Cyberscope

Audit Report

Clover

September 2023

CloverPot 712b47be94437b18662044a04ff7d7255809f6d7e804d9419f965c8587df2ab9

CloverFarm 206f368ee64c27c8337eca8e3fae5ebbff5f82e15495696b0987c4037433064b

Audited by © cyberscope

Table of Contents

Review

Audit Updates

Initial Audit	14 Sep 2023

Source Files

Filename	SHA256
CloverPot.sol	712b47be94437b18662044a04ff7d7255809f6d7e804d9419f965c8587df2ab9
CloverFarm.sol	206f368ee64c27c8337eca8e3fae5ebbff5f82e15495696b0987c4037433064b

Overview

Cyberscope audited and comprehensively evaluated the security and functionality of two smart contracts from the Clover ecosystem, "CloverFarm" and "CloverPot". CloverFarm presents a decentralized finance (DeFi) platform with staking and referral mechanisms, enabling users to stake Ether (ETH) in various plans, compound their earnings, and receive rewards and referral bonuses. On the other hand, CloverPot serves as a reward distribution system, tracking user interactions and distributing rewards at specific intervals.

Roles

CloverPot

Owner

The Owner role has authority over the following functions:

- `function renounceOwnership()`
- `function transferOwnership(address payable newOwner)`
- `function setContractAddress(address _contractAddress)`

Contract

The Contract role has authority over the following functions:

- `function update(uint256 _time, address _user)`

User

The User role can interact with the following functions:

- `function owner()`
- `function getHistories()`

CloverFarm

Owner

The Owner role has authority over the following functions:

- `function renounceOwnership()`
- `function transferOwnership(address payable newOwner)`

FeeWallet

The FeeWallet role has authority over the following functions:

- `function updateFeeWallet(address wallet)`

User

The User role can interact with the following functions:

- `function stake(address referrer, uint8 plan)`
- `function withdraw()`
- `function withdrawReferralBonus()`
- `function getRefInfo(address _addr)`
- `function compound(uint8 plan)`
- `function canHarvest(address userAddress)`
- `function canRestake(address userAddress, uint8 plan)`
- `function getContractBalance()`
- `function getPlanInfo(uint8 plan)`
- `function getUserDividends(address userAddress)`
- `function getUserActiveStaking(address userAddress)`
- `function getUserTotalWithdrawn(address userAddress)`
- `function getUserCheckpoint(address userAddress)`
- `function getUserReferrer(address userAddress)`
- `function getUserTotalReferrals(address userAddress)`
- `function getUserReferralBonus(address userAddress)`
- `function getUserReferralTotalBonus(address userAddress)`
- `function getUserReferralWithdrawn(address userAddress)`
- `function getUserAvailable(address userAddress)`
- `function getUserAmountOfDeposits(address userAddress)`

- `function getUserTotalDeposits(address userAddress)`
- `function getUserDepositInfo(address userAddress, uint256 index)`
- `function getUserPlanInfo(address userAddress)`
- `function getSiteInfo()`
- `function getUserInfo(address userAddress)`

