KNOWNSEC

Degree of secrets ★ | Project team open

# Smart contract audit report

## Gate USD (USDG)

## FiatTokenProxy

Security status

**Security**

★ ★ ★ ★ ★

Chief test Officer: Knownsec blockchain security team

# Version Summary

| Content | Date | Team | Version |
| --- | --- | --- | --- |
| Editing Document | 20200930 | Knownsec blockchain security team | V1.0 |

# Report Information

| Title | Version | Document Number | Type |
| --- | --- | --- | --- |
| Gate USD (USDG) FiatTokenProxy contract audit report | V1.0 | 【FIATTOKENPROXY-ZNNY-20200930】 | Open to project team |

# Copyright Notice

# Table of Contents

# 1. Introduction

The effective test time of this report is from September 29, 2020 to September 30, 2020. During this period, the security and standardization of **the smart contract code of the Gate USD (USDG) FiatTokenProxy** will be audited and used as the statistical basis for the report.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3), No related security issues found, **the smart contract code of the Gate USD (USDG) FiatTokenProxy** is comprehensively assessed as **SAFE**.

**Results of this smart contract security audit：SAFE**

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

**Target information of the Gate USD (USDG) FiatTokenProxy audit:**

| Target information | |
|---|---|
| **Token name** | Gate USD (USDG) FiatTokenProxy |
| **Code type** | Token code、defi protocol code、ETH smart contract code |
| **Contract address** | 0x1c0b9c1995ea61f8fb505e4d94aa32b0c024899e |
| **Code language** | solidity |

## Contract documents and hash:

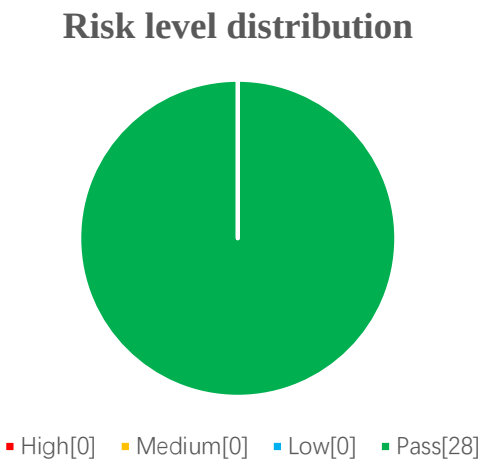
| Contract documents | MD5 |
| --- | --- |
| **FiatTokenProxy.sol** | 76EB142D5014A15FADB39B074F042A7A |
| **Address.sol** | 1A1BA1CF1A033D8ED33674E8447ADCC6 |
| **Proxy.sol** | EB1B511F8E3F4D42E1EC5DAD26379CE7 |
| **UpgradeabilityProxy.sol** | 2EB84F9E3DFDCA2F6CF9678848002213 |
| **AdminUpgradeabilityProxy.sol** | EEA51BFA90778FBD73EA572C7E714904 |

# 2. Code vulnerability analysis

## 2.1. Vulnerability Level Distribution

Vulnerability risk statistics by level:

| Vulnerability risk level statistics table | | | |
|---|---|---|---|
| High | Medium | Low | Pass |
| 0 | 0 | 0 | 29 |



## 2.2. Audit Result

| Result of audit | | | |
|---|---|---|---|
| **Audit Target** | **Audit** | **Status** | **Audit Description** |
| **Smart Contract** | Reentry attack detection | Pass | After testing, there is no such safety vulnerability. |
| | Replay attack detection | Pass | After testing, there is no such safety vulnerability |
| | Rearrangement attack detection | Pass | After testing, there is no such safety vulnerability |
| | Numerical overflow detection | Pass | After testing, there is no such safety vulnerability. |
| | Arithmetic accuracy error | Pass | After testing, there is no such safety vulnerability. |
| | Access control defect detection | Pass | After testing, there is no such safety vulnerability. |
| | tx.progin authentication | Pass | After testing, there is no such safety vulnerability. |
| | call injection attack | Pass | After testing, there is no such safety vulnerability. |
| | Unverified return value call | Pass | After testing, there is no such safety vulnerability. |
| | Uninitialized storage pointer | Pass | After testing, there is no such safety vulnerability. |
| | Wrong use of random number detection | Pass | After testing, there is no such safety vulnerability. |
| | Transaction order dependency detection | Pass | After testing, there is no such safety vulnerability. |
| | Denial of service attack detection | Pass | After testing, there is no such safety vulnerability. |

