KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: Knownsec blockchain security team

Version Summary

Content	Date	Version
Editing Document	20210305	V1.0

Report Information

Title	Version	Document Number	Type
WAH Smart Contract Audit Report	V1.0	**34df35af8f3f4693b703b25e6cfc2258**	Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from From March 04, 2021 to March 05, 2021 . During this period, the security and standardization of **the smart contract code of the WAH** 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). **The smart contract code of the WAH** 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.

Report information of this audit:

Report Number：34df35af8f3f4693b703b25e6cfc2258

Report query address link:

https://attest.im/attestation/searchResult?qurey=34df35af8f3f4693b703b25e6cfc2258

Target information of the WAH audit:

Target information	
Token name	WAH
Code type	Ethereum smart contract code
Code language	solidity

Contract documents and hash:

Contract documents	MD5

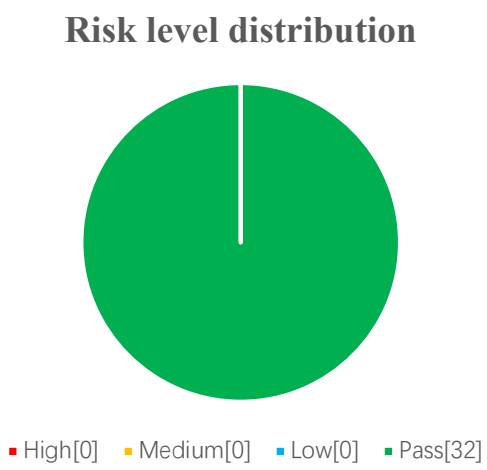
Migrations.sol	15a1bffb582537753dc9788968d5a6e9
Staking.sol	15f6e88d80e6b237bda6b16f638c5a0a
WAHToken.sol	ac57c5b3b52f13e19ad2044bea94bd9c
YieldFarm.sol	c0b3cfa67ccfce2f4e4660414f78596b
YieldFarmLP.sol	86195bbce2c863b4130638357c2e00c0

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	32



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**Staking contract deposit function**	Pass	After testing, there is no such safety vulnerability.
	Staking contract withdraw function	Pass	After testing, there is no such safety vulnerability.
	Staking contract emergencyWithdraw function	Pass	After testing, there is no such safety vulnerability.
	YieldFarm contract massHarvest function	Pass	After testing, there is no such safety vulnerability.
	YieldFarm contract harvest function	Pass	After testing, there is no such safety vulnerability.
Basic code vulnerability detection	**Compiler version security**	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.
	Not recommended encoding	Pass	After testing, there is no such safety vulnerability.
	Reasonable use of require/assert	Pass	After testing, there is no such safety vulnerability.
	fallback function safety	Pass	After testing, there is no such safety vulnerability.

	tx.oriigin authentication	Pass	After testing, there is no such safety vulnerability.
	Owner permission control	Pass	After testing, there is no such safety vulnerability.
	Gas consumption detection	Pass	After testing, there is no such safety vulnerability.
	call injection attack	Pass	After testing, there is no such safety vulnerability.
	Low-level function safety	Pass	After testing, there is no such safety vulnerability.
	Vulnerability of additional token issuance	Pass	After testing, there is no such safety vulnerability.
	Access control defect 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.
	Wrong use of random number detection	Pass	After testing, there is no such safety vulnerability.
	Unsafe interface use	Pass	After testing, there is no such safety vulnerability.
	Variable coverage	Pass	After testing, there is no such safety vulnerability.
	Uninitialized storage pointer	Pass	After testing, there is no such safety vulnerability.

	Return value call verification	Pass	After testing, there is no such safety vulnerability.
	Transaction order dependency detection	Pass	After testing, there is no such safety vulnerability.
	Timestamp dependent attack	Pass	After testing, there is no such safety vulnerability.
	Denial of service attack detection	Pass	After testing, there is no such safety vulnerability.
	Fake recharge vulnerability detection	Pass	After testing, there is no such safety vulnerability.
	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.

3. Analysis of code audit results

3.1. Staking contract deposit function【PASS】

Audit analysis: Perform security audits on the deposit logic audit in the Staking contract, check the validity of the incoming parameters, transfer the number of user pledges, and update the epoch and checkpoints existing in the epoch.

```
function deposit(address tokenAddress, uint256 amount) public nonReentrant {
        require(amount > 0, "Staking: Amount must be > 0"); //knownsec// The amount of pledge must be greater than 0

        IERC20 token = IERC20(tokenAddress);
        uint256 allowance = token.allowance(msg.sender, address(this));
        require(allowance >= amount, "Staking: Token allowance too small"); //knownsec// The authorized amount is greater than the pledged amount

        balances[msg.sender][tokenAddress] = balances[msg.sender][tokenAddress].add(amount); //knownsec// Update the user's pledge amount of the token

        token.transferFrom(msg.sender, address(this), amount);

        // epoch logic
        uint128 currentEpoch = getCurrentEpoch(); //knownsec// Get the current epoch
        uint128 currentMultiplier = currentEpochMultiplier(); //knownsec// Get the current Multiplier

        if (!epochIsInitialized(tokenAddress, currentEpoch)) { //knownsec// Determine whether the token in the current epoch has been initialized
            address[] memory tokens = new address[](1);
            tokens[0] = tokenAddress;
            manualEpochInit(tokens, currentEpoch); //knownsec// Initialize manually if not initialized
```

```
        }

        // update the next epoch pool size
        Pool storage pNextEpoch = poolSize[tokenAddress][currentEpoch + 1];
        pNextEpoch.size = token.balanceOf(address(this)); //knownsec// The size of the next epoch pool is the token balance in the current contract
        pNextEpoch.set = true; //knownsec// Set the next epoch pool as changed

        Checkpoint[] storage checkpoints = balanceCheckpoints[msg.sender][tokenAddress]; //knownsec// Record user checkpoint

        uint256 balanceBefore = getEpochUserBalance(msg.sender, tokenAddress, currentEpoch); //knownsec// Get the user's token balance in the current epoch

        // if there's no checkpoint yet, it means the user didn't have any activity
        // we want to store checkpoints both for the current epoch and next epoch because
        // if a user does a withdraw, the current epoch can also be modified and
        // we don't want to insert another checkpoint in the middle of the array as that could be expensive
        if (checkpoints.length == 0) {
            checkpoints.push(Checkpoint(currentEpoch, currentMultiplier, 0, amount)); //knownsec// If checkpoints.length is equal to 0, it means that the user has not pledged or has withdrawn cash. Set the current epoch user's starting balance to 0 and the pledge amount to amount

            // next epoch => multiplier is 1, epoch deposits is 0
            checkpoints.push(Checkpoint(currentEpoch + 1, BASE_MULTIPLIER, amount, 0)); //knownsec// Set the initial balance of the next epoch to amount, and the amount of pledge to 0
        } else {
            uint256 last = checkpoints.length - 1;

            // the last action happened in an older epoch (e.g. a deposit in epoch 3, current epoch is >=5)
            if (checkpoints[last].epochId < currentEpoch) { //knownsec// The epochid where the
```

```
checkpoint is located must be smaller than the current epochid
                uint128 multiplier = computeNewMultiplier( //knownsec// Calculate the new
multiplier
                    getCheckpointBalance(checkpoints[last]),
                    BASE_MULTIPLIER,
                    amount,
                    currentMultiplier
                );
                checkpoints.push(Checkpoint(currentEpoch, multiplier,
getCheckpointBalance(checkpoints[last]), amount)); //knownsec// Update the checkpoint in the
current epoch, the starting balance is the sum of the balances in all checkpoints
                checkpoints.push(Checkpoint(currentEpoch + 1, BASE_MULTIPLIER,
balances[msg.sender][tokenAddress], 0)); /knownsec// Set the checkpoint in the next epoch
            }
            // the last action happened in the previous epoch
            else if (checkpoints[last].epochId == currentEpoch) {
                checkpoints[last].multiplier = computeNewMultiplier(
                    getCheckpointBalance(checkpoints[last]),
                    checkpoints[last].multiplier,
                    amount,
                    currentMultiplier
                );
                checkpoints[last].newDeposits = checkpoints[last].newDeposits.add(amount);
//knownsec// Continue to pledge

                checkpoints.push(Checkpoint(currentEpoch + 1, BASE_MULTIPLIER,
balances[msg.sender][tokenAddress], 0));
            }
            // the last action happened in the current epoch
            else {
                if (last >= 1 && checkpoints[last - 1].epochId == currentEpoch) {
                    checkpoints[last - 1].multiplier = computeNewMultiplier(
                        getCheckpointBalance(checkpoints[last - 1]),
```

```
                    checkpoints[last - 1].multiplier,
                    amount,
                    currentMultiplier
                );
                checkpoints[last - 1].newDeposits = checkpoints[last - 1].newDeposits.add(amount);
            }

            checkpoints[last].startBalance = balances[msg.sender][tokenAddress]; //knownsec// The starting balance is set to the amount currently pledged by the user
        }
    }

    uint256 balanceAfter = getEpochUserBalance(msg.sender, tokenAddress, currentEpoch); //knownsec// Get the token balance in the current epoch after the user pledge

    poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.add(balanceAfter.sub(balanceBefore)); //knownsec// The size of the pool in the current eopch plus the difference before and after the pledge

    emit Deposit(msg.sender, tokenAddress, amount);
}
```

