BAIL security

BAILSEC.IO

EMAIL : OFFICE@BAILSEC.IO

TWITTER : @BAILSECURITY

TELEGRAM : @HELLOATBAILSEC

FINAL REPORT:

Pharaoh Farm Pot

May 2023

Disclaimer:

Security assessments are limited by time and the information provided by the client. Therefore, the findings in this report should not be considered a complete list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damage incurred because of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1.Executive Overview

This executive summary provides a summary of the audit report conducted for the PharaoFarmPoT smart contract, which is an investment contract enabling token deposits and rewards over a 365-day period. The contract has undergone several resolution rounds to address bugs, and it is advised that the developer team conducts comprehensive testing before deploying it on the mainnet.

Users can deposit tokens through the "deposit" function, which transfers tokens to the contract and applies a 15% fee. After tax, users receive a "maxPayout" equal to 3.65 times the deposited amount. Once tokens are deposited, they become permanently locked in the contract and cannot be withdrawn.

The expected payout depends on the initial deposit and the contract balance at the time of the claim. Daily rewards are calculated based on the contract balance as follows:

- Contract balance >= 10,000,000 tokens: 1% of the initial deposit
- Contract balance >= 5,000,000 tokens: 0.75% of the initial deposit
- Contract balance >= 2,500,000 tokens: 0.5% of the initial deposit
- Contract balance below 2,500,000 tokens: 0.25% of the initial deposit

A withdrawal tax is applied based on the ratio of the initial deposit to the contract balance. The tax rates range from 60% for an initial deposit of 10% or more of the contract balance to 10% for an initial deposit below 1% of the contract balance.

The "maxPayout" amount is compared to the total claimed amount before applying the tax. Referral rewards are also implemented, allowing up to 7 referral levels. Referrers receive a percentage of the deposited amount, with the direct referrer receiving 10%, the referrer of the referrer receiving 2.5%, and other upstream referrers receiving 0.5%.

If a user exceeds the "maxPayout," the contract provides a "doubleUp" function, enabling the user to start a new round of rewards by depositing twice the initial amount. Any excess amount beyond the "maxPayout" is stored in the "previousDoubleUp" variable and added to the available rewards when "doubleUp" is called. An edge case occurs when a user accumulates rewards that fully cover the new initial balance within "doubleUp"; in such cases, the excess amount is assigned to the "previousDoubleUp" variable for the users next reward claim.

During deployment, the deployer address is assigned to "firstAddress" and accumulates rewards even if the "maxPayout" is exceeded. However, this feature was removed in the resolution rounds, and rewards cannot be granted to the deployer address anymore. The "claimRewardsFirstAddress" function allows the deployer address to claim rewards without any withdrawal fee.

The audit report recommends that the management carefully review the reports contents, address the identified issues, and implement the recommended improvements. These steps will enhance PharaoFarmPoTs reputation as a trusted provider of decentralized financial services on the BNB blockchain.

NOTE: Since several resolution rounds were needed due to bugs which were introduced within the resolutions, we highly encourage the developer team to perform extensive testing before mainnet deployment. All these bugs were fixed now, but due to the changes to the contract, tests are still mandatory.

The provided link highlights the differences between the pre-audit version and Resolution 3 of the PharaoFarmPoT contract: https://www.diffchecker.com/5hS0PpUn/

1. Project Details

Project	Pharao Farm POT
Website	https://cairofinance.app/farmpot
Type	Decentralized Financial Services (DEFI)
Language	Solidity
Methods	Manual review
Deployed contract	-

2. Detections Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (No change made)
High	3	2		1
Medium	1			1
Low	2	2		
Informational	2	2		
Total	8	6		2

2.1 Detections Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users sensitive data. It also has the potential to cause severe damage to the clients reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless, the issue should be fixed immediately
Informational	Effects are small and do not pose an immediate danger to the project or users

3. Detections

Issue	The availableRewards function is severely flawed
Severity	**High**
Description	The current implementation of the availableRewards function in the contract has critical flaws that affect the payout calculations. Two scenarios illustrate these issues: 1. In the first scenario, when a user reaches the maximum payout and calls the doubleUp function, the flawed availableRewards function assigns an excessively large value to the _availableRewards variable. This results in granting a completely new maximum payout without the user transferring any tokens to the contract. 2. The second scenario occurs when a user reaches their exact maximum payout without exceeding it. In this case, the availableRewards function incorrectly returns zero, preventing the user from accumulating rewards for the previousDoubleUp variable. There is also a minor issue with the code that compares the claimed rewards and available rewards to the maximum payout. The comparison should use > instead of >= to handle the situation correctly. It is crucial to address these issues to ensure accurate payout calculations and prevent unintended consequences for users.
Recommendations	To address the first issue, there are two potential solutions for modifying the availableRewards function. One option is to return the remaining amount until reaching the maximum payout limit when claimedRewards + availRewards exceeds the limit. This eliminates the need for the previousDoubleUp variable. Alternatively, the function could be adjusted to return the entire available rewards, aligning it with the logic of previousDoubleUp. The decision on which solution to implement should be made by the contract developer, considering their project requirements. They should carefully evaluate the implications and choose the approach that best