| | | | |
|---|---|---|---|
| | Logical design defect detection | Pass | After testing, there is no such safety vulnerability. |
| | Fake recharge vulnerability detection | Pass | After testing, there is no such safety vulnerability. |
| | Additional token issuance vulnerability detection | Pass | After testing, there is no such safety vulnerability. |
| | Frozen account bypass detection | Pass | After testing, there is no such safety vulnerability. |
| | Compiler version security | Pass | After testing, there is no such safety vulnerability. |
| | Not recommended encoding | Pass | After testing, there is no such safety vulnerability. |
| | Redundant code | Pass | After testing, there is no such safety vulnerability. |
| | Use of safe arithmetic library | Pass | After testing, there is no such safety vulnerability. |
| | Use of require/assert | Pass | After testing, there is no such safety vulnerability. |
| | gas consumption | Pass | After testing, there is no such safety vulnerability. |
| | Use of fallback function | Pass | After testing, there is no such safety vulnerability. |
| | Owner permission control | Pass | After testing, there is no such safety vulnerability. |
| | Low-level function safety | Pass | After testing, there is no such safety vulnerability. |
| | Variable coverage | Pass | After testing, there is no such safety vulnerability. |
| | Timestamp dependent attack | Pass | After testing, there is no such safety vulnerability. |
| | Use of unsafe API | Pass | After testing, there is no such safety vulnerability. |

# 3. Analysis of code audit results

## 3.1. Reentry attack detection【PASS】

Re-entry vulnerability is the most famous Ethereum smart contract vulnerability, which caused the fork of Ethereum(The DAO hack).

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send Ether. When the **call.value()** function to send Ether occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.2. Replay attack detection【PASS】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

**Audit results：** After testing, the smart contract does not use the call function, and this vulnerability does not exist.

**Recommendation：** nothing.

## 3.3. Rearrangement attack detection【PASS】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping, so that the attacker has the opportunity to store their own information in the contract. in.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.4. Numerical overflow detection【PASS】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.5. Arithmetic accuracy error【PASS】

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.6. Access control defect detection【PASS】

Different functions in the contract should be set with reasonable permissions, check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid unauthorized access problem.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.7. tx.progin authentication【PASS】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.8. call inject attack【PASS】

When the call function is called, strict permission control should be done, or the function called by the call should be written dead.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.9. Unverified return value call【PASS】

This vulnerability mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are **transfer()**, **send()**, **call.value()** and other currency transfer

methods, which can all be used to send Ether to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when **call.value** fails to be sent; all available gas will be passed for calling (can be Limit by passing in the gas_value parameter), which cannot effectively prevent reentry attacks.

If the return value of the above send and **call.value** transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to Ether sending failure.

**Audit results：** After auditing, this security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.10. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and local variables in a function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

**Audit results：** After auditing, this security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.11. Wrong use of random number detection【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access obviously unpredictable values, such as **block.number** and **block.timestamp**, they are usually either more public than they appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

**Audit results：** After auditing, the relevant random number function is not used in the smart contract code.

**Recommendation：** nothing.

## 3.12. Transaction order dependency detection【PASS】

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the others the content of the pending transaction. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

**Audit results：** After auditing, the security vulnerability does not exist in the smart

contract code.

**Recommendation：** nothing.

## 3.13. Denial of service attack detection【PASS】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of a smart contract, including malicious behavior as the recipient of a transaction , Artificially increasing the gas required for computing functions leads to gas exhaustion, abuse of access control to access private components of smart contracts, use of confusion and negligence, etc.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.14. Logical design defect detection【PASS】

Detect security issues related to business design in smart contract code.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code. For details, see Appendix A: Contract Code.