Recommendation： nothing.

3.2. Staking contract withdraw function【PASS】

Audit analysis: Perform security audits on the withdraw logic audit in the Staking contract, check the validity of the incoming parameters, transfer the amount of user pledges, and update the epoch and checkpoints existing in the epoch.

```
function withdraw(address tokenAddress, uint256 amount) public nonReentrant { //knownsec// Withdraw the amount of tokens to the user account
        require(balances[msg.sender][tokenAddress] >= amount, "Staking: balance too small"); //knownsec// The amount of balance that the user has pledged is sufficient

        balances[msg.sender][tokenAddress] = balances[msg.sender][tokenAddress].sub(amount);

        IERC20 token = IERC20(tokenAddress);
        token.transfer(msg.sender, amount);

        // epoch logic
        uint128 currentEpoch = getCurrentEpoch();

        lastWithdrawEpochId[tokenAddress] = currentEpoch;

        if (!epochIsInitialized(tokenAddress, currentEpoch)) {
            address[] memory tokens = new address[](1);
            tokens[0] = tokenAddress;
            manualEpochInit(tokens, currentEpoch);
        }

        // update the pool size of the next epoch to its current balance
        Pool storage pNextEpoch = poolSize[tokenAddress][currentEpoch + 1];
        pNextEpoch.size = token.balanceOf(address(this));
        pNextEpoch.set = true;
```

```
        Checkpoint[] storage checkpoints = balanceCheckpoints[msg.sender][tokenAddress];
        uint256 last = checkpoints.length - 1;

        // note: it's impossible to have a withdraw and no checkpoints because the balance would be 0 and revert

        // there was a deposit in an older epoch (more than 1 behind [eg: previous 0, now 5]) but no other action since then
        if (checkpoints[last].epochId < currentEpoch) {
            checkpoints.push(Checkpoint(currentEpoch, BASE_MULTIPLIER, balances[msg.sender][tokenAddress], 0));

            poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.sub(amount);
        }
        // there was a deposit in the `epochId - 1` epoch => we have a checkpoint for the current epoch
        else if (checkpoints[last].epochId == currentEpoch) {
            checkpoints[last].startBalance = balances[msg.sender][tokenAddress];
            checkpoints[last].newDeposits = 0;
            checkpoints[last].multiplier = BASE_MULTIPLIER;

            poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.sub(amount);
        }
        // there was a deposit in the current epoch
        else {
            Checkpoint storage currentEpochCheckpoint = checkpoints[last - 1];

            uint256 balanceBefore = getCheckpointEffectiveBalance(currentEpochCheckpoint);

            // in case of withdraw, we have 2 branches:
```

```
            // 1. the user withdraws less than he added in the current epoch
            // 2. the user withdraws more than he added in the current epoch (including 0)
            if (amount < currentEpochCheckpoint.newDeposits) {
                uint128 avgDepositMultiplier = uint128(
balanceBefore.sub(currentEpochCheckpoint.startBalance).mul(BASE_MULTIPLIER).div(current
EpochCheckpoint.newDeposits)
                );

                currentEpochCheckpoint.newDeposits =
currentEpochCheckpoint.newDeposits.sub(amount);

                currentEpochCheckpoint.multiplier = computeNewMultiplier(
                    currentEpochCheckpoint.startBalance,
                    BASE_MULTIPLIER,
                    currentEpochCheckpoint.newDeposits,
                    avgDepositMultiplier
                );
            } else {
                currentEpochCheckpoint.startBalance =
currentEpochCheckpoint.startBalance.sub(
                    amount.sub(currentEpochCheckpoint.newDeposits)
                );
                currentEpochCheckpoint.newDeposits = 0;
                currentEpochCheckpoint.multiplier = BASE_MULTIPLIER;
            }

            uint256 balanceAfter = getCheckpointEffectiveBalance(currentEpochCheckpoint);

            poolSize[tokenAddress][currentEpoch].size =
poolSize[tokenAddress][currentEpoch].size.sub(balanceBefore.sub(balanceAfter));

            checkpoints[last].startBalance = balances[msg.sender][tokenAddress];
```

```
        }

        emit Withdraw(msg.sender, tokenAddress, amount);
    }
```

Recommendation： nothing.

3.3. Staking contract emergencyWithdraw function【PASS】

Audit analysis: Perform a security audit on the emergencyWithdraw logic audit in the Staking contract, first determine whether emergency withdrawal is allowed, and obtain all the user's balance and perform emergency withdrawal.

```
    function emergencyWithdraw(address tokenAddress) public {
        require((getCurrentEpoch() - lastWithdrawEpochId[tokenAddress]) >= 10, "At least 10 epochs must pass without success"); //knownsec// The current epoch must be greater than 10 epochs from the last withdrawal

        uint256 totalUserBalance = balances[msg.sender][tokenAddress];
        require(totalUserBalance > 0, "Amount must be > 0"); //knownsec// All balances are greater than 0

        balances[msg.sender][tokenAddress] = 0; //knownsec// First set the amount of user pledge to 0

        IERC20 token = IERC20(tokenAddress);
        token.transfer(msg.sender, totalUserBalance); //knownsec// User withdrawal

        emit EmergencyWithdraw(msg.sender, tokenAddress, totalUserBalance);
    }
```

Recommendation： nothing.