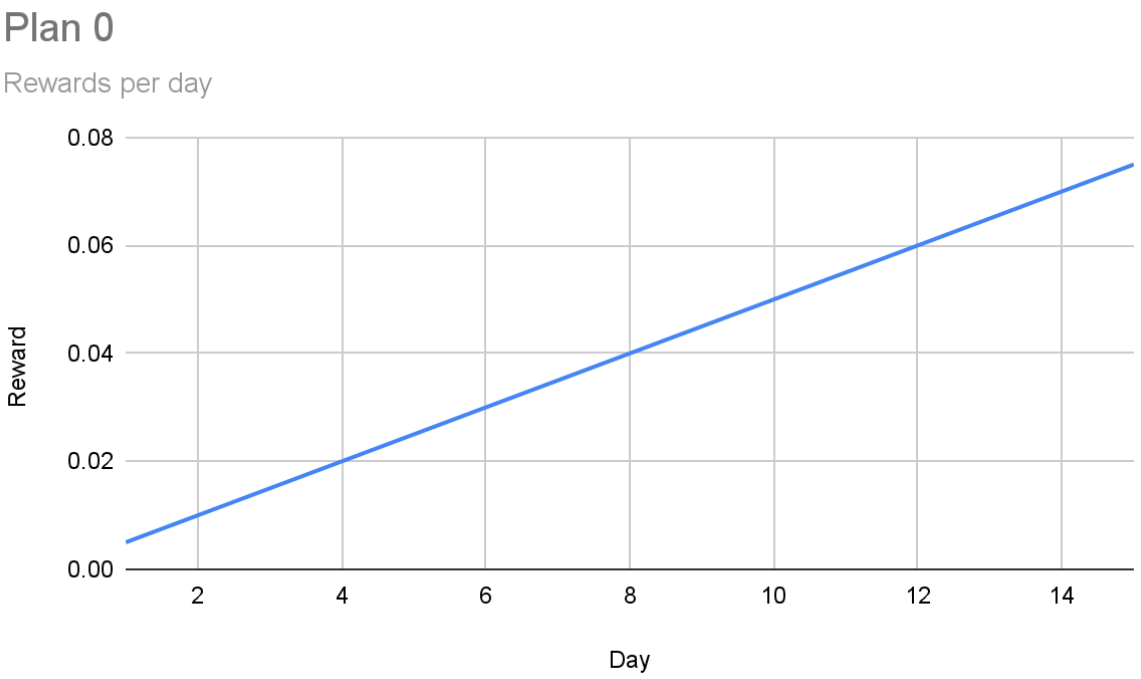
Reward Distribution

The contract's reward distribution mechanism operates on a linear distribution model, commencing from the initiation of staking and continuing throughout the staking period. Users participating in the staking process have the opportunity to choose from one of four preconfigured plans, each with its distinct rewards multiplier. The rewards multiplier is detailed in the rewards sheet and is graphically represented on the Y-Axis.

NOTE: All graphs were generated using the minimum participation amount for each plan as a base.

