

TLB staking 安全审计报告

TLB



审计结果：通过

文档信息

文档名称	文档版本号	文档编号	保密级别
TLB staking审计报告	V1.0	【TLB staking-2021-7-1】	公开

审计声明

蓝莓安全团队仅就本报告出具前已经发生或存在的事实出具本报告。对于出具以后发生或存在的事实，蓝莓无法判断其智能合约安全状况，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于信息提供者截至本报告出具时向蓝莓提供的文件和资料（简称“已提供资料”）。蓝莓假设：已提供资料不存在缺失、被篡改、删减或隐瞒的情形。如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，蓝莓对由此而导致的损失和不利影响不承担任何责任。

审计不会对代码的实用性、代码的安全性、商业模式的适用性、商业模式的监管制度或任何其他有关合约适用性的说明以及合约在无错状态的行为做出声明或担保。

目录

文档信息	2
审计声明	2
1. 综述	1
2. 代码漏洞分析	2
3. 代码审计结果分析	3
重入攻击检测【通过】	3
数值溢出检测【通过】	3
访问控制检测【通过】	4
返回值调用验证【通过】	4
错误使用随机数【通过】	5
事务顺序依赖【通过】	5
拒绝服务攻击【通过】	6
逻辑设计缺陷【通过】	6
假充值漏洞【通过】	6
增发代币漏洞【通过】	7
冻结账户绕过【通过】	7
4. 附录 A: 合约代码	8
5. 附录 B: 漏洞风险评级标准	14
6. 附录 C: 漏洞测试工具简介	15
Manticore	15
Oyente	15
securify.sh	15
Echidna	16
MAIAN	16
TLBersplay	16
ida-evm	16
Remix-ide	16

1. 综述

本次报告有效测试时间为 2021 年 1 月 18 日，针对 **TLB 智能合约代码** 的安全性和规范性进行审计并以此作为报告统计依据。

此次测试中，蓝莓安全团队对智能合约的常见漏洞（见第三章节）进行了全面的分析， TLB 合约代码符合TRC-20规范，未发现已知漏洞，故综合评定为**通过**。

本次智能合约安全审计结果：通过

由于本次测试过程在非生产环境下进行，所有代码均为最新版本，测试过程均与相关接口人进行沟通，并在操作风险可控的情况下进行相关测试操作，以规避测试过程中的生产运营风险、代码安全风险。

本次测试的目标信息：

模块名称	
Token 名称	TLB
代码类型	代币代码
合约地址	TMWvPDoAzhtbAboizvXTrrDLj6hhL1
区块地址	https://tronscan.io/#/token20/TMWvPDoAzhtbAboizvXTrrDLj6hhL1
代码语言	Solidity

2. 代码漏洞分析

漏洞等级分布

本次漏洞风险按等级统计：

漏洞风险等级个数统计表			
高危	中危	低危	通过
0	0	0	11

审计结果汇总

审计结果			
测试项目	测试内容	状态	描述
智能合约	重入攻击检测	通过	检查 call.value() 函数使用安全
	数值溢出检测	通过	检查 add 和 sub 函数使用安全
	访问控制缺陷检测	通过	检查各操作访问权限控制
	未验证返回值的调用	通过	检查转账方法看是否验证返回值
	错误使用随机数检测	通过	检查是否具备统一的内容过滤器
	事务顺序依赖检测	通过	检查事务顺序依赖
	拒绝服务攻击检测	通过	检查代码在使用资源时是否存在资源滥用问题
	逻辑设计缺陷检测	通过	检查智能合约代码中与业务设计相关的安全问题
	假充值漏洞检测	通过	检查智能合约代码中是否存在假充值漏洞
	增发代币漏洞检测	通过	检查智能合约代码中是否存在增发代币的功能
	冻结账户绕过	通过	检查智能合约代码中是否存在冻结账户绕过问题

