (uint256) {
        require(value >= 0, "SafeCast: value must be positive");
        return uint256(value);
    }

    /**
     * @dev Returns the downcasted int128 from int256, reverting on
     * overflow (when the input is less than smallest int128 or
     * greater than largest int128).
     *
     * Counterpart to Solidity's `int128` operator.
     *
     * Requirements:
     *
     * - input must fit into 128 bits
     *
     * _Available since v3.1._
     */
    function toInt128(int256 value) internal pure returns (int128) {
        require(value >= type(int128).min && value <= type(int128).max, "SafeCast: value
doesn't fit in 128 bits");
        return int128(value);
    }

    /**
     * @dev Returns the downcasted int64 from int256, reverting on
     * overflow (when the input is less than smallest int64 or
     * greater than largest int64).
     *
     * Counterpart to Solidity's `int64` operator.
     *
     * Requirements:
     *
     * - input must fit into 64 bits
     *
     * _Available since v3.1._
     */
    function toInt64(int256 value) internal pure returns (int64) {
        require(value >= type(int64).min && value <= type(int64).max, "SafeCast: value doesn't
fit in 64 bits");
        return int64(value);
    }

    /**
     * @dev Returns the downcasted int32 from int256, reverting on
     * overflow (when the input is less than smallest int32 or
     * greater than largest int32).
     *
     * Counterpart to Solidity's `int32` operator.
     *
     * Requirements:
     *
     * - input must fit into 32 bits
     *
     * _Available since v3.1._
     */
    function toInt32(int256 value) internal pure returns (int32) {
        require(value >= type(int32).min && value <= type(int32).max, "SafeCast: value doesn't
fit in 32 bits");
        return int32(value);
    }

    /**
     * @dev Returns the downcasted int16 from int256, reverting on
     * overflow (when the input is less than smallest int16 or
     * greater than largest int16).
     *
     * Counterpart to Solidity's `int16` operator.
     *
     * Requirements:
     *
     * - input must fit into 16 bits
     *
     * _Available since v3.1._
     */
    function toInt16(int256 value) internal pure returns (int16) {
        require(value >= type(int16).min && value <= type(int16).max, "SafeCast: value doesn't
fit in 16 bits");
        return int16(value);
    }

    /**
     * @dev Returns the downcasted int8 from int256, reverting on
     * overflow (when the input is less than smallest int8 or
     * greater than largest int8).
     *
     * Counterpart to Solidity's `int8` operator.
     *
     * Requirements:
     *
     * - input must fit into 8 bits.
     *
     * _Available since v3.1._
     */
    function toInt8(int256 value) internal pure returns (int8) {
        require(value >= type(int8).min && value <= type(int8).max, "SafeCast: value doesn't
fit in 8 bits");
        return int8(value);
    }