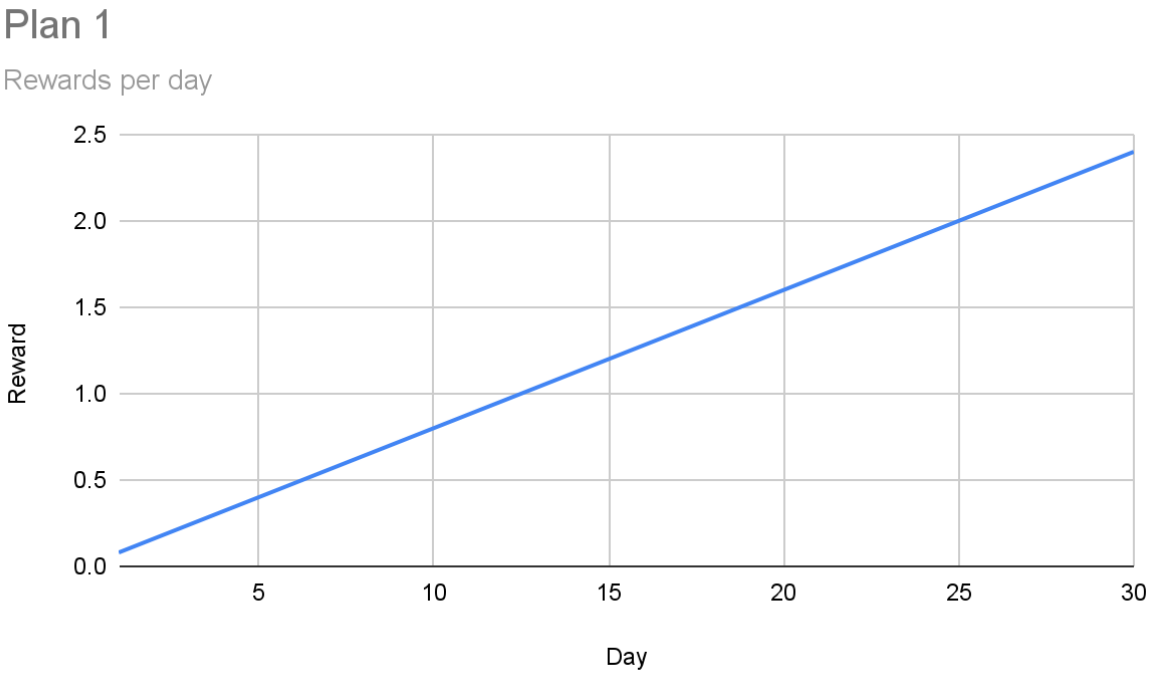
Plan 0

- Time: 15 days
- Percentage Multiplier: 5%
- Minimum Participation Amount: 0.1 ether



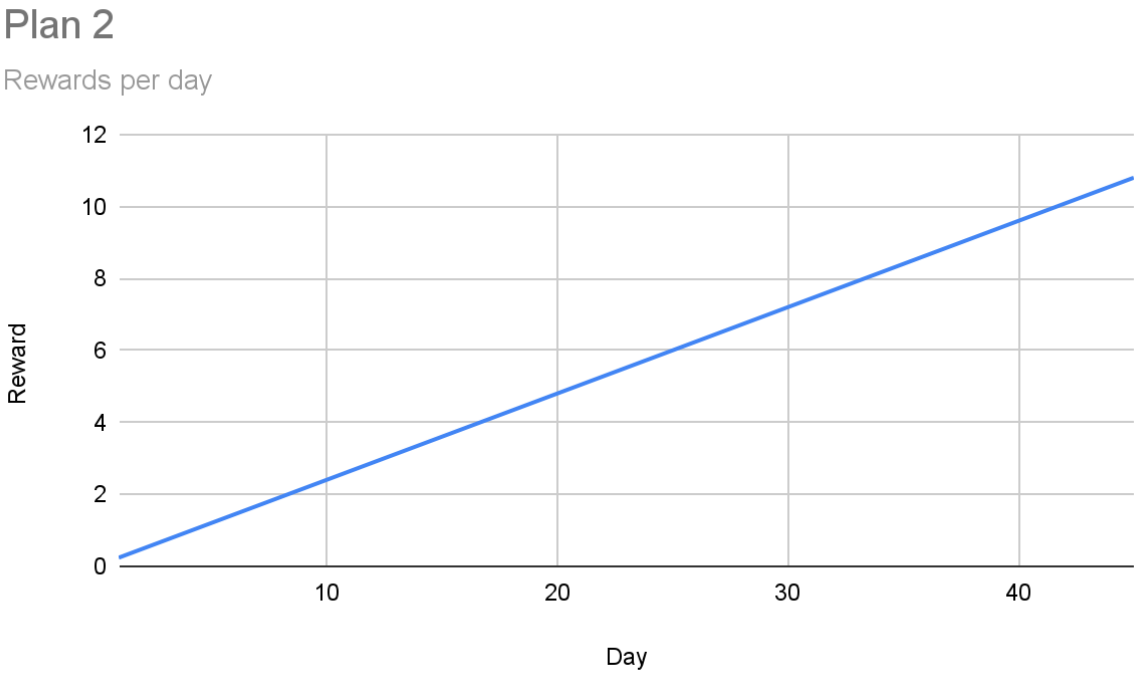
Plan 1

- Time: 30 days
- Percentage Multiplier: 8%
- Minimum Participation Amount: 1 ether