3. 代码审计结果分析

重入攻击检测 【通过】

重入漏洞是最著名的TRC20智能合约漏洞，曾导致区块的分叉（The DAO hack）。

Solidity 中的call.value()函数在被用来发送 TLBer 的时候会消耗它接收到的所有 gas，当调用 call.value()函数发送 TLBer 的操作发生在实际减少发送者账户的余额之前时，就会存在重入攻击的风险。

检测结果：经检测，合约代码中不存在相关 call 外部合约调用。

安全建议：无。

数值溢出检测 【通过】

智能合约中的算数问题是指整数溢出和整数下溢。

Solidity 最多能处理 256 位的数字 ($2^{256}-1$)，最大数字增加 1 会溢出得到 0。同样，当数字为无符号类型时，0 减去 1 会下溢得到最大数字值。整数溢出和下溢不是一种新类型的漏洞，但它们在智能合约中尤其危险。溢出情况会导致不正确的结果，特别是如果可能性未被预期，可能会影响程序的可靠性和安全性。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

访问控制检测 【通过】

访问控制缺陷是所有程序中都可能存在的安全风险，智能合约也同样会存在类似问题，著名的 Parity Wallet 智能合约就受到过该问题的影响。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

返回值调用验证 【通过】

此问题多出现在和转币相关的智能合约中，故又称作静默失败发送或未经检查发送。

在 Solidity 中存在 transfer()、send()、call.value() 等转币方法，都可以用于向某一地址发送 TLB er，其区别在于： transfer 发送失败时会 throw，并且进行状态回滚；只会传递 2300gas 供调用，防止重入攻击； send 发送失败时会返回 false；只会传递 2300gas 供调用，防止重入攻击； call.value 发送失败时会返回 false；传递所有可用 gas 进行调用（可通过传入 gas_value 参数进行限制），不能有效防止重入攻击。

如果在代码中没有检查以上 send 和 call.value 转币函数的返回值，合约会继续执行后面的代码，可能由于 TLB er 发送失败而导致意外的结果。

检测结果：经检测，智能合约代码中不存在相关漏洞。

安全建议：无。

错误使用随机数 【通过】

智能合约中可能需要使用随机数，虽然 Solidity 提供的函数和变量可以访问明显难以预测的值，如 `block.number` 和 `block.timestamp`，但是它们通常或者比看起来更公开，或者受到矿工的影响，即这些随机数在一定程度上是可预测的，所以恶意用户通常可以复制它并依靠其不可预知性来攻击该功能。

检测结果：经检测，智能合约代码中不存在该问题。

安全建议：无。

事务顺序依赖 【通过】

由于矿工总是通过代表外部拥有地址 (EOA) 的代码获取 gas 费用，因此用户可以指定更高的费用以便更快地开展交易。由于以太坊区块链是公开的，每个人都可以看到其他人未决交易的内容。这意味着，如果某个用户提交了一个有价值的解决方案，恶意用户可以窃取该解决方案并以较高的费用复制其交易，以抢占原始解决方案。

检测结果：经检测，TLB 代币合约中的 `approve` 函数中不存在事务顺序依赖攻击风险。

安全建议：无。

拒绝服务攻击 【通过】

在以太坊的世界中，拒绝服务是致命的，遭受该类型攻击的智能合约可能永远无法恢复正常工作状态。导致智能合约拒绝服务的原因可能有很多种，包括在作为交易接收方时的恶意行为，人为增加计算功能所需 gas 导致 gas 耗尽，滥用访问控制访问智能合约的 private 组件，利用混淆和疏忽等等。

检测结果：经检测，智能合约代码中不存在相关漏洞。

安全建议：无。

逻辑设计缺陷 【通过】

检测智能合约代码中与业务设计相关的安全问题。

检测结果：经检测，智能合约代码中不存在相关漏洞。

安全建议：无。

假充值漏洞 【通过】

在代币合约的 transfer 函数对转账发起人(msg.sender)的余额检查用的是 if 判断方式，当 balances[msg.sender] < value 时进入 else 逻辑部分并 return false，最终没有抛出异常，我们认为仅 if/else 这种温和的判断方式在 transfer 这类敏感函数场景中是一种不严谨的编码方式。