3.4. YieldFarm contract massHarvest function【PASS】

Audit analysis: Perform a security audit on the massHarvest logic audit in the YieldFarm contract to determine whether the current epoch exceeds the maximum reward epoch of 800, and then receive rewards in batches and update them.

```
function massHarvest() external returns (uint){
        uint totalDistributedValue;
        uint epochId = _getEpochId().sub(1); // fails in epoch 0
        // force max number of epochs
        if (epochId > NR_OF_EPOCHS) {
            epochId = NR_OF_EPOCHS;
        }

        for (uint128 i = lastEpochIdHarvested[msg.sender] + 1; i <= epochId; i++) { //knownsec// Start batch harvesting from the last reward epoch until the current epoch
            // i = epochId
            // compute distributed Value and do one single transfer at the end
            totalDistributedValue += _harvest(i); //knownsec// The total amount is limited, there is no overflow, but SafeMath is recommended
        }

        emit MassHarvest(msg.sender, epochId.sub(lastEpochIdHarvested[msg.sender]), totalDistributedValue);

        if (totalDistributedValue > 0) {
            _wah.transferFrom(_vault, msg.sender, totalDistributedValue); //knownsec// Transfer rewards from vault to users
        }

        return totalDistributedValue;
    }
```

Recommendation： nothing.

3.5. YieldFarm contract harvest function【PASS】

Audit analysis: Perform security audit on the harvest logic audit in the YieldFarm contract, verify the validity of the incoming parameters, and get epoch rewards.

```
function harvest (uint128 epochId) external returns (uint){
        // checks for requested epoch
        require (_getEpochId() > epochId, "This epoch is in the future"); //knownsec// epochid must already exist
        require(epochId <= NR_OF_EPOCHS, "Maximum number of epochs is 8"); //knownsec// epochid cannot be greater than the maximum
        require (lastEpochIdHarvested[msg.sender].add(1) == epochId, "Harvest in order"); //knownsec// The harvested blocks should also be in order
        uint userReward = _harvest(epochId); //knownsec// The user's reward in the epoch
        if (userReward > 0) {
            _wah.transferFrom(_vault, msg.sender, userReward); //knownsec// Transfer rewards from vault to users
        }
        emit Harvest(msg.sender, epochId, userReward);
        return userReward;
    }
```

Recommendation： nothing.

4. Basic code vulnerability detection

4.1. Compiler version security【PASS】

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

Audit result: After testing, the smart contract code has formulated the compiler version 0.6.0 within the major version, and there is no such security problem.

Recommendation： nothing.

4.2. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.3. Use of safe arithmetic library【PASS】

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

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

Recommendation： nothing.

4.4. Not recommended encoding【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.5. Reasonable use of require/assert【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.6. Fallback function safety【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.7. tx.origin 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.8. 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.9. Gas consumption detection【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.10. call injection 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 result: After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation： nothing.

4.11. 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.12. Vulnerability of additional token issuance【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 result: After testing, the smart contract code does not have the function of issuing additional tokens, and the upper limit is set, so it is passed.

Recommendation： nothing.

4.13. Access control defect detection【PASS】

Different functions in the contract should set 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 problems caused by unauthorized access.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.14. 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.15. 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.16. Incorrect use of random numbers【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.17. Unsafe interface usage【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.18. Variable coverage【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.19. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the 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 result: After testing, the smart contract code does not use structure, there is no such problem.

Recommendation： nothing.

4.20. Return value call verification【PASS】

This problem 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 ETH 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 gas_value parameters), 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 ETH sending failure.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.21. Transaction order dependency【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 content of other people's pending transactions. 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 result: After testing, the _approve function in the contract has a transaction sequence dependency attack risk, but the vulnerability is extremely difficult to exploit, so it is rated as passed. The code is as follows:

```
function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");
```

```
        _allowances[owner][spender] = amount;//knownnsec// Transaction order depends on
risk        emit Approval(owner, spender, amount);
    }
```

The possible security risks are described as follows:

1. By calling the approve function, user A allows user B to transfer money on his behalf to N (N>0);

2. After a period of time, user A decides to change N to M (M>0), so call the approve function again;

3. User B quickly calls the transferFrom function to transfer N number of tokens before the second call is processed by the miner;

4. After user A's second call to approve is successful, user B can obtain M's transfer quota again, that is, user B obtains N+M's transfer quota through the transaction sequence attack.

Recommendation：

1. Front-end restriction, when user A changes the quota from N to M, he can first change from N to 0, and then from 0 to M.

2. Add the following code at the beginning of the approve function:

require((_value == 0) || (allowed[msg.sender][_spender] == 0));

4.22. Timestamp dependency 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 result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.23. Denial of service attack【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 the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc. Wait.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.24. Fake recharge vulnerability【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 if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.25. Reentry attack detection【PASS】

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send ETH. When the **call.value()** function to send ETH 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.

4.26. 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.

4.27. 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.

5. Appendix A：Contract code

Source code：

Migrations.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.9.0;

contract Migrations {
  address public owner = msg.sender;
  uint public last_completed_migration;

  modifier restricted() {
    require(
      msg.sender == owner,
      "This function is restricted to the contract's owner"
    );
    _;
  }

  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }
}
```

Staking.sol

```
/**
 *Submitted for verification at Etherscan.io on 2021-02-21
*/

// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.6.0;

contract Initializable {

    /**
     * @dev Indicates that the contract has been initialized.
     */
    bool private _initialized;

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

    /**
     * @dev Modifier to protect an initializer function from being invoked twice.
     */
    modifier initializer() {
        require(_initializing || !_initialized, "Initializable: contract is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }
}
/**
 * @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 `recipient`.
```

```
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, 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 `sender` to `recipient` 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 sender, address recipient, 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);
}

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when 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.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
```

```
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
```

```
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
 * available, which can be applied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
 * points to them.
 *
 * TIP: If you would like to learn more about reentrancy and alternative ways
 * to protect against it, check out our blog post
 * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
 */
contract ReentrancyGuardUpgradeable is Initializable {
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.

    // The values being non-zero value makes deployment a bit more expensive,
    // but in exchange the refund on every call to nonReentrant will be lower in
    // amount. Since refunds are capped to a percentage of the total
    // transaction's gas, it is best to keep them low in cases like this one, to
    // increase the likelihood of the full refund coming into effect.
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    function __ReentrancyGuard_init() internal initializer {
        __ReentrancyGuard_init_unchained();
    }

    function __ReentrancyGuard_init_unchained() internal initializer {
        _status = _NOT_ENTERED;
    }

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     * Calling a `nonReentrant` function from another `nonReentrant`
     * function is not supported. It is possible to prevent this from happening
     * by making the `nonReentrant` function external, and make it call a
     * `private` function that does the actual work.
```

```
     */
    modifier nonReentrant() {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;

        _;

        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }
    uint256[49] private __gap;
}

contract Staking is ReentrancyGuardUpgradeable {
    using SafeMath for uint256;

    uint128 constant private BASE_MULTIPLIER = uint128(1 * 10 ** 18);

    // timestamp for the epoch 1
    // everything before that is considered epoch 0 which won't have a reward but allows for the initial stake
    uint256 public epoch1Start;