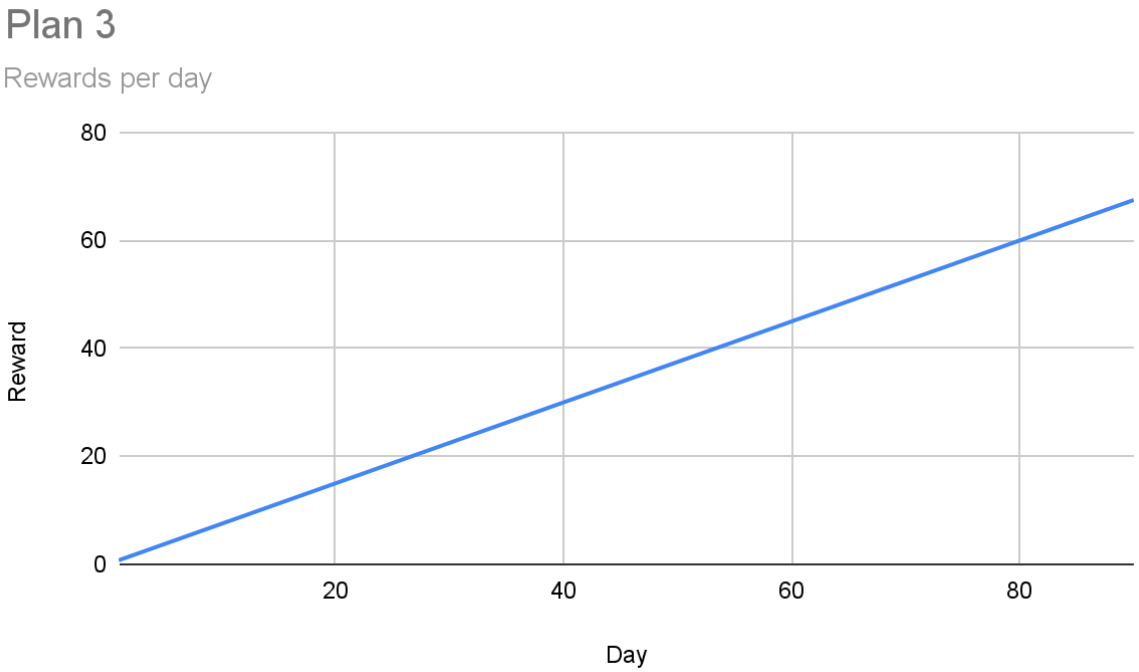
Plan 2

- Time: 45 days
- Percentage Multiplier: 12%
- Minimum Participation Amount: 2 ether

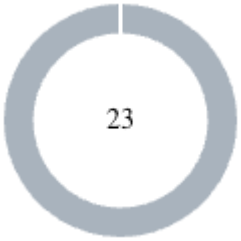


Plan 3

- Time: 90 days
- Percentage Multiplier: 15%
- Minimum Participation Amount: 5 ether



Findings Breakdown



Critical	0
Medium	0
Minor / Informative	23

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	23	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	MRBH	Misleading Ref Bonus Handling	Unresolved
	REA	Redundant Event Argument	Unresolved
	RCS	Redundant Conditional Statement	Unresolved
	URA	Unset Referrer Address	Unresolved
	TVO	Time Variables Optimization	Unresolved
	AAO	Accumulated Amount Overflow	Unresolved
	IRD	Inconsistent Reward Distribution	Unresolved
	CR	Code Repetition	Unresolved
	ADO	Array Declaration Optimization	Unresolved
	MSC	Missing Sanity Check	Unresolved
	RED	Redundant Event Declaration	Unresolved
	RSML	Redundant SafeMath Library	Unresolved
	RSK	Redundant Storage Keyword	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved

●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L09	Dead Code Elimination	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

MRBH - Misleading Ref Bonus Handling

Criticality	Minor / Informative
Location	CloverFarm.sol#L316
Status	Unresolved

Description

Within the contract's code, there is an inconsistency in how the `totalRefBonus` variable is handled based on the contract's balance condition. When `contractBalance` is less than `totalAmount`, the code correctly adjusts the totalAmount and assigns the remaining amount to the user's bonus. However, when contractBalance is greater than or equal to totalAmount, the totalRefBonus is updated only in this case. This can be misleading, as it suggests that the totalRefBonus is related to the contractBalance condition, which may not be accurate.

```
if (contractBalance < totalAmount) {
    user.bonus = totalAmount.sub(contractBalance);
    totalAmount = contractBalance;
} else {
    if (referralBonus > 0)
        totalRefBonus = totalRefBonus.add(referralBonus);
}
```

Recommendation

To ensure clarity and consistency in the code, it is recommended to separate the handling of `totalRefBonus` from the `contractBalance` condition. The adjustment of totalRefBonus should be done based on the `referralBonus` condition itself, rather than being tied to the contractBalance. This ensures that the logic accurately reflects when and how the totalRefBonus is modified, making the code more transparent and less prone to misunderstanding.

