知道创宇

密级 ★ | 项目组公开

智能合约审计报告

安全状态

安全

★ ★ ★ ★ ★

主测人： 知道创宇区块链安全研究团队

版本说明

修订内容	时间	修订者	版本号
编写文档	20210301	知道创宇区块链安全研究团队	V1.0

文档信息

文档名称	文档版本	报告编号	保密级别
Golff 智能合约审计报告	V1.0		项目组公开

声明

目录

1. 综述

本次报告有效测试时间是从 2021 年 02 月 25 日开始到 2021 年 03 月 01 日结束，在此期间针对 **Golff 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，知道创宇工程师对智能合约的常见漏洞（见第三章节）进行了全面的分析，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，所有代码均为最新备份，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。

本次审计的报告信息：

报告编号：

报告查询地址链接:

https://attest.im/attestation/searchResult?qurey=

本次审计的目标信息：

条目	描述	
Token 名称	Golff	
合约地址	golff-heco-vault-master	https://github.com/golfffinance/golff-heco-vault.git
代码类型	代币代码、DeFi 协议代码、HECO 智能合约代码	
代码语言	Solidity	

合约文件及哈希：

合约文件	MD5
GOFControllerV1.sol	A8A172742E6CE06E63351840B50D6D74

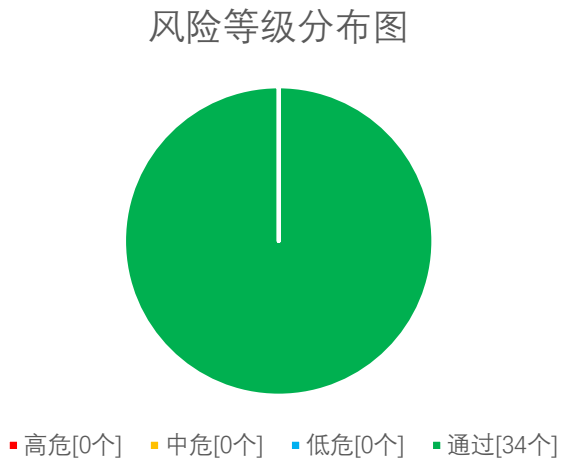
IHecoPool.sol	E8EE71F966B013E789CC724816F432E4
IHswapV2Callee.sol	CCC47D99F32D369DD72BC9A5472A2615
IMdexERC20.sol	BC6414685851EB298929D211CBF8AD5D
IMdexFactory.sol	839B7AB157EAE19AB15CE1952BFE9EBF
IMdexPair.sol	87B421251B975E8551DCA381C92046FD
IMdexRouter.sol	D1E1DA23D96202CD5B1614FC5776393D
IMdx.sol	5D6A7FB6DCFDFEE792481434E9B91C03
ISwapMining.sol	DBF4B081668434FFA5307C4159739 4C2
IERC20Detailed.sol	013EBC4B2BB34D2AB68ED65FB6BD3F6C
IGOFController.sol	17AE917B858C3F1BE1603FA79AC19398
IGOFStrategy.sol	BA87DCE2A9C67B87240B36D3ACD1F315
IWHT.sol	9A13F90AB3EAC04602D004AD0B32D0AA
HecoPool.sol	96486B2365A67B1374CAA73FE52F6086
MdxToken.sol	44C6907625A416906626FEA6051D462B
StrategyForMDEX.sol	A8B33BB191CEAD5A30F657DF24349D0C
MockToken.sol	66829E6FF824AADE72D6B99EAE812B51
WHT.sol	72D2B0277DD99644CC006760BE076CE2
GofVault.sol	6BFAE7E9F760FF95E59A3722B562E14F
GofVaultHT.sol	E4EDF8B97E1C1A188327135DF1CACD6B
Migrations.sol	CA8D6CA8A6EDF34F149A5095A8B074C9