    /**
     * @dev Converts an unsigned uint256 into a signed int256.
     *
     * Requirements:
     *
     * - input must be less than or equal to maxInt256.
     */
    function toInt256(uint256 value) internal pure returns (int256) {
        // Note: Unsafe cast below is okay because `type(int256).max` is guaranteed to be
positive
        require(value <= uint256(type(int256).max), "SafeCast: value doesn't fit in an
int256");
        return int256(value);
    }
}
```

ECDSA.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (utils/cryptography/ECDSA.sol)

pragma solidity ^0.8.0;

import "../Strings.sol";

/**
```

```
 * @dev Elliptic Curve Digital Signature Algorithm (ECDSA) operations.
 *
 * These functions can be used to verify that a message was signed by the holder
 * of the private keys of a given address.
 */
library ECDSA {
    enum RecoverError {
        NoError,
        InvalidSignature,
        InvalidSignatureLength,
        InvalidSignatureS,
        InvalidSignatureV
    }

    function _throwError(RecoverError error) private pure {
        if (error == RecoverError.NoError) {
            return; // no error: do nothing
        } else if (error == RecoverError.InvalidSignature) {
            revert("ECDSA: invalid signature");
        } else if (error == RecoverError.InvalidSignatureLength) {
            revert("ECDSA: invalid signature length");
        } else if (error == RecoverError.InvalidSignatureS) {
            revert("ECDSA: invalid signature 's' value");
        } else if (error == RecoverError.InvalidSignatureV) {
            revert("ECDSA: invalid signature 'v' value");
        }
    }

    /**
     * @dev Returns the address that signed a hashed message (`hash`) with
     * `signature` or error string. This address can then be used for verification purposes.
     *
     * The `ecrecover` EVM opcode allows for malleable (non-unique) signatures:
     * this function rejects them by requiring the `s` value to be in the lower
     * half order, and the `v` value to be either 27 or 28.
     *
     * IMPORTANT: `hash` _must_ be the result of a hash operation for the
     * verification to be secure: it is possible to craft signatures that
     * recover to arbitrary addresses for non-hashed data. A safe way to ensure
     * this is by receiving a hash of the original message (which may otherwise
     * be too long), and then calling {toEthSignedMessageHash} on it.
     *
     * Documentation for signature generation:
     * - with https://web3js.readthedocs.io/en/v1.3.4/web3-eth-accounts.html#sign[Web3.js]
     * - with https://docs.ethers.io/v5/api/signer/#Signer-signMessage[ethers]
     *
     * _Available since v4.3._
     */
    function tryRecover(bytes32 hash, bytes memory signature) internal pure returns (address,
RecoverError) {
        // Check the signature length
        // - case 65: r,s,v signature (standard)
        // - case 64: r,vs signature (cf https://eips.ethereum.org/EIPS/eip-2098) _Available
since v4.1._
        if (signature.length == 65) {
            bytes32 r;
            bytes32 s;
            uint8 v;
            // ecrecover takes the signature parameters, and the only way to get them
            // currently is to use assembly.
            assembly {
                r := mload(add(signature, 0x20))
                s := mload(add(signature, 0x40))
                v := byte(0, mload(add(signature, 0x60)))
            }
            return tryRecover(hash, v, r, s);
        } else if (signature.length == 64) {
            bytes32 r;
            bytes32 vs;
            // ecrecover takes the signature parameters, and the only way to get them
            // currently is to use assembly.
            assembly {
                r := mload(add(signature, 0x20))
                vs := mload(add(signature, 0x40))
            }
            return tryRecover(hash, r, vs);
        } else {
            return (address(0), RecoverError.InvalidSignatureLength);
        }
    }

    /**
     * @dev Returns the address that signed a hashed message (`hash`) with
     * `signature`. This address can then be used for verification purposes.
     *
     * The `ecrecover` EVM opcode allows for malleable (non-unique) signatures:
     * this function rejects them by requiring the `s` value to be in the lower
     * half order, and the `v` value to be either 27 or 28.
     *
     * IMPORTANT: `hash` _must_ be the result of a hash operation for the
     * verification to be secure: it is possible to craft signatures that
     * recover to arbitrary addresses for non-hashed data. A safe way to ensure
     * this is by receiving a hash of the original message (which may otherwise
     * be too long), and then calling {toEthSignedMessageHash} on it.
     */
    function recover(bytes32 hash, bytes memory signature) internal pure returns (address) {
        (address recovered, RecoverError error) = tryRecover(hash, signature);
        _throwError(error);
        return recovered;
    }

    /**
     * @dev Overload of {ECDSA-tryRecover} that receives the `r` and `vs` short-signature
fields separately.
     *
     * See https://eips.ethereum.org/EIPS/eip-2098[EIP-2098 short signatures]
     *
     * _Available since v4.3._
     */
    function tryRecover(
        bytes32 hash,
        bytes32 r,
        bytes32 vs
    ) internal pure returns (address, RecoverError) {
        bytes32 s = vs &
bytes32(0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
        uint8 v = uint8((uint256(vs) >> 255) + 27);
        return tryRecover(hash, v, r, s);
    }

    /**
     * @dev Overload of {ECDSA-recover} that receives the `r and `vs` short-signature fields
separately.
     *
     * _Available since v4.2._
     */
    function recover(
        bytes32 hash,
        bytes32 r,
        bytes32 vs
    ) internal pure returns (address) {
        (address recovered, RecoverError error) = tryRecover(hash, r, vs);
        _throwError(error);
        return recovered;
    }

    /**
     * @dev Overload of {ECDSA-tryRecover} that receives the `v`,
     * `r` and `s` signature fields separately.
     *
     * _Available since v4.3._
     */
    function tryRecover(
        bytes32 hash,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal pure returns (address, RecoverError) {
        // EIP-2 still allows signature malleability for ecrecover(). Remove this possibility
and make the signature
        // unique. Appendix F in the Ethereum Yellow paper
(https://ethereum.github.io/yellowpaper/paper.pdf), defines
        // the valid range for s in (301): 0 < s < secp256k1n ÷ 2 + 1, and for v in (302): v
∈ {27, 28}. Most
        // signatures from current libraries generate a unique signature with an s-value in
the lower half order.
        //
        // If your library generates malleable signatures, such as s-values in the upper
range, calculate a new s-value
        // with 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and
flip v from 27 to 28 or
        // vice versa. If your library also generates signatures with 0/1 for v instead 27/28,
add 27 to v to accept
        // these malleable signatures as well.
        if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
            return (address(0), RecoverError.InvalidSignatureS);
        }
        if (v != 27 && v != 28) {
            return (address(0), RecoverError.InvalidSignatureV);
        }

        // If the signature is valid (and not malleable), return the signer address
        address signer = ecrecover(hash, v, r, s);
        if (signer == address(0)) {
            return (address(0), RecoverError.InvalidSignature);
        }
        return (signer, RecoverError.NoError);
    }

    /**
     * @dev Overload of {ECDSA-recover} that receives the `v`,
     * `r` and `s` signature fields separately.
     */
    function recover(
        bytes32 hash,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal pure returns (address) {
        (address recovered, RecoverError error) = tryRecover(hash, v, r, s);
        _throwError(error);
        return recovered;
    }

    /**
     * @dev Returns an Ethereum Signed Message, created from a `hash`. This
     * produces hash corresponding to the one signed with the
     * https://eth.wiki/json-rpc/API#eth_sign[`eth_sign`]
     * JSON-RPC method as part of EIP-191.
     *
     * See {recover}.
     */
    function toEthSignedMessageHash(bytes32 hash) internal pure returns (bytes32) {
        // 32 is the length in bytes of hash,
        // enforced by the type signature above
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }

    /**
     * @dev Returns an Ethereum Signed Message, created from `s`. This
     * produces hash corresponding to the one signed with the
     * https://eth.wiki/json-rpc/API#eth_sign[`eth_sign`]
     * JSON-RPC method as part of EIP-191.
     *
     * See {recover}.
     */
    function toEthSignedMessageHash(bytes memory s) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n",
Strings.toString(s.length), s));
```

```
    }

    /**
     * @dev Returns an Ethereum Signed Typed Data, created from a
     * `domainSeparator` and a `structHash`. This produces hash corresponding
     * to the one signed with the
     * https://eips.ethereum.org/EIPS/eip-712[`eth_signTypedData`]
     * JSON-RPC method as part of EIP-712.
     *
     * See {recover}.
     */
    function toTypedDataHash(bytes32 domainSeparator, bytes32 structHash) internal pure
returns (bytes32) {
        return keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
    }
}
```