    // duration of each epoch
    uint256 public epochDuration;

    // holds the current balance of the user for each token
    mapping(address => mapping(address => uint256)) private balances;

    struct Pool {
        uint256 size;
        bool set;
    }

    // for each token, we store the total pool size
    mapping(address => mapping(uint256 => Pool)) private poolSize;

    // a checkpoint of the valid balance of a user for an epoch
    struct Checkpoint {
        uint128 epochId;
        uint128 multiplier;
        uint256 startBalance;
        uint256 newDeposits;
    }

    // balanceCheckpoints[user][token][]
    mapping(address => mapping(address => Checkpoint[])) private balanceCheckpoints;

    mapping(address => uint128) private lastWithdrawEpochId;

    event Deposit(address indexed user, address indexed tokenAddress, uint256 amount);
    event Withdraw(address indexed user, address indexed tokenAddress, uint256 amount);
    event ManualEpochInit(address indexed caller, uint128 indexed epochId, address[] tokens);
    event EmergencyWithdraw(address indexed user, address indexed tokenAddress, uint256 amount);

    function initialize (uint256 _epoch1Start, uint256 _epochDuration) public initializer {
        epoch1Start = _epoch1Start;
        epochDuration = _epochDuration;
    }

    /*
     * Stores `amount` of `tokenAddress` tokens for the `user` into the vault
     */
    function deposit(address tokenAddress, uint256 amount) public nonReentrant {
        require(amount > 0, "Staking: Amount must be > 0"); //knownsec//质押数量要大于 0

        IERC20 token = IERC20(tokenAddress);
        uint256 allowance = token.allowance(msg.sender, address(this));
        require(allowance >= amount, "Staking: Token allowance too small"); //knownsec//授权的额度要大于
质押的数量

        balances[msg.sender][tokenAddress] = balances[msg.sender][tokenAddress].add(amount); //knownsec//
更新用户该 token 的质押数量

        token.transferFrom(msg.sender, address(this), amount);

        // epoch logic
        uint128 currentEpoch = getCurrentEpoch(); //knownsec//获取当前的 epoch
        uint128 currentMultiplier = currentEpochMultiplier(); //knownsec//获取当前的 Multiplier

        if (!epochIsInitialized(tokenAddress, currentEpoch)) { //knownsec//判断当前 epoch 中 token 是否已经初
始化
            address[] memory tokens = new address[](1);
            tokens[0] = tokenAddress;
```

```
            manualEpochInit(tokens, currentEpoch); //knownsec//未初始化则手动初始化
        }

        // update the next epoch pool size
        Pool storage pNextEpoch = poolSize[tokenAddress][currentEpoch + 1];
        pNextEpoch.size = token.balanceOf(address(this)); //knownsec//下一个 epoch pool 的大小是目前合约中的 token 余额
        pNextEpoch.set = true; //knownsec//将下一个 epoch pool 设置为已更改

        Checkpoint[] storage checkpoints = balanceCheckpoints[msg.sender][tokenAddress]; //knownsec//记录用户 checkpoint

        uint256 balanceBefore = getEpochUserBalance(msg.sender, tokenAddress, currentEpoch); //knownsec//获得当前 epoch 中用户的 token 余额

        // if there's no checkpoint yet, it means the user didn't have any activity
        // we want to store checkpoints both for the current epoch and next epoch because
        // if a user does a withdraw, the current epoch can also be modified and
        // we don't want to insert another checkpoint in the middle of the array as that could be expensive
        if (checkpoints.length == 0) {
            checkpoints.push(Checkpoint(currentEpoch, currentMultiplier, 0, amount)); //knownsec//checkpoints.length 等于 0 说明用户未质押或者提现过，设置当前 epoch 用户起始余额为 0，质押的数量为 amount

            // next epoch => multiplier is 1, epoch deposits is 0
            checkpoints.push(Checkpoint(currentEpoch + 1, BASE_MULTIPLIER, amount, 0)); //knownsec//设置下一个 epoch 的初始余额为 amount，质押的数量为 0
        } else {
            uint256 last = checkpoints.length - 1;

            // the last action happened in an older epoch (e.g. a deposit in epoch 3, current epoch is >=5)
            if (checkpoints[last].epochId < currentEpoch) { //knownsec//checkpoint 所在的 epochid 要小于当前 epochid
                uint128 multiplier = computeNewMultiplier( //knownsec//计算新的 multiplier
                    getCheckpointBalance(checkpoints[last]),
                    BASE_MULTIPLIER,
                    amount,
                    currentMultiplier
                );
                checkpoints.push(Checkpoint(currentEpoch, multiplier, getCheckpointBalance(checkpoints[last]), amount)); //knownsec//更新当前 epoch 中的 checkpoint，起始余额为所有 checkpoint 中的余额之和
                checkpoints.push(Checkpoint(currentEpoch + 1, BASE_MULTIPLIER, balances[msg.sender][tokenAddress], 0)); //knownsec//设置下一个 epoch 中的 checkpoint
            }
            // the last action happened in the previous epoch
            else if (checkpoints[last].epochId == currentEpoch) {
                checkpoints[last].multiplier = computeNewMultiplier(
                    getCheckpointBalance(checkpoints[last]),
                    checkpoints[last].multiplier,
                    amount,
                    currentMultiplier
                );
                checkpoints[last].newDeposits = checkpoints[last].newDeposits.add(amount); //knownsec//继续质押

                checkpoints.push(Checkpoint(currentEpoch + 1, BASE_MULTIPLIER, balances[msg.sender][tokenAddress], 0));
            }
            // the last action happened in the current epoch
            else {
                if (last >= 1 && checkpoints[last - 1].epochId == currentEpoch) {
                    checkpoints[last - 1].multiplier = computeNewMultiplier(
                        getCheckpointBalance(checkpoints[last - 1]),
                        checkpoints[last - 1].multiplier,
                        amount,
                        currentMultiplier
                    );
                    checkpoints[last - 1].newDeposits = checkpoints[last - 1].newDeposits.add(amount);
                }

                checkpoints[last].startBalance = balances[msg.sender][tokenAddress]; //knownsec//起始余额设置为目前用户已经质押的数量
            }
        }

        uint256 balanceAfter = getEpochUserBalance(msg.sender, tokenAddress, currentEpoch); //knownsec//获取用户质押后当前 epoch 中的的 token 余额

        poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.add(balanceAfter.sub(balanceBefore)); //knownsec//当前 eopch 中 pool 的大小加上质押前后的差值

        emit Deposit(msg.sender, tokenAddress, amount);
    }

    /*
```

```
     * Removes the deposit of the user and sends the amount of `tokenAddress` back to the `user`
     */
    function withdraw(address tokenAddress, uint256 amount) public nonReentrant { //knownsec//提现 amount 数量的 token 到用户账户
        require(balances[msg.sender][tokenAddress] >= amount, "Staking: balance too small"); //knownsec//用户已质押的 balance 数量足够

        balances[msg.sender][tokenAddress] = balances[msg.sender][tokenAddress].sub(amount);

        IERC20 token = IERC20(tokenAddress);
        token.transfer(msg.sender, amount);