**Recommendation：** nothing.

## 3.15. Fake recharge vulnerability detection【PASS】

The transfer function of the token contract uses the if judgment method to check

the balance of the transfer initiator **(msg.sender)**. When **balances[msg.sender] <value**, enter the else logic part and return false, and finally no exception is thrown. We believe that only using if/else as a judgment method is not rigorous in sensitive function scenarios such as transfer.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.16. Additional token issuance vulnerability detection 【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** This issue is not a security issue, but some exchanges will restrict the use of additional issuance functions, and the specific situation depends on the requirements of the exchange.

## 3.17. Frozen account bypass detection【PASS】

Detect whether there is an operation that has not verified whether the source account of the token, the originating account, and the target account are frozen when the token is transferred in the token contract.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.18. Compiler version security【PASS】

Check whether a safe compiler version is used in the contract code implementation.

**Audit results：** After testing, the compiler version 0.6.0 or above is specified in the smart contract code, and there is no such security issue.

**Recommendation：** nothing.

## 3.19. Not recommended encoding【PASS】

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.20. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.21. Use of safe arithmetic library【PASS】

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

**Audit results：** After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

**Recommendation：** nothing.

## 3.22. Use of require/assert【PASS】

Check the rationality of the use of require and assert statements in the contract code implementation.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.23. gas consumption【PASS】

Check whether the consumption of gas exceeds the maximum block limit.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.24. Use of fallback function【PASS】

Check whether the fallback function is used correctly in the contract code

implementation.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.25. Owner permission control【PASS】

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.26. Low-level function safety【PASS】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.27. Variable coverage【PASS】

Check whether there are security issues caused by variable coverage in the contract code implementation.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.28. Timestamp dependent attack【PASS】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block and the error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 3.29. Use of unsafe API【PASS】

Check whether unsafe interfaces are used in the contract code implementation.