draft-IERC20Permit.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (token/ERC20/extensions/draft-IERC20Permit.sol)

pragma solidity ^0.8.0;

/**
 * @dev Interface of the ERC20 Permit extension allowing approvals to be made via signatures,
as defined in
 * https://eips.ethereum.org/EIPS/eip-2612[EIP-2612].
 *
 * Adds the {permit} method, which can be used to change an account's ERC20 allowance (see
{IERC20-allowance}) by
 * presenting a message signed by the account. By not relying on {IERC20-approve}, the token
holder account doesn't
 * need to send a transaction, and thus is not required to hold Ether at all.
 */
interface IERC20Permit {
    /**
     * @dev Sets `value` as the allowance of `spender` over ``owner``'s tokens,
     * given ``owner``'s signed approval.
     *
     * IMPORTANT: The same issues {IERC20-approve} has related to transaction
     * ordering also apply here.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `deadline` must be a timestamp in the future.
     * - `v`, `r` and `s` must be a valid `secp256k1` signature from `owner`
     * over the EIP712-formatted function arguments.
     * - the signature must use ``owner``'s current nonce (see {nonces}).
     *
     * For more information on the signature format, see the
     * https://eips.ethereum.org/EIPS/eip-2612#specification[relevant EIP
     * section].
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external;