        // epoch logic
        uint128 currentEpoch = getCurrentEpoch();

        lastWithdrawEpochId[tokenAddress] = currentEpoch;

        if (!epochIsInitialized(tokenAddress, currentEpoch)) {
            address[] memory tokens = new address[](1);
            tokens[0] = tokenAddress;
            manualEpochInit(tokens, currentEpoch);
        }

        // update the pool size of the next epoch to its current balance
        Pool storage pNextEpoch = poolSize[tokenAddress][currentEpoch + 1];
        pNextEpoch.size = token.balanceOf(address(this));
        pNextEpoch.set = true;

        Checkpoint[] storage checkpoints = balanceCheckpoints[msg.sender][tokenAddress];
        uint256 last = checkpoints.length - 1;

        // note: it's impossible to have a withdraw and no checkpoints because the balance would be 0 and revert

        // there was a deposit in an older epoch (more than 1 behind [eg: previous 0, now 5]) but no other action since then
        if (checkpoints[last].epochId < currentEpoch) {
            checkpoints.push(Checkpoint(currentEpoch, BASE_MULTIPLIER, balances[msg.sender][tokenAddress], 0));

            poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.sub(amount);
        }
        // there was a deposit in the `epochId - 1` epoch => we have a checkpoint for the current epoch
        else if (checkpoints[last].epochId == currentEpoch) {
            checkpoints[last].startBalance = balances[msg.sender][tokenAddress];
            checkpoints[last].newDeposits = 0;
            checkpoints[last].multiplier = BASE_MULTIPLIER;

            poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.sub(amount);
        }
        // there was a deposit in the current epoch
        else {
            Checkpoint storage currentEpochCheckpoint = checkpoints[last - 1];

            uint256 balanceBefore = getCheckpointEffectiveBalance(currentEpochCheckpoint);

            // in case of withdraw, we have 2 branches:
            // 1. the user withdraws less than he added in the current epoch
            // 2. the user withdraws more than he added in the current epoch (including 0)
            if (amount < currentEpochCheckpoint.newDeposits) {
                uint128 avgDepositMultiplier = uint128(
                    balanceBefore.sub(currentEpochCheckpoint.startBalance).mul(BASE_MULTIPLIER).div(currentEpochCheckpoint.newDeposits)
                );

                currentEpochCheckpoint.newDeposits = currentEpochCheckpoint.newDeposits.sub(amount);

                currentEpochCheckpoint.multiplier = computeNewMultiplier(
                    currentEpochCheckpoint.startBalance,
                    BASE_MULTIPLIER,
                    currentEpochCheckpoint.newDeposits,
                    avgDepositMultiplier
                );
            } else {
                currentEpochCheckpoint.startBalance = currentEpochCheckpoint.startBalance.sub(
                    amount.sub(currentEpochCheckpoint.newDeposits)
                );
                currentEpochCheckpoint.newDeposits = 0;
                currentEpochCheckpoint.multiplier = BASE_MULTIPLIER;
            }

            uint256 balanceAfter = getCheckpointEffectiveBalance(currentEpochCheckpoint);

            poolSize[tokenAddress][currentEpoch].size = poolSize[tokenAddress][currentEpoch].size.sub(balanceBefore.sub(balanceAfter));
```

```
                checkpoints[last].startBalance = balances[msg.sender][tokenAddress];
            }

            emit Withdraw(msg.sender, tokenAddress, amount);
        }

        /*
         * manualEpochInit can be used by anyone to initialize an epoch based on the previous one
         * This is only applicable if there was no action (deposit/withdraw) in the current epoch.
         * Any deposit and withdraw will automatically initialize the current and next epoch.
         */
        function manualEpochInit(address[] memory tokens, uint128 epochId) public {
            require(epochId <= getCurrentEpoch(), "can't init a future epoch"); //knownsec//要初始化的 epoch 必须小于等于最新的 epoch

            for (uint i = 0; i < tokens.length; i++) {
                Pool storage p = poolSize[tokens[i]][epochId];

                if (epochId == 0) {
                    p.size = uint256(0);
                    p.set = true;
                } else {
                    require(!epochIsInitialized(tokens[i], epochId), "Staking: epoch already initialized");
                    require(epochIsInitialized(tokens[i], epochId - 1), "Staking: previous epoch not initialized"); //knownsec//目标 epoch 未初始化，且上一个 epoch 已经初始化

                    p.size = poolSize[tokens[i]][epochId - 1].size;
                    p.set = true;
                }
            }

            emit ManualEpochInit(msg.sender, epochId, tokens);
        }

        function emergencyWithdraw(address tokenAddress) public {
            require((getCurrentEpoch() - lastWithdrawEpochId[tokenAddress]) >= 10, "At least 10 epochs must pass without success"); //knownsec//目前的 epoch 必须大于最后一次提现的 epoch10 个

            uint256 totalUserBalance = balances[msg.sender][tokenAddress];
            require(totalUserBalance > 0, "Amount must be > 0"); //knownsec//所有余额大于 0

            balances[msg.sender][tokenAddress] = 0; //knownsec//先将用户质押的数量设为 0

            IERC20 token = IERC20(tokenAddress);
            token.transfer(msg.sender, totalUserBalance); //knownsec//用户提现

            emit EmergencyWithdraw(msg.sender, tokenAddress, totalUserBalance);
        }

        /*
         * Returns the valid balance of a user that was taken into consideration in the total pool size for the epoch
         * A deposit will only change the next epoch balance.
         * A withdraw will decrease the current epoch (and subsequent) balance.
         */
        function getEpochUserBalance(address user, address token, uint128 epochId) public view returns (uint256) {
            Checkpoint[] storage checkpoints = balanceCheckpoints[user][token];

            // if there are no checkpoints, it means the user never deposited any tokens, so the balance is 0
            if (checkpoints.length == 0 || epochId < checkpoints[0].epochId) { //knownsec//checkpoint 有效
                return 0;
            }

            uint min = 0;
            uint max = checkpoints.length - 1;

            // shortcut for blocks newer than the latest checkpoint == current balance
            if (epochId >= checkpoints[max].epochId) {
                return getCheckpointEffectiveBalance(checkpoints[max]);
            }

            // binary search of the value in the array
            while (max > min) {
                uint mid = (max + min + 1) / 2;
                if (checkpoints[mid].epochId <= epochId) {
                    min = mid;
                } else {
                    max = mid - 1;
                }
            }

            return getCheckpointEffectiveBalance(checkpoints[min]);
        }

        /*
         * Returns the amount of `token` that the `user` has currently staked
         */
```

```
    function balanceOf(address user, address token) public view returns (uint256) {
        return balances[user][token];
    }

    /*
     * Returns the id of the current epoch derived from block.timestamp
     */
    function getCurrentEpoch() public view returns (uint128) {
        if (block.timestamp < epoch1Start) {
            return 0;
        }

        return uint128((block.timestamp - epoch1Start) / epochDuration + 1);
    }

    /*
     * Returns the total amount of `tokenAddress` that was locked from beginning to end of epoch identified by `epochId`
     */
    function getEpochPoolSize(address tokenAddress, uint128 epochId) public view returns (uint256) {
        // Premises:
        // 1. it's impossible to have gaps of uninitialized epochs
        // - any deposit or withdraw initialize the current epoch which requires the previous one to be initialized
        if (epochIsInitialized(tokenAddress, epochId)) {
            return poolSize[tokenAddress][epochId].size;
        }