2. 代码漏洞分析

2.1 漏洞等级分布

本次漏洞风险按等级统计：

安全风险等级个数统计表			
高危	中危	低危	通过
0	0	0	34



2.2 审计结果汇总说明

审计结果			
审计项目	审计内容	状态	描述
业务安全性检测	控制器合约变量及构造函数	通过	经检测，不存在安全问题。
	控制器合约 yearn 功能	通过	经检测，不存在安全问题。
	GofVault 合约构造函数	通过	经检测，不存在安全问题。
	GofVault 合约 deposit 函数	通过	经检测，不存在安全问题。
	GofVault 合约 withdraw 函数	通过	经检测，不存在安全问题。
	Strategy 合约 doHarvest 函数	通过	经检测，不存在安全问题。
	Strategy 合约 withdraw 函数	通过	经检测，不存在安全问题。
代码基本漏洞检测	编译器版本安全	通过	经检测，不存在该安全问题。
	冗余代码	通过	经检测，不存在该安全问题。
	安全算数库的使用	通过	经检测，不存在该安全问题。
	不推荐的编码方式	通过	经检测，不存在该安全问题。
	require/assert 的合理使用	通过	经检测，不存在该安全问题。
	fallback 函数安全	通过	经检测，不存在该安全问题。
	tx.orgin 身份验证	通过	经检测，不存在该安全问题。
	owner 权限控制	通过	经检测，不存在该安全问题。
	gas 消耗检测	通过	经检测，不存在该安全问题。
	call 注入攻击	通过	经检测，不存在该安全问题。
	低级函数安全	通过	经检测，不存在该安全问题。
	增发代币漏洞	通过	经检测，不存在该安全问题。
	访问控制缺陷检测	通过	经检测，不存在该安全问题。
	数值溢出检测	通过	经检测，不存在该安全问题。
	算数精度误差	通过	经检测，不存在该安全问题。
	错误使用随机数检测	通过	经检测，不存在该安全问题。

	不安全的接口使用	通过	经检测，不存在该安全问题。
	变量覆盖	通过	经检测，不存在该安全问题。
	未初始化的存储指针	通过	经检测，不存在该安全问题。
	返回值调用验证	通过	经检测，不存在该安全问题。
	交易顺序依赖检测	通过	经检测，不存在该安全问题。
	时间戳依赖攻击	通过	经检测，不存在该安全问题。
	拒绝服务攻击检测	通过	经检测，不存在该安全问题。
	假充值漏洞检测	通过	经检测，不存在该安全问题。
	重入攻击检测	通过	经检测，不存在该安全问题。
	重放攻击检测	通过	经检测，不存在该安全问题。
	重排攻击检测	通过	经检测，不存在该安全问题。

3. 业务安全性检测

3.1. 控制器合约变量及构造函数【通过】

审计分析：控制器 GOFControllerV1.sol 合约变量定义及构造函数设计合理。

```
contract GOFControllerV1 is Ownable{
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public strategist;
    address public onesplit;
    address public rewards;
    address public factory;
    mapping(address => address) public vaults;
    mapping(address => address) public strategies;
    mapping(address => mapping(address => address)) public converters;
    mapping(address => mapping(address => bool)) public approvedStrategies;

    uint public split = 500;
    uint public constant max = 10000;
    address constant public wht = address(0x8543A3E99174913cEeaF8b10e890bC99c4a89420);

    constructor(address _rewards) public {
        strategist = tx.origin;
        onesplit = address(0xbb87D38beEecac5eB602791703763722e3E9F359);
        rewards = _rewards;
    }
    ...
}
```

安全建议：无。

3.2. 控制器合约 yearn 功能【通过】

审计分析：控制器 GOFControllerV1.sol 合约的 yearn 功能是收取指定策略的指定代币，从收益中抽成提取奖励后剩余再投入该策略中赚取利息。

```
function yearn(address _strategy, address _token) public checkStrategist{
    // This contract should never have value in it, but just incase since this is a public call
    uint _before = IERC20(_token).balanceOf(address(this));
    IGOFStrategy(_strategy).withdraw(_token);
    uint _after =  IERC20(_token).balanceOf(address(this));
    if (_after > _before) {
        uint _amount = _after.sub(_before);
        address _want = IGOFStrategy(_strategy).want();

        _before = IERC20(_want).balanceOf(address(this));
        IERC20(_token).safeApprove(onesplit, 0);
        IERC20(_token).safeApprove(onesplit, _amount);

        //swap by imdex
        address[] memory swap2TokenRouting;
        swap2TokenRouting[0] = _token;
        swap2TokenRouting[1] = wht;
        swap2TokenRouting[2] = _want;
        IMdexRouter(onesplit).swapExactTokensForTokens(_amount,                    0,
swap2TokenRouting, address(this), now.add(1800));

        _after = IERC20(_want).balanceOf(address(this));
        if (_after > _before) {
            _amount = _after.sub(_before);
            uint _reward = _amount.mul(split).div(max);
            earn(_want, _amount.sub(_reward));
            IERC20(_want).safeTransfer(rewards, _reward);
```

```
            }
        }
    }
```