**Audit results：** After auditing, the security vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

# 4. Appendix A：Contract code

**Source code**:

```
/**
  *Submitted for verification at Etherscan.io on 2020-09-27
*/

// File: contracts/upgradeability/Proxy.sol


pragma solidity 0.6.12;

/**
  * @notice Implements delegation of calls to other contracts, with proper
  * forwarding of return values and bubbling of failures.
  * It defines a fallback function that delegates all calls to the address
  * returned by the abstract _implementation() internal function.
  *                @dev                Forked                from                https://github.com/zeppelinos/zos-lib/blob/8a16ef3ad17ec7430e3a9d2b5e3f39b8204f8c8d/contracts/upgradeability/Proxy.sol
  * Modifications:
  * 1. Reformat and conform to Solidity 0.6 syntax (5/13/20)
  */
abstract contract Proxy {
    /**
      * @dev Fallback function.
      * Implemented entirely in `_fallback`.
      */
    fallback() external payable {
        _fallback();
    }

    /**
      * @return The Address of the implementation.
      */
    function _implementation() internal virtual view returns (address);

    /**
      * @dev Delegates execution to an implementation contract.
      * This is a low level function that doesn't return to its internal call site.
      * It will return to the external caller whatever the implementation returns.
      * @param implementation Address to delegate.
      */
    function _delegate(address implementation) internal {
        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 Function that is run as the first thing in the fallback function.
      * Can be redefined in derived contracts to add functionality.
      * Redefinitions must call super._willFallback().
      */
    function _willFallback() internal virtual {}
```

```
    /**
     * @dev fallback implementation.
     * Extracted to enable manual triggering.
     */
    function _fallback() internal {
        _willFallback();
        _delegate(_implementation());
    }
}

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

pragma solidity ^0.6.2;

/**
 * @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) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
        // solhint-disable-next-line no-inline-assembly
        assembly { codehash := extcodehash(account) }
        return (codehash != accountHash && codehash != 0x0);
    }

    /**
     * @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");
        return _functionCallWithValue(target, data, value, errorMessage);
    }

    function _functionCallWithValue(address target, bytes memory data, uint256 weiValue, string memory errorMessage) private returns (bytes memory) {
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(data);
        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: contracts/upgradeability/UpgradeabilityProxy.sol


pragma solidity 0.6.12;



/**
 * @notice This contract implements a proxy that allows to change the
 * implementation address to which it will delegate.
 * Such a change is called an implementation upgrade.
 *          @dev          Forked          from          https://github.com/zeppelinos/zos-lib/blob/8a16ef3ad17ec7430e3a9d2b5e3f39b8204f8c8d/contracts/upgradeability/UpgradeabilityProxy.sol
```

```
 * Modifications:
 * 1. Reformat, conform to Solidity 0.6 syntax, and add error messages (5/13/20)
 * 2. Use Address utility library from the latest OpenZeppelin (5/13/20)
 */
contract UpgradeabilityProxy is Proxy {
    /**
     * @dev Emitted when the implementation is upgraded.
     * @param implementation Address of the new implementation.
     */
    event Upgraded(address implementation);

    /**
     * @dev Storage slot with the address of the current implementation.
     * This is the keccak-256 hash of "org.zeppelinos.proxy.implementation", and is
     * validated in the constructor.
     */
    bytes32
        private constant IMPLEMENTATION_SLOT =
0x7050c9e0f4ca769c69bd3a8ef740bc37934f8e2c036e5a723fd8ee048ed3f8c3;

    /**
     * @dev Contract constructor.
     * @param implementationContract Address of the initial implementation.
     */
    constructor(address implementationContract) public {
        assert(
            IMPLEMENTATION_SLOT ==
                keccak256("org.zeppelinos.proxy.implementation")
        );

        _setImplementation(implementationContract);
    }

    /**
     * @dev Returns the current implementation.
     * @return impl Address of the current implementation
     */
    function _implementation() internal override view returns (address impl) {
        bytes32 slot = IMPLEMENTATION_SLOT;
        assembly {
            impl := sload(slot)
        }
    }

    /**
     * @dev Upgrades the proxy to a new implementation.
     * @param newImplementation Address of the new implementation.
     */
    function _upgradeTo(address newImplementation) internal {
        _setImplementation(newImplementation);
        emit Upgraded(newImplementation);
    }

    /**
     * @dev Sets the implementation address of the proxy.
     * @param newImplementation Address of the new implementation.
     */
    function _setImplementation(address newImplementation) private {
        require(
            Address.isContract(newImplementation),
            "Cannot set a proxy implementation to a non-contract address"
        );

        bytes32 slot = IMPLEMENTATION_SLOT;

        assembly {
            sstore(slot, newImplementation)
        }
    }
}

// File: contracts/upgradeability/AdminUpgradeabilityProxy.sol

pragma solidity 0.6.12;


/**
 * @notice This contract combines an upgradeability proxy with an authorization
 * mechanism for administrative tasks.
 * @dev Forked from https://github.com/zeppelinos/zos-
lib/blob/8a16ef3ad17ec7430e3a9d2b5e3f39b8204f8c8d/contracts/upgradeability/AdminUpgradeabilityProxy.sol
 * Modifications:
 * 1. Reformat, conform to Solidity 0.6 syntax, and add error messages (5/13/20)
 * 2. Remove ifAdmin modifier from admin() and implementation() (5/13/20)
 */
contract AdminUpgradeabilityProxy is UpgradeabilityProxy {
    /**
```

```
     * @dev Emitted when the administration has been transferred.
     * @param previousAdmin Address of the previous admin.
     * @param newAdmin Address of the new admin.