REA - Redundant Event Argument

Criticality	Minor / Informative
Location	CloverFarm.sol#L194,286
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 `RefBonus` event is declared with an argument called `level`. However, when the event is emitted, this argument is always given the value of 0. As a result, this argument is redundant.

```
event RefBonus(
    address indexed referrer,
    address indexed referral,
    uint256 indexed level,
    uint256 amount
);
emit RefBonus(referrer, msg.sender, 0, refamount);
```

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.

RCS - Redundant Conditional Statement

Criticality	Minor / Informative
Location	CloverFarm.sol#L244
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 plans used for staking are preconfigured and cannot be modified. Hence, the `require(plan < 4, "Invalid plan")` check is redundant.

```
require(msg.value >= plans[plan].minAmount, "Below than min amount");
require(plan < 4, "Invalid plan");
```

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.

URA - Unset Referrer Address

Criticality	Minor / Informative
Location	CloverFarm.sol#L261
Status	Unresolved

Description

Within the contract's staking flow, certain operations are executed when the referrer address is not the zero address. However, upon closer inspection, it becomes evident that the `referrer` property of the `User` struct is never being set. Consequently, the `referrer` property is effectively set to the zero address for every user, regardless of whether a referrer exists or not.

```
if (referrer != address(0)) {

    if (refInfos[referrer].totalAmount == 0) {
        refInfos[referrer].count = refInfos[referrer].count + 1;
    }
    refInfos[referrer].totalAmount =
refInfos[referrer].totalAmount.add(amount);
    uint256 refamount = 0;
    if (refInfos[referrer].totalAmount >= 50 * (10 ** 18)) {
        refamount =
amount.mul(REFERRAL_PERCENTS[0]).div(PERCENTS_DIVIDER);
    } else if (refInfos[referrer].totalAmount >= 25 * (10 ** 18)) {
        refamount =
amount.mul(REFERRAL_PERCENTS[1]).div(PERCENTS_DIVIDER);
    } else if (refInfos[referrer].totalAmount >= 10 * (10 ** 18)) {
        refamount =
amount.mul(REFERRAL_PERCENTS[2]).div(PERCENTS_DIVIDER);
    } else if (refInfos[referrer].totalAmount >= 5 * (10 ** 18)) {
        refamount =
amount.mul(REFERRAL_PERCENTS[3]).div(PERCENTS_DIVIDER);
    } else if (refInfos[referrer].totalAmount >= 2 * (10 ** 18)) {
        refamount =
amount.mul(REFERRAL_PERCENTS[4]).div(PERCENTS_DIVIDER);
    } else if (refInfos[referrer].totalAmount >= 1 * (10 ** 18)) {
        refamount =
amount.mul(REFERRAL_PERCENTS[5]).div(PERCENTS_DIVIDER);
    } else {
        refamount =
amount.mul(REFERRAL_PERCENTS[6]).div(PERCENTS_DIVIDER);
    }
    users[referrer].bonus = users[referrer].bonus.add(refamount);
    users[referrer].totalBonus =
users[referrer].totalBonus.add(refamount);

    emit RefBonus(referrer, msg.sender, 0, refamount);
}
```

Recommendation

To enable the intended functionality of tracking referrers for users in the staking flow, it is recommended to implement a mechanism for setting and updating the `referrer` property within the `User` struct. With the proper implementation, the contract can correctly attribute referrers to users and execute operations accordingly, enhancing the referral functionality.

TVO - Time Variables Optimization

Criticality	Minor / Informative
Location	CloverFarm.sol#L360,494,532,645
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 uses repetitively the `1 days` value for certain operations. However, this value has already been calculated and assigned to the variable `TIME_STEP`. As a result, the contract consumes more gas for executing these operations.

```
uint256 finish = user.deposits[i].start.add(
    plans[user.deposits[i].plan].time.mul(1 days)
);
```

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.

