Cyberscope

Audit Report

Smartrade

October 2023

GitHub	https://github.com/SmarTradeDev/DepositContract
Commit	313f67ff072729a7c7491bf88da0105ed96741b5
Audited by	© cyberscope

Table of Contents

Review

Repository	https://github.com/SmarTradeDev/DepositContract
Commit	313f67ff072729a7c7491bf88da0105ed96741b5
Testing Deploy	https://testnet.bscscan.com/address/0x0ad36cfb3557dc1755556659339d09dbdb79f1a9

Audit Updates

Initial Audit	12 Oct 2023

Source Files

Filename	SHA256
contracts/deposit.sol	9041aa0d3863d3a2f79cb66180bebb5f3590392b63a5cde4e8798a79185c5457

Overview

Cyberscope audited the SmarTradeContract contract from the SmartTrade ecosystem. Its primary purpose is to manage staking of a specific token and provide rewards to users. It supports various features, including adding and editing staking packages with specified amounts, durations, and interest rates. Users can stake their tokens into these packages, specifying a referrer, and receive rewards based on the staking package's rate. The contract allows users to claim their rewards and provides a mechanism to unstake tokens. Additionally, the contract is designed with an owner who can modify package details, precision, and the referrer rate. It also includes some security measures to prevent contract calls from other contracts, ensuring that staking and claiming are primarily for externally owned accounts (EOAs). The contract's core logic revolves around staking, tracking rewards, and enabling users to interact with these features securely.

Roles

Owner

The owner has the authority to:

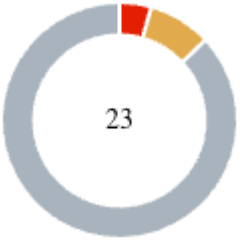
- `Set the precision of the contract.`
- `Set the referrer rate.`
- `Add and edit staking packages.`
- `Renew the owner's address.`
- `Claim contract tokens.`

User

The user has the authority to:

- `Stake tokens into specific packages.`
- `Claim rewards.`
- `Unstake their tokens.`

Findings Breakdown



Critical	1
Medium	2
Minor / Informative	20

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	2	0	0	0
Minor / Informative	20	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	OCV	Ownership Control Vulnerability	Unresolved
	ZD	Zero Division	Unresolved
	PNP	Potential Non-Existent Package	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	RVD	Redundant Variable Declaration	Unresolved
	OCTD	Transfers Contract's Tokens	Unresolved
	TSI	Tokens Sufficiency Insurance	Unresolved
	MU	Modifiers Usage	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	MVN	Misleading Variables Naming	Unresolved
	LOV	Lack of Validation	Unresolved
	TUU	Time Units Usage	Unresolved
	MEE	Missing Events Emission	Unresolved
	RSW	Redundant Storage Writes	Unresolved

●	L02	State Variables could be Declared Constant	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L23	ERC20 Interface Misuse	Unresolved

OCV - Ownership Control Vulnerability

Criticality	Critical
Location	contracts/deposit.sol#L49,67
Status	Unresolved

Description

The contract contains two functions, `initialize` and `renewOwner`, which together create a potential vulnerability related to ownership control. The issue arises when the original owner decides to renounce ownership by calling the `renewOwner` function with the zero address as the new owner. This action effectively removes the contract's owner. Subsequently, any user could exploit this situation by calling the `initialize` function and taking control of the contract's ownership. This poses a security risk and undermines the contract's intended owner control mechanism.

```
function initialize() public {
    require(owner == address(0));
    owner = msg.sender;
}

function renewOwner(address newOwner) public onlyOwner {
    owner = newOwner;
}
```

Recommendation

To enhance the security of the contract and prevent unauthorized ownership changes, the team should consider implementing a more robust ownership management mechanism. One approach would be to use a multisig wallet or a timelock mechanism to manage ownership changes, ensuring that they are securely executed and involve multiple authorized parties. Additionally, the team should thoroughly review the ownership-related functions and events to identify and address any potential vulnerabilities in the contract's ownership control logic.