	suits their needs. Additionally, it is suggested to exclude the case of claimedRewards = maxPayout from returning zero. This adjustment ensures that any previousDoubleUp rewards can be granted correctly, allowing users to receive their entitled rewards.
Recommendations	It has been observed that the desired logic for follow-up rounds to be free appears contradictory based on a specific line of code. However, it was decided to keep the code as-is despite the potential misunderstanding in the clients desired logic. The assumption is supported by the fact that the doubleUp function can only be called when the sum of available and claimed rewards exceeds the maximum payout. This ensures that the return value of the availableRewards function is always equal to the maximum payout. Regarding the second issue, it has also been acknowledged and addressed. Resolution 2 was implemented after discussing it with the team, modifying the availableRewards function to return the available rewards (avail) for the second case. This modification ensures that both the claimRewards and doubleUp functions work correctly, effectively resolving the first issue.

Issue	Users will lose rewards if they call claim within two days
Severity	**High**
Description	Due to Solidity's rounding down behavior, users will only receive rewards for a single complete day, resulting in potential loss of rewards. For example, if a user invokes the claimRewards function after 100,000 seconds since their last claim, they will only receive rewards for 86,400 seconds, losing out on 13,600 seconds of potential rewards. This situation applies to any time duration between two days.
Recommendations	To prevent the potential loss of user rewards, it is suggested to

	consider changing the reward calculation logic from days to hours or even seconds. By making this switch, the system can ensure that users receive rewards for the exact duration they have participated, without any loss.
Comments	The issue of potential reward losses due to Solidity's rounding down behavior has been resolved by switching to an hourly reward calculation logic. The contract now offers different reward rates per hour, allowing users to claim rewards more frequently. The check restricting users to claiming rewards once per day has been replaced with a new check allowing them to claim rewards once per hour, ensuring no potential reward losses. Alternatively, if desired, the daily check can be retained, allowing users to claim rewards once per day, still avoiding reward losses. Both approaches are considered satisfactory solutions.

Issue	**Mathematical operations within refer are flawed.**
Severity	**High**
Description	The refer function is currently the only function that correctly determines if and how much rewards the referrer should receive. It appropriately checks available + claimed + rew < max and grants the full amount of rew to the referralRewards variable. However, there is a flaw in the calculation for the leftover amount when exceeding the max value. The code snippet is as follows: *else if (max - claimed + available > 0)* *uint256 gib = max - claimed + available;* *if (gib > rew) {* *gib = rew;* *}*

	The issue lies in the missing parentheses in the calculation for the leftover amount. It should be corrected to: *else if (max - (claimed + available) > 0)* This correction ensures the calculation of the correct leftover amount. In the current implementation, this case will always follow the logic below, even if the claimed amount alone exceeds max, resulting in rewards being granted when they shouldnt be. To avoid potential reverts due to underflow in the else clause, all used variables (max, claimed, and available) should be converted to int and then back to uint for their secondary uses. Additionally, the following line should be modified to use <= instead of <: *if (available + claimed + rew < max)* In fact, it can be <= because the equal case means that the entire rew amount can still be granted.
Recommendations	To prevent granting rewards when the referrer exceeds the maxPayout, it is recommended to add parentheses around the highlighted code snippets. This will ensure that the condition is evaluated correctly. Here is the modified code: *else if ((max - (claimed + available)) > 0) {* *uint256 gib = max - claimed + available;* *if (gib > rew) {* *gib = rew;* *}* *}* By adding the parentheses, the correct calculation for the leftover amount is performed, and rewards will not be granted if the referrer

	exceeds the maxPayout. Additionally, it is advised to address the exclusion of the equal case. The condition available + claimed + rew < max can be modified to available + claimed + rew <= max. This adjustment ensures that the equal case is considered, allowing the entire rew amount to be granted.
Comments	The issue has been resolved by addressing the missing parentheses and converting the necessary variables to the correct type. However, it has been identified that in line 1180, parentheses are still missing, which can lead to an incorrect calculation of the leftover amount and potentially exceeding the maxPayout. To fully resolve the issue, the missing parentheses in line 1180 need to be added to ensure the proper calculation. By making this correction, the potential for exceeding the maxPayout will be eliminated. Resolution 2: The client has also fixed the missing parentheses issue in line 1180, further ensuring the accuracy of the calculation.