        // epochId not initialized and epoch 0 not initialized => there was never any action on this pool
        if (!epochIsInitialized(tokenAddress, 0)) {
            return 0;
        }

        // epoch 0 is initialized => there was an action at some point but none that initialized the epochId
        // which means the current pool size is equal to the current balance of token held by the staking contract
        IERC20 token = IERC20(tokenAddress);
        return token.balanceOf(address(this));
    }

    /*
     * Returns the percentage of time left in the current epoch
     */
    function currentEpochMultiplier() public view returns (uint128) {
        uint128 currentEpoch = getCurrentEpoch();
        uint256 currentEpochEnd = epoch1Start + currentEpoch * epochDuration; //knownsec// 建议使用 SafeMath 进行运算
        uint256 timeLeft = currentEpochEnd - block.timestamp;
        uint128 multiplier = uint128(timeLeft * BASE_MULTIPLIER / epochDuration);

        return multiplier;
    }

    function computeNewMultiplier(uint256 prevBalance, uint128 prevMultiplier, uint256 amount, uint128 currentMultiplier) public pure returns (uint128) {
        uint256 prevAmount = prevBalance.mul(prevMultiplier).div(BASE_MULTIPLIER);
        uint256 addAmount = amount.mul(currentMultiplier).div(BASE_MULTIPLIER);
        uint128 newMultiplier = uint128(prevAmount.add(addAmount).mul(BASE_MULTIPLIER).div(prevBalance.add(amount)));

        return newMultiplier;
    }

    /*
     * Checks if an epoch is initialized, meaning we have a pool size set for it
     */
    function epochIsInitialized(address token, uint128 epochId) public view returns (bool) {
        return poolSize[token][epochId].set;
    }

    function getCheckpointBalance(Checkpoint memory c) internal pure returns (uint256) {
        return c.startBalance.add(c.newDeposits);
    }

    function getCheckpointEffectiveBalance(Checkpoint memory c) internal pure returns (uint256) {
        return getCheckpointBalance(c).mul(c.multiplier).div(BASE_MULTIPLIER);
    }
}
```

WAHToken.sol

```
/**
 *Submitted for verification at Etherscan.io on 2021-01-24
*/

// File: @openzeppelin/contracts/GSN/Context.sol


pragma solidity 0.6.0;
```

```
/*
 * @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 GSN 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 payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
}

/**
 * @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 `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, 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 `sender` to `recipient` 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 sender, address recipient, 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);
}

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when 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.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
```

```
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

/**
 * @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 guidelines: functions revert instead
 * of 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 {
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

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

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
     * a default value of 18.
     *
     * To select a different value for {decimals}, use {_setupDecimals}.
     *
     * All three of these values are immutable: they can only be set once during
     * construction.
     */
    constructor (string memory name_, string memory symbol_) public {
        _name = name_;
        _symbol = symbol_;
        _decimals = 18;
    }

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

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view 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 {_setupDecimals} is
     * called.
     *
     * 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 returns (uint8) {
        return _decimals;
    }

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

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view override returns (uint256) {
```

```
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `recipient` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        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}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        _approve(_msgSender(), spender, amount);
        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}.
     *
     * Requirements:
     *
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     * - the caller must have allowance for ``sender``'s tokens of at least
     * `amount`.
     */
    function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool)
{
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
amount exceeds allowance"));
        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) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(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) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));
        return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is 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:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, 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:
     *
     * - `to` 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 = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(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);

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(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");
        require(spender != address(0), "ERC20: approve to the zero address"); //knownsec//此处存在事务顺序
```

```
依赖风险,建议添加下面的语句进行判断
            //require((amout == 0) || (_allowed[owner][spender] == 0));
            _allowances[owner][spender] = amount;
            emit Approval(owner, spender, amount);
        }

        /**
         * @dev Sets {decimals} to a value other than the default one of 18.
         *
         * WARNING: This function should only be called from the constructor. Most
         * applications that interact with token contracts will not expect
         * {decimals} to ever change, and may work incorrectly if it does.
         */
        function _setupDecimals(uint8 decimals_) internal {
            _decimals = decimals_;
        }

        /**
         * @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 to 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 { }
}

contract WAHToken is ERC20 {
        constructor() public ERC20("FM Gallery", "WAH") {
            _mint(msg.sender, 100000000 * 10**18);
        }
}
```

YieldFarm.sol

```
/**
 *Submitted for verification at Etherscan.io on 2021-02-21
*/

// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.6.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when 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.
 */
library SafeMath {
        /**
         * @dev Returns the addition of two unsigned integers, reverting on
         * overflow.
         *
         * Counterpart to Solidity's `+` operator.
         *
         * Requirements:
         * - Addition cannot overflow.
         */
        function add(uint256 a, uint256 b) internal pure returns (uint256) {
            uint256 c = a + b;
            require(c >= a, "SafeMath: addition overflow");

            return c;
        }

        /**
         * @dev Returns the subtraction of two unsigned integers, reverting on
         * overflow (when the result is negative).
         *
         * Counterpart to Solidity's `-` operator.
         *
         * Requirements:
         * - Subtraction cannot overflow.
```

```
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
```

```
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}


/**
 * @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 `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, 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 `sender` to `recipient` 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 sender, address recipient, 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);
}

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
}
/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */

abstract contract Initializable {

    /**
     * @dev Indicates that the contract has been initialized.
     */
    bool private _initialized;

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

    /**
     * @dev Modifier to protect an initializer function from being invoked twice.
     */
    modifier initializer() {
        require(_initializing || _isConstructor() || !_initialized, "Initializable: contract is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }

    /// @dev Returns true if and only if the function is running in the constructor
    function _isConstructor() private view returns (bool) {
        // extcodesize checks the size of the code stored in an address, and
        // address returns the current address. Since the code is still not
        // deployed when running a constructor, any checks on its code size will
        // yield zero, making it an effective way to detect if a contract is
        // under construction or not.
        address self = address(this);
        uint256 cs;
        // solhint-disable-next-line no-inline-assembly
        assembly { cs := extcodesize(self) }
        return cs == 0;
    }
}