ZD - Zero Division

Criticality	Medium
Location	contracts/deposit.sol#L136,174
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

```
newStaking.blockRewards = newStaking.stakeAmount * package.rate /
precision / 7200;
IToken(staking.stakeToken).transfer(staking.referrer,
(staking.claimedAmount - staking.stakeAmount) * referrerRate / precision);
```

Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

PNP - Potential Non-Existent Package

Criticality	Medium
Location	contracts/deposit.sol#L129
Status	Unresolved

Description

The contract's `stake` function allows users to stake their tokens into different packages, but it lacks proper validation for the provided `packageIdx`. Without validation, there's a risk that users could specify an invalid or out-of-range package index, causing the transaction to revert. This can be problematic, as it leaves room for user error or potential misuse of the contract.

```
Package memory package = packages[packageIdx];
```

Recommendation

The team is advised to enhance the robustness and user-friendliness of the contract by implementing a validation step for the `packageIdx` parameter in the `stake` function. The validation should ensure that the provided index is within the bounds of the packages array and that it corresponds to a valid package. If the provided index is not valid, the function should revert with an informative error message. This validation step will help prevent erroneous transactions and improve the overall usability of the contract.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/deposit.sol#L54,58,85,202
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setPrecision(uint256 precision_) public onlyOwner {
    precision = precision_;
}

function setReferrerRate(uint256 rRate) public onlyOwner {
    referrerRate = rRate;
}
...
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

RVD - Redundant Variable Declaration

Criticality	Minor / Informative
Location	contracts/deposit.sol#L43
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract declares some variables that are not used in a meaningful way from the contract. As a result, these variables are redundant.

```
bool public status;
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	contracts/deposit.sol#L202
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `depositToVault` function.

```
function depositToVault(address token, address to, uint256 amount) public
onlyOwner {
    IToken(token).transfer(to, amount);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/deposit.sol#L172,174,185
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
IToken(staking.stakeToken).transfer(staking.staker, rewards);
if(staking.closed) {
    IToken(staking.stakeToken).transfer(staking.referrer,
(staking.claimedAmount - staking.stakeAmount) * referrerRate / precision);
}

if(staking.claimedAmount < staking.stakeAmount) {
    IToken(staking.stakeToken).transfer(staking.staker,
staking.stakeAmount - staking.claimedAmount);
    staking.claimedAmount = staking.stakeAmount;
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/deposit.sol#L159,180
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(staking.staker == msg.sender, "not staker");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	contracts/deposit.sol#L132,143
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
newStaking.stakeAmount = package.amount * (10 **
IToken(tokenAddr).decimals());
IToken(newStaking.stakeToken).transferFrom(newStaking.staker,
address(this), newStaking.stakeAmount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	contracts/deposit.sol#L133,134,135
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

For instance, the `newStaking.startTime` is assigned the value of `block.number`, which in Solidity represents the current block number, not the current timestamp. This naming and usage discrepancy could lead to misunderstandings about the contract's logic.

```
newStaking.startTime = block.number;
newStaking.lastClaimTime = block.number;
newStaking.closeTime = block.number + package.period;
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

LOV - Lack of Validation

Criticality	Minor / Informative
Location	contracts/deposit.sol#L131,143
Status	Unresolved

Description

The contract's `stake` function assigns the `tokenAddr` to the `newStaking.stakeToken` variable without validating the tokenAddr itself. This lack of validation means that users could potentially pass the zero address as tokenAddr, which will lead to a transaction revert without a proper error message.