安全建议：无。

3.3. GofVault 合约构造函数【通过】

审计分析：以 GofVault.sol 为例，GOFVault 类合约提供以其他代币为的 iToken 流动性挖矿的功能。

```
    constructor (
        address _token,
        string memory _symbol,
        address _controller,
        uint _earnLowerlimit
    ) public ERC20(
        string(abi.encodePacked("Golff ", ERC20(_token).name())),
        string(abi.encodePacked("GH-", _symbol))
    ) {
        token = IERC20(_token);
        controller = _controller;
        earnLowerlimit = _earnLowerlimit;
        _setupDecimals(ERC20(_token).decimals());
    }
```

安全建议：无。

3.4. GofVualt 合约 deposit 函数【通过】

审计分析：以 GofVault.sol 为例，despoit 函数用于存入指定代币转化为相应的流动性生息代币 iToken 以获取收益利息。

```
function deposit(uint _amount) public {
```

```
        uint _pool = balance();
        uint _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint _after = token.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens
        uint _shares = 0;
        if (totalSupply() == 0) {
            _shares = _amount;
        } else {
            _shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, _shares);
        if (token.balanceOf(address(this)) > earnLowerlimit){
            earn();
        }
    }
```

安全建议：无。

3.5. GofVualt 合约 withdraw 函数【通过】

审计分析：以 GofVault.sol 为例，withdraw 函数用于提现策略合约中的流动性收益。

```
    function withdraw(uint _shares) public {
        uint r = (balance().mul(_shares)).div(totalSupply());
        _burn(msg.sender, _shares);

        uint b = token.balanceOf(address(this));
        if (b < r) {
            uint _withdraw = r.sub(b);
            IGOFController(controller).withdraw(address(token), _withdraw);
            uint _after = token.balanceOf(address(this));
```

```
            uint _diff = _after.sub(b);
            if (_diff < _withdraw) {
                r = b.add(_diff);
            }
        }

        token.safeTransfer(msg.sender, r);
    }
```

安全建议：无。

3.6. Strategy 合约 doHarvest 函数【通过】

审计分析：以 StrategyForMDEX.sol 为例，doHarvest 函数用于按指定策略进行投资生息。

```
function doHarvest() internal {
        //判断收益情况
        doswap();
        dosplit();//分 gof
    }

    function doswap() internal {
        uint256 _balance = IERC20(output).balanceOf(address(this));
        if(_balance > 0){
            uint256 _2token = _balance.mul(90).div(100); //90%
            uint256 _2gof = _balance.mul(10).div(100);    //10%
            IMdexRouter(mdexrouter).swapExactTokensForTokens(_2token,                0,
swap2TokenRouting, address(this), now.add(1800));
            IMdexRouter(mdexrouter).swapExactTokensForTokens(_2gof,                  0,
swap2GOFRouting, address(this), now.add(1800));
        }
    }
```

```
function dosplit() internal{
        uint b = IERC20(gof).balanceOf(address(this));
        if(b > 0){
            uint _fee = b.mul(fee).div(max);
            uint _callfee = b.mul(callfee).div(max);
            uint _foundationfee = b.mul(foundationfee).div(max);
            IERC20(gof).safeTransfer(IGOFController(controller).rewards(), _fee);
            IERC20(gof).safeTransfer(msg.sender, _callfee);
            IERC20(gof).safeTransfer(foundationAddress, _foundationfee);

            if (burnfee >0){
                uint _burnfee = b.mul(burnfee).div(max);
                IERC20(gof).safeTransfer(burnAddress, _burnfee);
            }
        }
}
```

安全建议：无。

3.7. Strategy 合约 withdraw 函数【通过】

审计分析：以 StrategyForMDEX.sol 为例，withdraw 函数用于提现流动性代币至 IGOFVault 合约。

```
function withdraw(uint _amount) external {
        require(msg.sender == controller, "Golff:!controller");
        uint _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
```

```
        uint _fee = 0;
        if (withdrawalFee>0){
            _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IGOFController(controller).rewards(), _fee);
        }

        address _vault = IGOFController(controller).vaults(address(want));
        require(_vault != address(0), "Golff:!vault"); // additional protection so we don't burn the
funds
        IERC20(want).safeTransfer(_vault, _amount.sub(_fee));
    }

    // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint balance) {
        require(msg.sender == controller, "Golff:!controller");
        _withdrawAll();
        balance = IERC20(want).balanceOf(address(this));

        address _vault = IGOFController(controller).vaults(address(want));
        require(_vault != address(0), "Golff:!vault"); // additional protection so we don't burn the
funds
        IERC20(want).safeTransfer(_vault, balance);
    }

    function _withdrawAll() internal {
        uint256 balance = balanceOfPool();
        if(balance > 0){
            IHecoPool(hecoPool).withdraw(pid, balance);
            doHarvest();
        }
    }
```