检测结果：经检测，合约代码中不存在相关漏洞。

安全建议：无。

增发代币漏洞 【通过】

检测在初始化代币总量后，代币合约中是否存在可能使代币总量增加的函数。

检测结果：经检测，TLB 代码中不存在增发代币漏洞。

安全建议：无

冻结账户绕过 【通过】

检测代币合约中在转移代币时，是否存在未校验代币来源账户、发起账户、目标账户是否被冻结的操作。

检测结果：经检测，智能合约代码中不存在该问题。

安全建议：无。

4. 附录 A：合约代码

备注：审计意见及建议见代码注释 //AUDIT//.....

```
//AUDIT// 合约不存在溢出、条件竞争问题

pragma solidity ^0.3.11;

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
    address public owner;

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

    /**
     * @dev The Ownable constructor sets the original `owner` of the contract to the sender
     * account.
     */
    constructor() public {
        owner = msg.sender;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(msg.sender == owner);
    }

    /**
     * @dev Allows the current owner to transfer control of the contract to a newOwner.
     * @param newOwner The address to transfer ownership to.
     */
    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
```

```

/**
 * @dev Multiplies two numbers, throws on overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
    if (a == 0) {
        return 0;
    }
    c = a * b;
    assert(c / a == b);
    return c;
}

/**
 * @dev Integer division of two numbers, truncating the quotient.
 */
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    // uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return a / b;
}

/**
 * @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend).
 */
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
}

/**
 * @dev Adds two numbers, throws on overflow.
 */
function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
    c = a + b;
    assert(c >= a);
    return c;
}

/** 
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
 * @dev see https://github.com/GECSereum/EIPs/issues/179
 */
contract ERC20Basic {
    function totalSupply() public view returns (uint256);

    function balanceOf(address who) public view returns (uint256);

    function transfer(address to, uint256 value) public returns (bool);

    event Transfer(address indexed from, address indexed to, uint256 value);
}

/** 
 * @title Basic token
 */

```

```

* @dev Basic version of StandardToken, with no allowances.
*/
contract BasicToken is ERC20Basic {

    using SafeMath for uint256;

    mapping(address => uint256) balances;

    uint256 totalSupply_;

    /**
     * @dev total number of tokens in existence
     */
    function totalSupply() public view returns (uint256) {
        return totalSupply_;
    }

    /**
     * @dev transfer token for a specified address
     * @param _to The address to transfer to.
     * @param _value The amount to be transferred.
     */
    function transfer(address _to, uint256 _value) public returns (bool) {
        require(_to != address(0));
        require(_value <= balances[msg.sender]);

        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
        return true;           //AUDIT//    返回值符合 ERC20 规范
    }

    /**
     * @dev Gets the balance of the specified address.
     * @param _owner The address to query the the balance of.
     * @return An uint256 representing the amount owned by the passed address.
     */
    function balanceOf(address _owner) public view returns (uint256) {
        return balances[_owner];
    }

}

/**
 * @title Burnable Token
 * @dev Token that can be irreversibly burned (destroyed).
 */
contract BurnableToken is BasicToken {

    event Burn(address indexed burner, uint256 value);

    /**
     * @dev Burns a specific amount of tokens.
     * @param _value The amount of token to be burned.
     */
    function burn(uint256 _value) public {
        _burn(msg.sender, _value);
    }

    function _burn(address _who, uint256 _value) internal {
        require(_value <= balances[_who]);
    }
}