```
newStaking.stakeToken = tokenAddr;
IToken(newStaking.stakeToken).transferFrom(newStaking.staker,
address(this), newStaking.stakeAmount);
```

Recommendation

To enhance the security and reliability of the contract, it's crucial the team adds a validation step for the tokenAddr parameter within the stake function. The team should ensure that tokenAddr is a valid, non-zero address before proceeding with the staking process. If the provided address is the zero address, the function should revert with a clear error message, preventing any unintended behavior and potential issues associated with invalid addresses.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	contracts/deposit.sol#L80,87,136
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
package.period = dayCount * 7200;
packages[packageIdx].period = dayCount * 7200;
newStaking.blockRewards = newStaking.stakeAmount * package.rate /
precision / 7200;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/deposit.sol#L55,59,68,82,86,87,88,127,157,178
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
precision = precision_;
referrerRate = rRate;
owner = newOwner;
packages.push(package);
packages[packageIdx].amount = amount;
packages[packageIdx].period = dayCount * 7200;
packages[packageIdx].rate = rate;
...
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	contracts/deposit.sol#L78,130
Status	Unresolved

Description

The contract's `stake` function creates a new Staking struct object by individually writing each of its properties to storage. This approach results in redundant and inefficient storage writes, consuming unnecessary gas. Each property assignment to storage triggers a separate storage update, leading to increased gas costs and potential performance inefficiencies. The same issue also exists in the `addPackage` function and the creation of the Package object.

```
Staking memory newStaking;
newStaking.stakeToken = tokenAddr;
newStaking.stakeAmount = package.amount * (10 **
IToken(tokenAddr).decimals());
newStaking.startTime = block.number;
newStaking.lastClaimTime = block.number;
newStaking.closeTime = block.number + package.period;
newStaking.blockRewards = newStaking.stakeAmount * package.rate /
precision / 7200;
newStaking.referrer = referrer;
newStaking.claimedAmount = 0;
newStaking.closed = false;
newStaking.staker = msg.sender;
newStaking.packageIdx = packageIdx;
```

Recommendation

To optimize gas usage and improve the efficiency of the `stake` function, the team should consider creating the Staking struct object in a single storage write operation. By bundling the struct properties together and writing them to storage as a whole, you can reduce gas costs and improve the contract's overall performance. This optimization will result in more efficient use of storage and contribute to lower transaction costs for users.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/deposit.sol#L43
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
bool public status
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	contracts/deposit.sol#L68
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
owner = newOwner
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	contracts/deposit.sol#L106,120,181,207
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
stakings[i].closed == false
stakings[i].closed == true
require(staking.closed == false, "already closed")
require(isContract(msg.sender) == false, "called by contract")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/deposit.sol#L198
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
rewards = rewards + rewards * (end - start) /
(stakings[stakeIdx].closeTime - start) * 4 / 11
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	contracts/deposit.sol#L78,130
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
Package memory package
Staking memory newStaking
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/deposit.sol#L68
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
owner = newOwner
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	contracts/deposit.sol#L213
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            size := extcodesize(_addr)
        }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/deposit.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.17;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L23 - ERC20 Interface Misuse

Criticality	Minor / Informative
Location	contracts/deposit.sol#L7,8,9
Status	Unresolved

Description

The ERC20 is a standard interface for tokens on the blockchain. It defines a set of functions and events that a contract must implement in order to be considered an ERC20 token. According to the ERC20 interface, the transfer function returns a bool value, which indicates the success or failure of the transfer. If the transfer is successful, the function returns true. If the transfer fails, the function returns false. The contract implements the transfer function without the return value.