安全建议：无。

4. 代码基本漏洞检测

4.1. 编译器版本安全【通过】

检查合约代码实现中是否使用了安全的编译器版本

检测结果：经检测，智能合约代码中制定了编译器版本 0.6.12 以上，不存在该安全问题。

安全建议：无。

4.2. 冗余代码【通过】

检查合约代码实现中是否包含冗余代码

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。

4.3. 安全算数库的使用【通过】

检查合约代码实现中是否使用了 SafeMath 安全算数库

检测结果：经检测，智能合约代码中已使用 SafeMath 安全算数库，不存在该安全问题。

安全建议：无。

4.4. 不推荐的编码方式【通过】

检查合约代码实现中是否有官方不推荐或弃用的编码方式

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。

4.5. require/assert 的合理使用【通过】

检查合约代码实现中 require 和 assert 语句使用的合理性

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。

4.6. fallback 函数安全【通过】

检查合约代码实现中是否正确使用 fallback 函数

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。

4.7. tx.origin 身份验证【通过】

tx.origin 是 Solidity 的一个全局变量，它遍历整个调用栈并返回最初发送调用（或事务）的帐户的地址。在智能合约中使用此变量进行身份验证会使合约容易受到类似网络钓鱼的攻击。

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。

4.8. owner 权限控制【通过】

检查合约代码实现中的 owner 是否具有过高的权限。例如，任意修改其他账户余额等。

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。

4.9. gas 消耗检测【通过】

检查 gas 的消耗是否超过区块最大限制

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。

4.10. call 注入攻击【通过】

call 函数调用时，应该做严格的权限控制，或直接写死 call 调用的函数。

检测结果：经检测，智能合约未使用 call 函数，不存在此漏洞。

安全建议：无。

4.11. 低级函数安全【通过】

检查合约代码实现中低级函数（call/delegatecall）的使用是否存在安全漏洞

call 函数的执行上下文是在被调用的合约中；而 delegatecall 函数的执行上下文是在当前调用该函数的合约中

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。

4.12. 增发代币漏洞【通过】

检查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，智能合约代码中存在增发代币的功能，但由于流动性挖矿需要增发代币，故通过。

安全建议：无。

4.13. 访问控制缺陷检测【通过】

合约中不同函数应设置合理的权限

检查合约中各函数是否正确使用了 public、private 等关键词进行可见性修饰，检查合约是否正确定义并使用了 modifier 对关键函数进行访问限制，避免越权导致的问题。

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。

4.14. 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。

4.15. 算术精度误差【通过】

Solidity 作为一门编程语言具备和普通编程语言相似的数据结构设计，比如：变量、常量、函数、数组、函数、结构体等等，Solidity 和普通编程语言也有一个较大的区别——Solidity 没有浮点型，且 Solidity 所有的数值运算结果都只会是整数，不会出现小数的情况，同时也不允许定义小数类型数据。合约中的数值运算必不可少，而数值运算的设计有可能造成相对误差，例如同级运算：5/2*10=20，而 5*10/2=25，从而产生误差，在数据更大时产生的误差也会更大，更明显。

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。

4.16. 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安全问题。

安全建议：无。

4.17. 不安全的接口使用【通过】

检查合约代码实现中是否使用了不安全的接口

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。

4.18. 变量覆盖【通过】

检查合约代码实现中是否存在变量覆盖导致的安全问题

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。

4.19. 未初始化的储存指针【通过】

在 solidity 中允许一个特殊的数据结构为 struct 结构体，而函数内的局部变量默认使用 storage 或 memory 储存。

而存在 storage(存储器)和 memory(内存)是两个不同的概念，solidity 允许指针指向一个未初始化的引用，而未初始化的局部 stroage 会导致变量指向其他储存变量，导致变量覆盖，甚至其他更严重的后果，在开发中应该避免在函数中初始化 struct 变量。