    /**
     * @dev Returns the current nonce for `owner`. This value must be
     * included whenever a signature is generated for {permit}.
     *
     * Every successful call to {permit} increases ``owner``'s nonce by one. This
     * prevents a signature from being used multiple times.
     */
    function nonces(address owner) external view returns (uint256);

    /**
     * @dev Returns the domain separator used in the encoding of the signature for {permit},
as defined by {EIP712}.
     */
    // solhint-disable-next-line func-name-mixedcase
    function DOMAIN_SEPARATOR() external view returns (bytes32);
}
```

draft-EIP712.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/cryptography/draft-EIP712.sol)

pragma solidity ^0.8.0;

import "./ECDSA.sol";

/**
 * @dev https://eips.ethereum.org/EIPS/eip-712[EIP 712] is a standard for hashing and signing
of typed structured data.
 *
 * The encoding specified in the EIP is very generic, and such a generic implementation in
Solidity is not feasible,
 * thus this contract does not implement the encoding itself. Protocols need to implement the
type-specific encoding
 * they need in their contracts using a combination of `abi.encode` and `keccak256`.
 *
 * This contract implements the EIP 712 domain separator ({_domainSeparatorV4}) that is used
as part of the encoding
 * scheme, and the final step of the encoding to obtain the message digest that is then signed
via ECDSA
 * ({_hashTypedDataV4}).
 *
 * The implementation of the domain separator was designed to be as efficient as possible
while still properly updating
 * the chain id to protect against replay attacks on an eventual fork of the chain.
 *
 * NOTE: This contract implements the version of the encoding known as "v4", as implemented by
the JSON RPC method
 * https://docs.metamask.io/guide/signing-data.html[`eth_signTypedDataV4` in MetaMask].
 *
 * _Available since v3.4._
 */
abstract contract EIP712 {
    /* solhint-disable var-name-mixedcase */
    // Cache the domain separator as an immutable value, but also store the chain id that it
corresponds to, in order to
    // invalidate the cached domain separator if the chain id changes.
    bytes32 private immutable _CACHED_DOMAIN_SEPARATOR;
    uint256 private immutable _CACHED_CHAIN_ID;
    address private immutable _CACHED_THIS;
    bytes32 private immutable _HASHED_NAME;
    bytes32 private immutable _HASHED_VERSION;
    bytes32 private immutable _TYPE_HASH;

    /* solhint-enable var-name-mixedcase */

    /**
     * @dev Initializes the domain separator and parameter caches.
     *
     * The meaning of `name` and `version` is specified in
     * https://eips.ethereum.org/EIPS/eip-712#definition-of-domainseparator[EIP 712]:
     *
     * - `name`: the user readable name of the signing domain, i.e. the name of the DApp or
the protocol.
     * - `version`: the current major version of the signing domain.
     *
     * NOTE: These parameters cannot be changed except through a xref:learn::upgrading-smart-
contracts.adoc[smart
     * contract upgrade].
     */
    constructor(string memory name, string memory version) {
        bytes32 hashedName = keccak256(bytes(name));
        bytes32 hashedVersion = keccak256(bytes(version));
        bytes32 typeHash = keccak256(
            "EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"
        );
        _HASHED_NAME = hashedName;
        _HASHED_VERSION = hashedVersion;
        _CACHED_CHAIN_ID = block.chainid;
        _CACHED_DOMAIN_SEPARATOR = _buildDomainSeparator(typeHash, hashedName, hashedVersion);
        _CACHED_THIS = address(this);
        _TYPE_HASH = typeHash;
    }

    /**
     * @dev Returns the domain separator for the current chain.
     */
    function _domainSeparatorV4() internal view returns (bytes32) {
        if (address(this) == _CACHED_THIS && block.chainid == _CACHED_CHAIN_ID) {
            return _CACHED_DOMAIN_SEPARATOR;
        } else {
            return _buildDomainSeparator(_TYPE_HASH, _HASHED_NAME, _HASHED_VERSION);
        }
    }

    function _buildDomainSeparator(
        bytes32 typeHash,
        bytes32 nameHash,
        bytes32 versionHash
    ) private view returns (bytes32) {
        return keccak256(abi.encode(typeHash, nameHash, versionHash, block.chainid,
address(this)));
    }

    /**
     * @dev Given an already https://eips.ethereum.org/EIPS/eip-712#definition-of-
hashstruct[hashed struct], this
     * function returns the hash of the fully encoded EIP712 message for this domain.
     *
     * This hash can be used together with {ECDSA-recover} to obtain the signer of a message.
For example:
     *
     * ```solidity
     * bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
     *     keccak256("Mail(address to,string contents)"),
     *     mailTo,
     *     keccak256(bytes(mailContents))
     * )));
     * address signer = ECDSA.recover(digest, signature);
     * ```
     */
    function _hashTypedDataV4(bytes32 structHash) internal view virtual returns (bytes32) {
        return ECDSA.toTypedDataHash(_domainSeparatorV4(), structHash);
    }
}
```