     */
    event AdminChanged(address previousAdmin, address newAdmin);

    /**
     * @dev Storage slot with the admin of the contract.
     * This is the keccak-256 hash of "org.zeppelinos.proxy.admin", and is
     * validated in the constructor.
     */
    bytes32
        private                 constant                 ADMIN_SLOT                 =
0x10d6a54a4754c8869d6886b5f5d7fbfa5b4522237ea5c60d11bc4e7a1ff9390b;

    /**
     * @dev Modifier to check whether the `msg.sender` is the admin.
     * If it is, it will run the function. Otherwise, it will delegate the call
     * to the implementation.
     */
    modifier ifAdmin() {
        if (msg.sender == _admin()) {
            _;
        } else {
            _fallback();
        }
    }

    /**
     * @dev Contract constructor.
     * It sets the `msg.sender` as the proxy administrator.
     * @param implementationContract address of the initial implementation.
     */
    constructor(address implementationContract)
        public
        UpgradeabilityProxy(implementationContract)
    {
        assert(ADMIN_SLOT == keccak256("org.zeppelinos.proxy.admin"));

        _setAdmin(msg.sender);
    }

    /**
     * @return The address of the proxy admin.
     */
    function admin() external view returns (address) {
        return _admin();
    }

    /**
     * @return The address of the implementation.
     */
    function implementation() external view returns (address) {
        return _implementation();
    }

    /**
     * @dev Changes the admin of the proxy.
     * Only the current admin can call this function.
     * @param newAdmin Address to transfer proxy administration to.
     */
    function changeAdmin(address newAdmin) external ifAdmin {
        require(
            newAdmin != address(0),
            "Cannot change the admin of a proxy to the zero address"
        );
        emit AdminChanged(_admin(), newAdmin);
        _setAdmin(newAdmin);
    }

    /**
     * @dev Upgrade the backing implementation of the proxy.
     * Only the admin can call this function.
     * @param newImplementation Address of the new implementation.
     */
    function upgradeTo(address newImplementation) external ifAdmin {
        _upgradeTo(newImplementation);
    }

    /**
     * @dev Upgrade the backing implementation of the proxy and call a function
     * on the new implementation.
     * This is useful to initialize the proxied contract.
     * @param newImplementation Address of the new implementation.
     * @param data Data to send as msg.data in the low level call.
     * It should include the signature and the parameters of the function to be
     * called, as described in
```

```
     * https://solidity.readthedocs.io/en/develop/abi-spec.html#function-selector-and-argument-encoding.
     */
    function upgradeToAndCall(address newImplementation, bytes calldata data)
        external
        payable
        ifAdmin
    {
        _upgradeTo(newImplementation);
        // prettier-ignore
        // solhint-disable-next-line avoid-low-level-calls
        (bool success,) = address(this).call{value: msg.value}(data);
        // solhint-disable-next-line reason-string
        require(success);
    }

    /**
     * @return adm The admin slot.
     */
    function _admin() internal view returns (address adm) {
        bytes32 slot = ADMIN_SLOT;

        assembly {
            adm := sload(slot)
        }
    }

    /**
     * @dev Sets the address of the proxy admin.
     * @param newAdmin Address of the new proxy admin.
     */
    function _setAdmin(address newAdmin) internal {
        bytes32 slot = ADMIN_SLOT;

        assembly {
            sstore(slot, newAdmin)
        }
    }

    /**
     * @dev Only fall back when the sender is not the admin.
     */
    function _willFallback() internal override {
        require(
            msg.sender != _admin(),
            "Cannot call fallback function from the proxy admin"
        );
        super._willFallback();
    }
}

// File: contracts/v1/FiatTokenProxy.sol

pragma solidity 0.6.12;


/**
 * @title FiatTokenProxy
 * @dev This contract proxies FiatToken calls and enables FiatToken upgrades
 */
contract FiatTokenProxy is AdminUpgradeabilityProxy {
    constructor(address implementationContract)
        public
        AdminUpgradeabilityProxy(implementationContract)
    {}
}
```

# 5. Appendix B：Vulnerability rating standard

| *Smart contract vulnerability rating standards* | |
|---|---|
| **Level** | **Level Description** |
| **High** | Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow vulnerability that can cause the value of tokens to zero, false recharge vulnerability that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.;<br>Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of key function access control, etc.<br>Vulnerabilities that can cause the token contract to not work properly, such as: a denial of service vulnerability caused by sending ETH to a malicious address, and a denial of service vulnerability caused by gas exhaustion. |
| **Medium** | High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc. |
| **Low** | Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of ETH or tokens to trigger, vulnerabilities where attackers cannot directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk Wait. |

# 6. Appendix C：Introduction to auditing tools

## 6.1. Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

## 6.2. Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

## 6.3. securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a specific

language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

## 6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

## 6.5. MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

## 6.6. ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

## 6.7. ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 6.8. Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 6.9. Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of

Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

KNOWNSEC

Beijing KnownSec Information Technology Co., Ltd.


| | |
|---|---|
| Advisory telephone | +86(10)400 060 9587 |
| E-mail | sec@knownsec.com |
| Website | www.knownsec.com |
| Address | wangjing soho T2-B2509,Chaoyang District, Beijing |