abstract contract ContextUpgradeable is Initializable {
    function __Context_init() internal initializer {
        __Context_init_unchained();
    }
```

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

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
    uint256[50] private __gap;
}

abstract contract OwnableUpgradeable is Initializable, ContextUpgradeable {
    address private _owner;

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

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

    function __Ownable_init_unchained() internal initializer {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

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

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
    uint256[49] private __gap;
}

interface IStaking {

    function getEpochId(uint timestamp) external view returns (uint); // get epoch id
    function getEpochUserBalance(address user, address token, uint128 epoch) external view returns(uint);
    function getEpochPoolSize(address token, uint128 epoch) external view returns (uint);
    function epoch1Start() external view returns (uint);
    function epochDuration() external view returns (uint);
}

contract YieldFarm is Initializable {

    // lib
```

```
    using SafeMath for uint;
    using SafeMath for uint128;

    // constants
    uint public constant TOTAL_DISTRIBUTED_AMOUNT = 2400000;
    uint public constant NR_OF_EPOCHS = 800;

     // state variables

    // addreses
    address private _usdc;
    address private _susd;
    address private _dai;
    address private _usdt;
    address private _vault;
    // contracts
    IERC20 private _wah;
    IStaking private _staking;


    // fixed size array holdings total number of epochs + 1 (epoch 0 doesn't count)
    uint[] private epochs;
    // pre-computed variable for optimization. total amount of WAH tokens to be distributed on each epoch
    uint private _totalAmountPerEpoch;

    // id of last init epoch, for optimization purposes moved from struct to a single id.
    uint128 public lastInitializedEpoch;

    // state of user harvest epoch
    mapping(address => uint128) private lastEpochIdHarvested;
    uint public epochDuration; // init from staking contract
    uint public epochStart; // init from staking contract

    // events
    event MassHarvest(address indexed user, uint256 epochsHarvested, uint256 totalValue);
    event Harvest(address indexed user, uint128 indexed epochId, uint256 amount);

    // constructor
    function initialize(address wahTokenAddress, address usdc, address susd, address dai, address usdt, address
stakeContract, address vault) public initializer {
        epochs = new uint[](NR_OF_EPOCHS + 1);
        _wah = IERC20(wahTokenAddress);
        _usdc = usdc;
        _susd = susd;
        _dai = dai;
        _usdt = usdt;
        _staking = IStaking(stakeContract);
        _vault = vault;
        epochStart = _staking.epoch1Start();
        epochDuration = _staking.epochDuration();
        _totalAmountPerEpoch = TOTAL_DISTRIBUTED_AMOUNT.mul(10**18).div(NR_OF_EPOCHS);


    }

    // public methods
    // public method to harvest all the unharvested epochs until current epoch - 1
    function massHarvest() external returns (uint){
        uint totalDistributedValue;
        uint epochId = _getEpochId().sub(1); // fails in epoch 0
        // force max number of epochs
        if (epochId > NR_OF_EPOCHS) {
            epochId = NR_OF_EPOCHS;
        }

        for (uint128 i = lastEpochIdHarvested[msg.sender] + 1; i <= epochId; i++) { //knownsec//从最后一个
奖励 epoch 开始批量收获直到目前的 epoch
            // i = epochId
            // compute distributed Value and do one single transfer at the end
            totalDistributedValue += _harvest(i); //knownsec//限制了总量，不存在上溢，但是建议使用
SafeMath
        }

        emit MassHarvest(msg.sender, epochId.sub(lastEpochIdHarvested[msg.sender]), totalDistributedValue);

        if (totalDistributedValue > 0) {
            _wah.transferFrom(_vault, msg.sender, totalDistributedValue); //knownsec//从 vault 中将奖励转移
给用户
        }

        return totalDistributedValue;
    }
    function harvest (uint128 epochId) external returns (uint){
        // checks for requested epoch
        require (_getEpochId() > epochId, "This epoch is in the future"); //knownsec//epochid 必须已经存在
        require(epochId <= NR_OF_EPOCHS, "Maximum number of epochs is 8"); //knownsec//epochid 不能大
于最大值
```

```
        require (lastEpochIdHarvested[msg.sender].add(1) == epochId, "Harvest in order"); //knownsec//收获的区块也要按照顺序
        uint userReward = _harvest(epochId); //knownsec//用户在该 epoch 中的奖励
        if (userReward > 0) {
            _wah.transferFrom(_vault, msg.sender, userReward); //knownsec//从 vault 中将奖励转移给用户
        }
        emit Harvest(msg.sender, epochId, userReward);
        return userReward;
    }

    // views
    // calls to the staking smart contract to retrieve the epoch total pool size
    function getPoolSize(uint128 epochId) external view returns (uint) {
        return _getPoolSize(epochId);
    }

    function getCurrentEpoch() external view returns (uint) {
        return _getEpochId();
    }

    // calls to the staking smart contract to retrieve user balance for an epoch
    function getEpochStake(address userAddress, uint128 epochId) external view returns (uint) {
        return _getUserBalancePerEpoch(userAddress, epochId);
    }

    function userLastEpochIdHarvested() external view returns (uint){
        return lastEpochIdHarvested[msg.sender];
    }

    // internal methods

    function _initEpoch(uint128 epochId) internal {
        require(lastInitializedEpoch.add(1) == epochId, "Epoch can be init only in order"); //knownsec//epoch 必须按顺序初始化
        lastInitializedEpoch = epochId; //knownsec//新增的 epoch 排在最后一个
        // call the staking smart contract to init the epoch
        epochs[epochId] = _getPoolSize(epochId);
    }

    function _harvest (uint128 epochId) internal returns (uint) {
        // try to initialize an epoch. if it can't it fails
        // if it fails either user either a FM Gallery account will init not init epochs
        if (lastInitializedEpoch < epochId) {
            _initEpoch(epochId);
        }
        // Set user last harvested epoch
        lastEpochIdHarvested[msg.sender] = epochId;
        // compute and return user total reward. For optimization reasons the transfer have been moved to an upper layer (i.e. massHarvest needs to do a single transfer)

        // exit if there is no stake on the epoch
        if (epochs[epochId] == 0) {
            return 0;
        }

        return _totalAmountPerEpoch
        .mul(_getUserBalancePerEpoch(msg.sender, epochId))
        .div(epochs[epochId]);
    }

    function _getPoolSize(uint128 epochId) internal view returns (uint) {
        // retrieve stable coins total staked in epoch
        uint valueUsdc = _staking.getEpochPoolSize(_usdc, epochId).mul(10 ** 12); // for usdc which has 6 decimals add a 10**12 to get to a common ground
        uint valueSusd = _staking.getEpochPoolSize(_susd, epochId);
        uint valueDai = _staking.getEpochPoolSize(_dai, epochId);
        uint valueUsdt = _staking.getEpochPoolSize(_usdt, epochId).mul(10 ** 12); // for usdt which has 6 decimals add a 10**12 to get to a common ground
        return valueUsdc.add(valueSusd).add(valueDai).add(valueUsdt);
    }



    function _getUserBalancePerEpoch(address userAddress, uint128 epochId) internal view returns (uint){
        // retrieve stable coins total staked per user in epoch
        uint valueUsdc = _staking.getEpochUserBalance(userAddress, _usdc, epochId).mul(10 ** 12); // for usdc which has 6 decimals add a 10**12 to get to a common ground
        uint valueSusd = _staking.getEpochUserBalance(userAddress, _susd, epochId);
        uint valueDai = _staking.getEpochUserBalance(userAddress, _dai, epochId);
        uint valueUsdt = _staking.getEpochUserBalance(userAddress, _usdt, epochId).mul(10 ** 12); // for usdt which has 6 decimals add a 10**12 to get to a common ground
        return valueUsdc.add(valueSusd).add(valueDai).add(valueUsdt);
    }

    // compute epoch id from blocktimestamp and epochstart date
    function _getEpochId() internal view returns (uint128 epochId) {
        if (block.timestamp < epochStart) {
```

```
            return 0;
        }
        epochId = uint128(block.timestamp.sub(epochStart).div(epochDuration).add(1));
    }
}
```

YieldFarmLP.sol

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.6.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when 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.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }
```

```
        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

/**
 * @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 `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, 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 `sender` to `recipient` 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 sender, address recipient, 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);
}

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
}
/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */

abstract contract Initializable {
```

```
    /**
     * @dev Indicates that the contract has been initialized.
     */
    bool private _initialized;