Counters.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/Counters.sol)

pragma solidity ^0.8.0;
```

```
/**
 * @title Counters
 * @author Matt Condon (@shrugs)
 * @dev Provides counters that can only be incremented, decremented or reset. This can be used
e.g. to track the number
 * of elements in a mapping, issuing ERC721 ids, or counting request ids.
 *
 * Include with `using Counters for Counters.Counter;`
 */
library Counters {
    struct Counter {
        // This variable should never be directly accessed by users of the library:
interactions must be restricted to
        // the library's function. As of Solidity v0.5.2, this cannot be enforced, though
there is a proposal to add
        // this feature: see https://github.com/ethereum/solidity/issues/4637
        uint256 _value; // default: 0
    }

    function current(Counter storage counter) internal view returns (uint256) {
        return counter._value;
    }

    function increment(Counter storage counter) internal {
        unchecked {
            counter._value += 1;
        }
    }

    function decrement(Counter storage counter) internal {
        uint256 value = counter._value;
        require(value > 0, "Counter: decrement overflow");
        unchecked {
            counter._value = value - 1;
        }
    }

    function reset(Counter storage counter) internal {
        counter._value = 0;
    }
}
```

Strings.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/Strings.sol)

pragma solidity ^0.8.0;

/**
 * @dev String operations.
 */
library Strings {
    bytes16 private constant _HEX_SYMBOLS = "0123456789abcdef";

    /**
     * @dev Converts a `uint256` to its ASCII `string` decimal representation.
     */
    function toString(uint256 value) internal pure returns (string memory) {
        // Inspired by OraclizeAPI's implementation - MIT licence
        // https://github.com/oraclize/ethereum-
api/blob/b42146b063c7d6ee1358846c198246239e9360e8/oraclizeAPI_0.4.25.sol

        if (value == 0) {
            return "0";
        }
        uint256 temp = value;
        uint256 digits;
        while (temp != 0) {
            digits++;
            temp /= 10;
        }
        bytes memory buffer = new bytes(digits);
        while (value != 0) {
            digits -= 1;
            buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
            value /= 10;
        }
        return string(buffer);
    }

    /**
     * @dev Converts a `uint256` to its ASCII `string` hexadecimal representation.
     */
    function toHexString(uint256 value) internal pure returns (string memory) {
        if (value == 0) {
            return "0x00";
        }
        uint256 temp = value;
        uint256 length = 0;
        while (temp != 0) {
            length++;
            temp >>= 8;
        }
        return toHexString(value, length);
    }

    /**
     * @dev Converts a `uint256` to its ASCII `string` hexadecimal representation with fixed
length.
     */
    function toHexString(uint256 value, uint256 length) internal pure returns (string memory)
{
        bytes memory buffer = new bytes(2 * length + 2);
        buffer[0] = "0";
        buffer[1] = "x";
        for (uint256 i = 2 * length + 1; i > 1; --i) {
            buffer[i] = _HEX_SYMBOLS[value & 0xf];
            value >>= 4;
        }
        require(value == 0, "Strings: hex length insufficient");
        return string(buffer);
    }
}
```

Statement

SlowMist 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, SlowMist 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 SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist 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 SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.