```

```

// no need to require value <= totalSupply, since that would imply the
// sender's balance is greater than the totalSupply, which *should* be an assertion
failure

    balances[_who] = balances[_who].sub(_value);
    totalSupply_ = totalSupply_.sub(_value);
    emit Burn(_who, _value);
    emit Transfer(_who, address(0), _value);
}
}

// File: contracts/token/ERC20/ERC20.sol

/**
* @title ERC20 interface
* @dev see https://github.com/GECSereum/EIPs/issues/20
*/
contract ERC20 is ERC20Basic {
    function allowance(address owner, address spender) public view returns (uint256);

    function transferFrom(address from, address to, uint256 value) public returns (bool);

    function approve(address spender, uint256 value) public returns (bool);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}

/**
* @title Standard ERC20 token
*
* @dev Implementation of the basic standard token.
* @dev https://github.com/GECSereum/EIPs/issues/20
* @dev Based on code by FirstBlood: https://github.com/Firstbloodio/token/blob/master/smar
t_contract/FirstBloodToken.sol
*/
contract StandardToken is ERC20, BasicToken {

    mapping(address => mapping(address => uint256)) internal allowed;

    /**
     * @dev Transfer tokens from one address to another
     * @param _from address The address which you want to send tokens from
     * @param _to address The address which you want to transfer to
     * @param _value uint256 the amount of tokens to be transferred
     */
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool)
    {
        require(_to != address(0));
        require(_value <= balances[_from]);
        require(_value <= allowed[_from][msg.sender]);

        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        emit Transfer(_from, _to, _value);
        return true;           //AUDIT//    返回值符合 ERC20 规范
    }

}

```

```

    * @dev Approve the passed address to spend the specified amount of tokens on behalf of
msg.sender.
    *
    * Beware that changing an allowance with this mGECsod 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 v
alue afterwards:
    * https://github.com/GECSereum/EIPs/issues/20#issuecomment-263524729
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
function approve(address _spender, uint256 _value) public returns (bool) {
    // To change the approve amount you first have to reduce the addresses'
    // allowance to zero by calling `approve(_spender, 0)` if it is not
    // already 0 to mitigate the race condition described here:
    // https://github.com/GECSereum/EIPs/issues/20#issuecomment-263524729
    if ((_value != 0) && (allowed[msg.sender][_spender] != 0)) revert();

    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}

/**
 * @dev Function to check the amount of tokens that an owner allowed to a spender.
 * @param _owner address The address which owns the funds.
 * @param _spender address The address which will spend the funds.
 * @return A uint256 specifying the amount of tokens still available for the spender.
 */
function allowance(address _owner, address _spender) public view returns (uint256) {
    return allowed[_owner][_spender];
}

/**
 * @dev Increase the amount of tokens that an owner allowed to a spender.
 *
 * approve should be called when allowed[_spender] == 0. To increment
 * allowed value is better to use this function to avoid 2 calls (and wait until
 * the first transaction is mined)
 * From MonolithDAO Token.sol
 * @param _spender The address which will spend the funds.
 * @param _addedValue The amount of tokens to increase the allowance by.
 */
function increaseApproval(address _spender, uint _addedValue) public returns (bool) {
    allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
    emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
}

/**
 * @dev Decrease the amount of tokens that an owner allowed to a spender.
 *
 * approve should be called when allowed[_spender] == 0. To decrement
 * allowed value is better to use this function to avoid 2 calls (and wait until
 * the first transaction is mined)
 * From MonolithDAO Token.sol
 * @param _spender The address which will spend the funds.
 * @param _subtractedValue The amount of tokens to decrease the allowance by.
*/