检测结果：经检测，智能合约代码不存在该问题。

安全建议：无。

4.20. 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 HT，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 HT 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该安全问题。

安全建议：无。

4.21. 交易顺序依赖【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，智能合约代码中不存在该安全问题。

```
function deposit() public {//knownsec// 流动性挖矿
    uint _want = IERC20(want).balanceOf(address(this));
    address _controller = For(fortube).controller();
    if (_want > 0) {
        //knownsec// 由于 HBTC 合约不能设置授权额为 0
        // IERC20(want).safeApprove(_controller, 0);
        IERC20(want).safeApprove(_controller, _want);
        For(fortube).deposit(want,_want);
    }
}
```

安全建议：无。

4.22. 时间戳依赖攻击【通过】

数据块的时间戳通常来说都是使用矿工的本地时间，而这个时间大约能有 900 秒的范围波动，当其他节点接受一个新区块时，只需要验证时间戳是否晚于之前的区块并且与本地时间误差在 900 秒以内。一个矿工可以通过设置区块的时间戳来尽可能满足有利于他的条件来从中获利。

检查合约代码实现中是否存在有依赖于时间戳的关键功能

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。

4.23. 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该安全问题。

安全建议：无。

4.24. 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，智能合约代码中不存在该安全问题。

安全建议：无。

4.25. 重入攻击检测【通过】

重入漏洞是最著名的以太坊智能合约漏洞，曾导致了以太坊的分叉（The DAO hack）。

Solidity 中的 call.value()函数在被用来发送 HT 的时候会消耗它接收到的所有 gas，当调用 call.value()函数发送 HT 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，智能合约代码中不存在该安全问题。

安全建议：无。

4.26. 重放攻击检测【通过】

合约中如果涉及委托管理的需求，应注意验证的不可复用性，避免重放攻击

在资产管理体系中，常有委托管理的情况，委托人将资产给受托人管理，委托人支付一定的费用给受托人。这个业务场景在智能合约中也比较普遍。。

检测结果：经检测，智能合约未使用 call 函数，不存在此漏洞。

安全建议：无。

4.27. 重排攻击检测【通过】

重排攻击是指矿工或其他方试图通过将自己的信息插入列表(list)或映射(mapping)中来与智能合约参与者进行“竞争”，从而使攻击者有机会将自己的

信息存储到合约中。

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。

5. 附录 A：合约代码

本次测试代码来源：

STAR.sol

```
/**
  *Submitted for verification at Etherscan.io on 2020-07-17
*/

pragma solidity ^0.5.8;

interface IERC20 {//knownsec// ERC20 代币标准接口
        function totalSupply() external view returns (uint);
        function balanceOf(address account) external view returns (uint);
        function transfer(address recipient, uint amount) external returns (bool);
        function allowance(address owner, address spender) external view returns (uint);
        function approve(address spender, uint amount) external returns (bool);
        function transferFrom(address sender, address recipient, uint amount) external returns (bool);
        event Transfer(address indexed from, address indexed to, uint value);
        event Approval(address indexed owner, address indexed spender, uint value);
}
```

6. 附录 B：安全风险评级标准

智能合约漏洞评级标准	
漏洞评级	**漏洞评级说明**
高危漏洞	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 HT 或代币的重入漏洞等； 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； 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 HT 导致的拒绝服务漏洞、因 gas 耗尽导致的拒绝服务漏洞。
中危漏洞	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。
低危漏洞	难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量 HT 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 触发的事务顺序依赖风险等。

7. 附录 C：智能合约安全审计工具简介

6.1 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 Ethersplay，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。

6.2 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。

6.3 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。

6.4 Echidna

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

6.5 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。

6.6 ethersplay

ethersplay 是一个 EVM 反汇编器，其中包含了相关分析工具。

6.7 ida-evm

ida-evm 是一个针对以太坊虚拟机（EVM）的 IDA 处理器模块。

6.8 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。

6.9 知道创宇区块链安全审计人员专用工具包

知道创宇渗透测试人员专用工具包，由知道创宇渗透测试工程师研发，收集和使用，包含专用于测试人员的批量自动测试工具，自主研发的工具、脚本或利用工具等。

知道创宇

北京知道创宇信息技术股份有限公司

咨询电话 +86(10)400 060 9587

邮　　箱 sec@knownsec.com

官　　网 www.knownsec.com

地　　址 北京市 朝阳区 望京 SOHO T2-B座-2509