AAO - Accumulated Amount Overflow

Criticality	Minor / Informative
Location	CloverFarm.sol#L297,321,413,416,429
Status	Unresolved

Description

The contract is using variables to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

```
totalStaked = totalStaked.add(amount)
totalRefBonus = totalRefBonus.add(referralBonus)
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum value that can be stored in its data type.

IRD - Inconsistent Reward Distribution

Criticality	Minor / Informative
Location	CloverPot.sol#L81,105CloverFarm.sol#L252
Status	Unresolved

Description

The contract features a staking flow with a reward mechanism that allows users to claim ETH from the `CloverPot` contract if they stake with a 2-hour delay between stakes. However, the current implementation results in inconsistent reward distribution. Users may claim varying amounts of ETH, including potentially receiving no reward or receiving a substantial amount of ETH.

```
cloverPot.update(block.timestamp, msg.sender);

if (_time - moment >= DELAY_TIME)
    getReward(lastUser);

function getReward(address _user) private {
    emit GetReward(_user, address(this).balance);
    histories.push(History(_user, address(this).balance, block.timestamp));
    payable(_user).transfer(address(this).balance);
}
```

Recommendation

To ensure a fair and consistent reward distribution system, it is recommended to revisit the logic for calculating and distributing rewards. The team could implement a consistent and predictable method for calculating rewards based on user actions. Consider using a formula or mechanism that ensures proportional rewards based on the user's staking behavior, such as the amount staked and the duration of the stake.

CR - Code Repetition

Criticality	Minor / Informative
Location	CloverPot.sol#L86,95 CloverFarm.sol#L359,493,531,644
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
moment = _time;
lastUser = _user;
emit UserUpdated(lastUser, moment);

uint256 finish = user.deposits[i].start.add(
    plans[user.deposits[i].plan].time.mul(1 days)
);
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

ADO - Array Declaration Optimization

Criticality	Minor / Informative
Location	CloverFarm.sol#L156
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 `plans` array is declared with a dynamic length. However, it is evident that this array will always have a fixed length of 4, as it is initialized with four elements. Using a dynamic array in this scenario is redundant and consumes additional storage that could be optimized.

```
Plan[] internal plans;

plans.push(Plan(15, 50, 0.1 ether));
plans.push(Plan(30, 80, 1 ether));
plans.push(Plan(45, 120, 2 ether));
plans.push(Plan(90, 150, 5 ether));

require(plan < 4, "Invalid plan");
```

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.

To optimize storage usage and improve code efficiency, it is recommended to declare the plans array with a fixed size of 4, given that it will always contain four elements.

MSC - Missing Sanity Check

Criticality	Minor / Informative
Location	CloverPot.sol#L102 CloverFarm.sol#L462
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The contract does not check if the following variables are valid contract addresses or not.

```
mainContract = _contractAddress
feeWallet = wallet
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

RED - Redundant Event Declaration

Criticality	Minor / Informative
Location	CloverPot.sol#L68 CloverFarm.sol#L201,202,211
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 certain events in its code. However, these events are not emitted within the contract's functions. As a result, these declared events are redundant and serve no purpose within the contract's current implementation.

```
event PoolUpdated(address _lastUser, uint256 _timestamp);
event SwapETHForTokens(uint256 amountIn, address[] path);
event SwapAndLiquify(
    uint256 ethSwapped,
    uint256 tokenReceived,
    uint256 ethsIntoLiqudity
);
event DevWalletUpdated(
    address indexed oldDevWallet,
    address indexed newDevWallet
);
```

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.

A recommended approach would be to either remove the declared events that are not being emitted or to incorporate the necessary emit statements within the contract's functions to actually emit these events when relevant actions occur.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	CloverFarm.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	CloverFarm.sol#L223
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
cloverPot
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	CloverPot.sol#L54
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.

```
uint256 public amountPerHour
```

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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	CloverPot.sol#L82,83,100,105CloverFarm.sol#L137,388
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 _time
address _user
address _contractAddress
uint256[] public REFERRAL_PERCENTS = [300, 250, 200, 150, 100, 70, 50]
address _addr
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	CloverPot.sol#L102
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.

```
mainContract = _contractAddress
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	CloverFarm.sol#L14,43
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function add32(uint32 a, uint32 b) internal pure returns (uint32) {
        uint32 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	CloverFarm.sol#L497,509
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.