```

```

    */
    function decreaseApproval(address _spender, uint _subtractedValue) public returns (bool)
    {
        uint oldValue = allowed[msg.sender][_spender];
        if (_subtractedValue > oldValue) {
            allowed[msg.sender][_spender] = 0;
        } else {
            allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
        }
        emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
        return true;
    }

}

contract MintableToken is StandardToken, Ownable {
    /**
     * @dev Function to mint tokens
     * @param _to The address that will receive the minted tokens.
     * @param _amount The amount of tokens to mint.
     * @return A boolean that indicates if the operation was successful.
    */
    function mint(
        address _to,
        uint256 _amount,
        bool _total
    )
    public
    onlyOwner
    returns (bool)
    {
        if (_total) {
            totalSupply_ = totalSupply_.add(_amount);
        }
        balances[_to] = balances[_to].add(_amount);
        emit Transfer(address(0), _to, _amount);
        return true;
    }
}

contract GECssion is StandardToken, BurnableToken, MintableToken {
    // Constants
    string public constant name = "TLB sion";
    string public constant symbol = " TLB ";
    uint8 public constant decimals = 18;
    uint256 public constant INITIAL_SUPPLY = 100000000 * (10 ** uint256(decimals));
    address constant holder = 0xb002247648A4193A23AbB414b2437E34812114a2;

    constructor() public {
        totalSupply_ = INITIAL_SUPPLY;
        balances[holder] = INITIAL_SUPPLY;
        emit Transfer(address(0), holder, INITIAL_SUPPLY);
    }

    function() external paGECSble {
        revert();
    }
}

```

5. 附录 B：漏洞风险评级标准

漏洞评级	漏洞评级说明
高危漏洞	<p>能直接造成代币合约或用户资金损失的漏洞，如：能造成代币价值归零的数值溢出漏洞、能造成交易所损失代币的假充值漏洞、能造成合约账户损失代币的重入漏洞等；</p> <p>能造成代币合约归属权丢失的漏洞，如：关键函数的访问控制缺陷、call 注入导致关键函数访问控制绕过等；</p> <p>能造成代币合约无法正常工作的漏洞，如：因向恶意地址发送区块导致的拒绝服务漏洞、因 gas 耗尽导致的拒绝服务漏洞。</p>
中危漏洞	<p>需要特定地址才能触发的高风险漏洞，如代币合约拥有者才能触发的数值溢出漏洞等；非关键函数的访问控制缺陷、不能造成直接资金损失的逻辑设计缺陷等。</p>
低危漏洞	<p>难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量GECS 或代币才能触发的数值溢出漏洞、触发数值溢出后攻击者无法直接获利的漏洞、通过指定高 gas 触发的事物顺序依赖风险等。</p>

6. 附录 C：漏洞测试工具简介

Manticore

Manticore 是一个分析二进制文件和智能合约的符号执行工具，Manticore 包含一个符号以太坊虚拟机 (EVM)，一个 EVM 反汇编器/汇编器以及一个用于自动编译和分析 Solidity 的方便界面。它还集成了 TLB ersplay，用于 EVM 字节码的 Bit of Traits of Bits 可视化反汇编程序，用于可视化分析。与二进制文件一样，

Manticore 提供了一个简单的命令行界面和一个用于分析 EVM 字节码的 Python API。

Oyente

Oyente 是一个智能合约分析工具，Oyente 可以用来检测智能合约中常见的 bug，比如 reentrancy、事务排序依赖等等。更方便的是，Oyente 的设计是模块化的，所以这让高级用户可以实现并插入他们自己的检测逻辑，以检查他们的合约中自定义的属性。

securify.sh

Securify 可以验证以太坊智能合约常见的安全问题，例如交易乱序和缺少输入验证，它在全自动化的同时分析程序所有可能的执行路径，此外，Securify 还具有用于指定漏洞的特定语言，这使 Securify 能够随时关注当前的安全性和其他可靠性问题。

Echidna

Echidna 是一个为了对 EVM 代码进行模糊测试而设计的 Haskell 库。

MAIAN

MAIAN 是一个用于查找以太坊智能合约漏洞的自动化工具，Maian 处理合约的字节码，并尝试建立一系列交易以找出并确认错误。

TLBersplay

TLBersplay 是一个 EVM 反汇编器，其中包含了相关分析工具。

ida-evm

ida-evm 是一个针对以太坊虚拟机 (EVM) 的 IDA 处理器模块。

Remix-ide

Remix 是一款基于浏览器的编译器和 IDE，可让用户使用 Solidity 语言构建以太坊合约并调试交易。



BLUEBERRY SECURITY