```
function approve(address spender, uint value) external;
function transfer(address spender, uint value) external;
function transferFrom(address from, address to, uint value) external;
```

Recommendation

The incorrect implementation of the ERC20 interface could potentially lead to problems when interacting with the contract, as other contracts or applications that expect the ERC20 interface may not behave as expected.

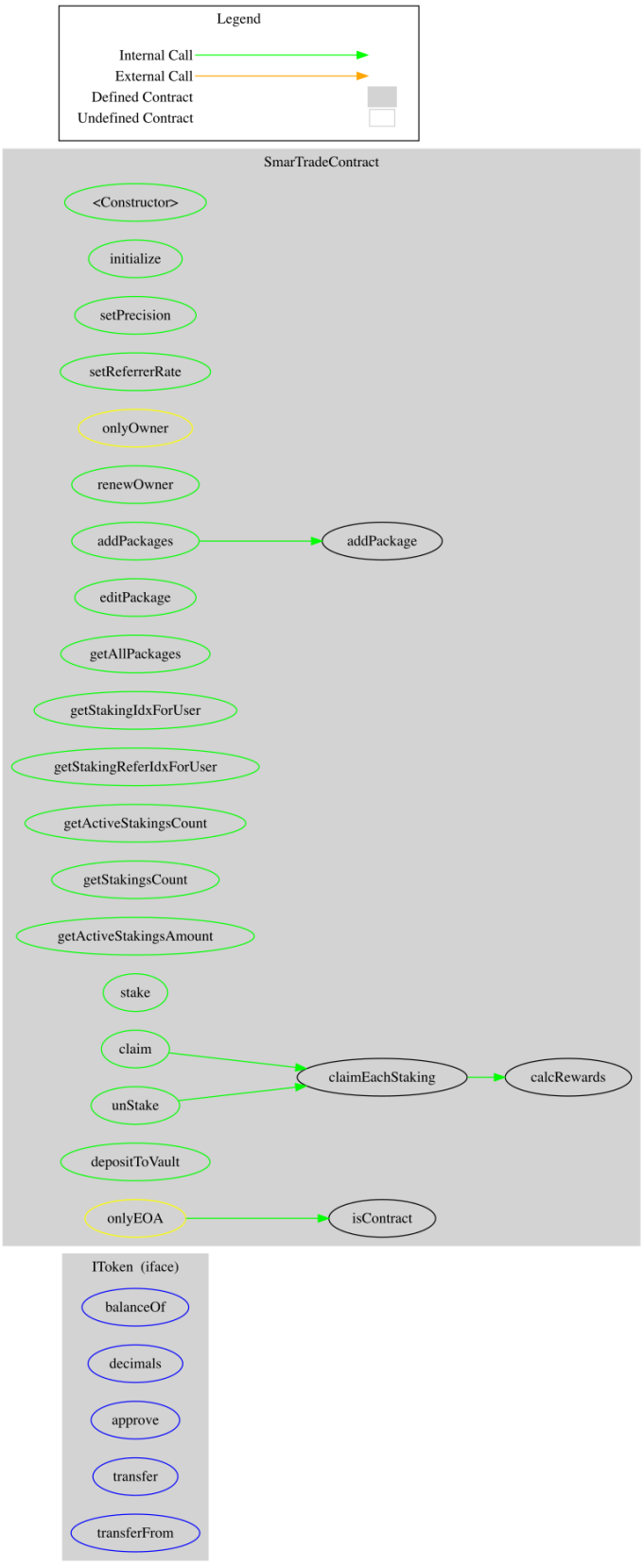
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IToken	Interface			
	balanceOf	External		-
	decimals	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
SmarTradeContract	Implementation			
		Public	✓	-
	initialize	Public	✓	-
	setPrecision	Public	✓	onlyOwner
	setReferrerRate	Public	✓	onlyOwner
	renewOwner	Public	✓	onlyOwner
	addPackages	Public	✓	onlyOwner
	addPackage	Public	✓	onlyOwner
	editPackage	Public	✓	onlyOwner
	getAllPackages	Public		-
	getStakingIdxForUser	Public		-

	getStakingReferIdxForUser	Public		-
	getActiveStakingsCount	Public		-
	getStakingsCount	Public		-
	getActiveStakingsAmount	Public		-
	stake	Public	✓	onlyEOA
	claim	Public	✓	onlyEOA
	claimEachStaking	Public	✓	onlyEOA
	unStake	Public	✓	onlyEOA
	calcRewards	Public		-
	depositToVault	Public	✓	onlyOwner
	isContract	Private		

Inheritance Graph

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
SmarTradeContract
<Constructor>
initialize
setPrecision
setReferrerRate
onlyOwner
renewOwner
addPackages
addPackage
editPackage
getAllPackages
getStakingIdxForUser
getStakingReferIdxForUser
getActiveStakingsCount
getStakingsCount
getActiveStakingsAmount
stake
claim
claimEachStaking
calcRewards
unStake
depositToVault
onlyEOA
isContract
IToken (iface)
balanceOf
decimals
approve
transfer
transferFrom

Summary

Smartrade contract implements a staking mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io