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

    /**
     * @dev Modifier to protect an initializer function from being invoked twice.
     */
    modifier initializer() {
        require(_initializing || _isConstructor() || !_initialized, "Initializable: contract is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }

    /// @dev Returns true if and only if the function is running in the constructor
    function _isConstructor() private view returns (bool) {
        // extcodesize checks the size of the code stored in an address, and
        // address returns the current address. Since the code is still not
        // deployed when running a constructor, any checks on its code size will
        // yield zero, making it an effective way to detect if a contract is
        // under construction or not.
        address self = address(this);
        uint256 cs;
        // solhint-disable-next-line no-inline-assembly
        assembly { cs := extcodesize(self) }
        return cs == 0;
    }
}

abstract contract ContextUpgradeable is Initializable {
    function __Context_init() internal initializer {
        __Context_init_unchained();
    }

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

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
    uint256[50] private __gap;
}

abstract contract OwnableUpgradeable is Initializable, ContextUpgradeable {
    address private _owner;

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

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

    function __Ownable_init_unchained() internal initializer {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
```

```
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

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

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
    uint256[49] private __gap;
}


interface IStaking {

    function getEpochId(uint timestamp) external view returns (uint); // get epoch id
    function getEpochUserBalance(address user, address token, uint128 epoch) external view returns(uint);
    function getEpochPoolSize(address token, uint128 epoch) external view returns (uint);
    function epoch1Start() external view returns (uint);
    function epochDuration() external view returns (uint);
}

contract YieldFarmLP is Initializable {

    // lib
    using SafeMath for uint;
    using SafeMath for uint128;

    // constants
    uint public constant TOTAL_DISTRIBUTED_AMOUNT = 2000000;
    uint public constant NR_OF_EPOCHS = 200;

    // state variables

    // addreses
    address private _uniLP;
    address private _vault;
    // contracts
    IERC20 private _wah;
    IStaking private _staking;


    uint[] private epochs;
    uint private _totalAmountPerEpoch;
    uint128 public lastInitializedEpoch;
    mapping(address => uint128) private lastEpochIdHarvested;
    uint public epochDuration; // init from staking contract
    uint public epochStart; // init from staking contract

    // events
    event MassHarvest(address indexed user, uint256 epochsHarvested, uint256 totalValue);
    event Harvest(address indexed user, uint128 indexed epochId, uint256 amount);

    // constructor
    function initialize(address wahTokenAddress, address uniLP, address stakeContract, address vault) public
initializer {
        epochs = new uint[](NR_OF_EPOCHS + 1);
        _wah = IERC20(wahTokenAddress);
        _uniLP = uniLP;
        _staking = IStaking(stakeContract);
        _vault = vault;
        epochDuration = _staking.epochDuration();
```

```
        epochStart = _staking.epoch1Start() + epochDuration;
        _totalAmountPerEpoch = TOTAL_DISTRIBUTED_AMOUNT.mul(10**18).div(NR_OF_EPOCHS);
    }

    // public methods
    // public method to harvest all the unharvested epochs until current epoch - 1
    function massHarvest() external returns (uint){
        uint totalDistributedValue;
        uint epochId = _getEpochId().sub(1); // fails in epoch 0
        // force max number of epochs
        if (epochId > NR_OF_EPOCHS) {
            epochId = NR_OF_EPOCHS;
        }

        for (uint128 i = lastEpochIdHarvested[msg.sender] + 1; i <= epochId; i++) {
            // i = epochId
            // compute distributed Value and do one single transfer at the end
            totalDistributedValue += _harvest(i);
        }

        emit MassHarvest(msg.sender, epochId - lastEpochIdHarvested[msg.sender], totalDistributedValue);

        if (totalDistributedValue > 0) {
            _wah.transferFrom(_vault, msg.sender, totalDistributedValue);
        }

        return totalDistributedValue;
    }
    function harvest (uint128 epochId) external returns (uint){
        // checks for requested epoch
        require (_getEpochId() > epochId, "This epoch is in the future");
        require(epochId <= NR_OF_EPOCHS, "Maximum number of epochs is 100");
        require (lastEpochIdHarvested[msg.sender].add(1) == epochId, "Harvest in order");
        uint userReward = _harvest(epochId);
        if (userReward > 0) {
            _wah.transferFrom(_vault, msg.sender, userReward);
        }
        emit Harvest(msg.sender, epochId, userReward);
        return userReward;
    }

    // views
    // calls to the staking smart contract to retrieve the epoch total pool size
    function getPoolSize(uint128 epochId) external view returns (uint) {
        return _getPoolSize(epochId);
    }

    function getCurrentEpoch() external view returns (uint) {
        return _getEpochId();
    }

    // calls to the staking smart contract to retrieve user balance for an epoch
    function getEpochStake(address userAddress, uint128 epochId) external view returns (uint) {
        return _getUserBalancePerEpoch(userAddress, epochId);
    }

    function userLastEpochIdHarvested() external view returns (uint){
        return lastEpochIdHarvested[msg.sender];
    }

    // internal methods

    function _initEpoch(uint128 epochId) internal {
        require(lastInitializedEpoch.add(1) == epochId, "Epoch can be init only in order");
        lastInitializedEpoch = epochId;
        // call the staking smart contract to init the epoch
        epochs[epochId] = _getPoolSize(epochId);
    }

    function _harvest (uint128 epochId) internal returns (uint) {
        // try to initialize an epoch. if it can't it fails
        // if it fails either user either a FM Gallery account will init not init epochs
        if (lastInitializedEpoch < epochId) {
            _initEpoch(epochId);
        }
        // Set user state for last harvested
        lastEpochIdHarvested[msg.sender] = epochId;
        // compute and return user total reward. For optimization reasons the transfer have been moved to an
upper layer (i.e. massHarvest needs to do a single transfer)

        // exit if there is no stake on the epoch
        if (epochs[epochId] == 0) {
            return 0;
        }
        return _totalAmountPerEpoch
        .mul(_getUserBalancePerEpoch(msg.sender, epochId))
        .div(epochs[epochId]);
```

```
    }

    function _getPoolSize(uint128 epochId) internal view returns (uint) {
        // retrieve unilp token balance
        return _staking.getEpochPoolSize(_uniLP, _stakingEpochId(epochId));
    }


    function _getUserBalancePerEpoch(address userAddress, uint128 epochId) internal view returns (uint){
        // retrieve unilp token balance per user per epoch
        return _staking.getEpochUserBalance(userAddress, _uniLP, _stakingEpochId(epochId));
    }

    // compute epoch id from blocktimestamp and epochstart date
    function _getEpochId() internal view returns (uint128 epochId) {
        if (block.timestamp < epochStart) {
            return 0;
        }
        epochId = uint128(block.timestamp.sub(epochStart).div(epochDuration).add(1));
    }

    // get the staking epoch which is 1 epoch more
    function _stakingEpochId(uint128 epochId) pure internal returns (uint128) {
        return epochId + 1;
    }
}
```

6. 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 loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending ETH to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.
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.

7. Appendix C：Introduction to auditing tools

7.1 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.

7.2 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.

7.3 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.

7.4 Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

7.5 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.

7.6 ethersplay

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

7.7 ida-evm

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

7.8 Remix-ide

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

7.9 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