```
uint256 share = user
                    .deposits[i]
                    .amount
                    .mul(plans[user.deposits[i].plan].percent)
                    .div(PERCENTS_DIVIDER)
totalAmount = totalAmount.add(
                        share.mul(to.sub(from)).div(TIME_STEP)
                    )
```

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	CloverFarm.sol#L490,528
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.

```
uint256 totalAmount
```

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	CloverPot.sol#L87,102CloverFarm.sol#L462
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.

```
lastUser = _user
mainContract = _contractAddress
feeWallet = wallet
```

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	CloverFarm.sol#L117,693
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 {
            sstore(NAME_SLOT, NAME_HASH)
        }

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	CloverPot.sol#L3CloverFarm.sol#L3
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 ^0.8.0;
pragma solidity ^0.8.4;
```

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.

Functions Analysis

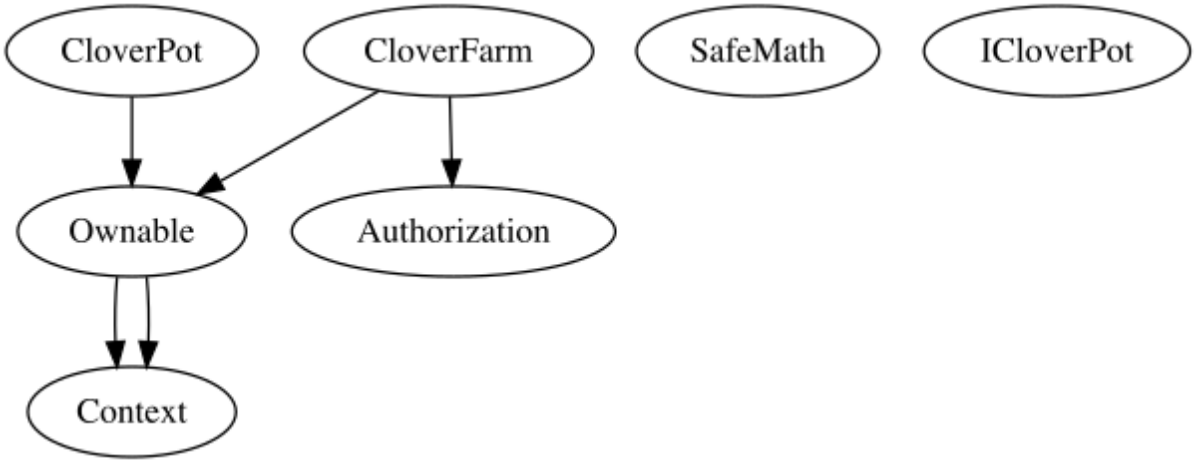
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Context	Implementation			
		Public	✓	-
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
CloverPot	Implementation	Ownable		
		External	✓	-
		External	Payable	-
		Public	✓	-
	update	Public	✓	onlyContract
	setContractAddress	External	✓	onlyOwner
	getReward	Private	✓	

	getHistories	Public		-
SafeMath	Library			
	add	Internal		
	add32	Internal		
	sub	Internal		
	sub	Internal		
	mul	Internal		
	mul32	Internal		
	div	Internal		
	div	Internal		
Context	Implementation			
		Public	✓	-
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner

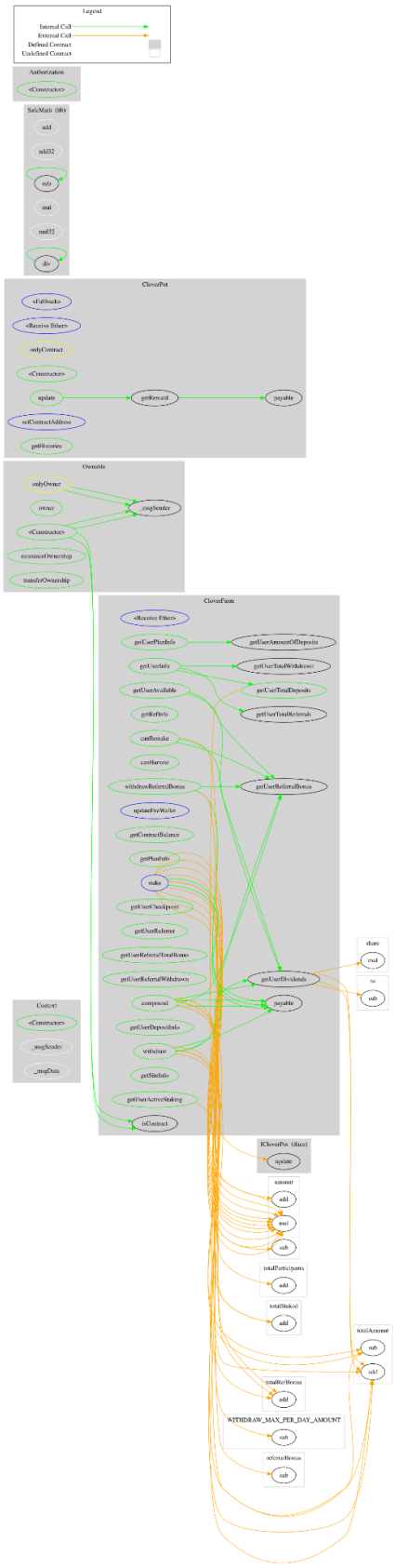
Authorization	Implementation			
		Public	✓	-
ICloverPot	Interface			
	update	External	✓	-
CloverFarm	Implementation	Ownable, Authorization		
		Public	✓	-
		External	Payable	-
	stake	External	Payable	-
	withdraw	Public	✓	-
	withdrawReferralBonus	Public	✓	-
	getRefInfo	Public		-
	compound	Public	Payable	-
	canHarvest	Public		-
	canRestake	Public		-
	updateFeeWallet	External	✓	-
	getContractBalance	Public		-
	getPlanInfo	Public		-
	getUserDividends	Public		-
	getUserActiveStaking	Public		-
	getUserTotalWithdrawn	Public		-
	getUserCheckpoint	Public		-

	getUserReferrer	Public		-
	getUserTotalReferrals	Public		-
	getUserReferralBonus	Public		-
	getUserReferralTotalBonus	Public		-
	getUserReferralWithdrawn	Public		-
	getUserAvailable	Public		-
	getUserAmountOfDeposits	Public		-
	getUserTotalDeposits	Public		-
	getUserDepositInfo	Public		-
	getUserPlanInfo	Public		-
	getSiteInfo	Public		-
	getUserInfo	Public		-
	isContract	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Authorization
<Constructor>
SafeMath (lib)
add
add32
sub
mul
mul32
div
CloverPot
<Fallback>
<Receive Ether>
onlyContract
<Constructor>
update
getReward
payable
setContractAddress
getHistories
Ownable
onlyOwner
owner
_msgSender
<Constructor>
renounceOwnership
transferOwnership
CloverFarm
<Receive Ether>
getUserPlanInfo
getUserAmountOfDeposits
getUserInfo
getUserTotalWithdrawn
getUserAvailable
getUserTotalDeposits
getRefInfo
getUserTotalReferrals
canRestake
canHarvest
withdrawReferralBonus
getUserReferralBonus
updateFeeWallet
getContractBalance
getPlanInfo
stake
getUserCheckpoint
getUserReferer
getUserReferralTotalBonus
getUserReferralWithdrawn
getUserDividends
compound
payable
getUserDepositInfo
withdraw
getSiteInfo
getUserActiveStaking
isContract
Context
<Constructor>
_msgSender
_msgData
share
mul
to
sub
ICloverPot (iface)
update
amount
add
sub
totalParticipants
add
totalStaked
add
totalAmount
sub
add
totalRefBonus
add
WITHDRAW_MAX_PER_DAY_AMOUNT
sub
referralBonus
sub

Summary

Clover contract implements a staking and rewards 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