Issue	Contract does not adhere to the checks-effects-interactions pattern
Severity	**Medium**
Description	The contract currently has instances where the checks-effects-interactions pattern is not followed. It is important to adhere to this pattern to mitigate the risk of reentrancy attacks. The checks-effects-interactions pattern suggests executing all external calls at the end of the function, if the function flow allows for it. Failing to follow this pattern can expose the contract to vulnerabilities, particularly in scenarios where a token with a fund-receiving hook is used. In such cases, the claimRewardsFirstAddress function, as shown in the example below, is susceptible to reentrancy

	attacks, potentially allowing the firstAddress to drain the entire contract within a single transaction: *function claimRewardsFirstAddress() public {* *require(msg.sender == firstAddress);* *uint256 refRew = user_data[msg.sender].referralRewards;* *CAFToken.safeTransfer(msg.sender, refRew);* *user_data[msg.sender].referralRewards = 0;* *}* To address this vulnerability, it is crucial to refactor the function and apply the checks-effects-interactions pattern. By moving the external call to the end of the function, after all necessary state changes have been made, the contract can avoid the potential risks associated with reentrancy attacks.
Recommendations	It is recommended to adhere to the checks-effects-interactions pattern consistently throughout the entire contract. By following this pattern, the contract can enhance its security and mitigate the risk of reentrancy attacks. To adhere to the checks-effects-interactions pattern, ensure that all external calls, such as token transfers or interactions with other contracts, are executed at the end of the function, after all checks and state changes have been completed. This ensures that any potential vulnerabilities caused by reentrancy attacks are minimized. Review the contract code and identify any instances where the checks-effects-interactions pattern is not followed. Refactor those functions accordingly, moving the external calls to the end of the function, if the function flow permits it. This consistent adherence to the pattern will help improve the security and robustness of the contract.
Comments	Acknowledged. Its good to hear that the client is aware of the issue and has taken measures to address it by adding nonReentrant modifiers. Although using nonReentrant modifiers can be a functional

	solution, its important to note that it doesnt fully align with the checks-effects-interactions pattern. Ideally, the checks-effects-interactions pattern recommends deferring external calls to the end of the function to minimize the risk of reentrancy attacks. While nonReentrant modifiers can prevent reentrancy issues, they dont strictly adhere to the pattern as they allow external calls to be made within the function. However, if the client has thoroughly reviewed the code and is confident that the nonReentrant modifiers adequately protect against reentrancy attacks, it can be an acceptable solution. Its crucial to ensure that the modifiers are applied consistently to all functions where reentrancy could be a concern. Regular code audits and testing should still be conducted to verify the effectiveness of the implemented solution.

Issue	Inconsistent referrer logic
Severity	**Low**
Description	The contract contains different logics for checking whether a referrer should receive rewards based on the combination of their claimed rewards, available rewards, and potential referral rewards compared to the maxPayout. The first logic simply checks if the sum of available rewards and claimed rewards is below the maxPayout, and if so, it grants referral rewards to the referrer. However, it doesnt consider the potential referral rewards, which could result in the referrer exceeding the maxPayout. The second logic, after addressing previous flaws, appears to be more comprehensive. It checks if the sum of claimed rewards, referral rewards, and newly received rewards exceeds the maxPayout. If it does, it increases the referralRewards by the difference needed to stay within the maxPayout limit. Otherwise, it increases the

	referralRewards by the maximum amount possible without surpassing the limit. From our perspective, the second logic (after fixing the identified issues) seems clearer and more appropriate. However, the final decision on which logic to implement ultimately rests with the developer.
Recommendations	It is recommended to use the second logic consistently throughout the contract. By applying the second logic consistently, you ensure a unified and clear approach to determining the referrers rewards based on the maxPayout limit. Using the second logic provides a comprehensive and reliable method for handling the calculation of referral rewards. It takes into account the claimed rewards, available rewards, and potential referral rewards, ensuring that the referrer does not exceed the maxPayout while still receiving the appropriate rewards. Consistency in implementing the second logic will improve the overall clarity and maintainability of the contract. It also reduces the risk of discrepancies or confusion caused by using different logics in different parts of the code. Therefore, it is advisable to update the contract to use the second logic consistently for determining referral rewards based on the maxPayout limit.
Comments	Acknowledged. The client has chosen to implement the second logic consistently throughout the contract for determining referral rewards based on the maxPayout limit. However, a bug was introduced in line 1180, which could potentially allow the referrer to exceed the maxPayout. To resolve this issue, it is recommended to add parentheses around the vulnerable area in line 1180, following the pattern used in other referral sections of the contract. Resolution 2 entails addressing the issue in line 1180 by adding the necessary parentheses to ensure the correct calculation and prevent the referrer from exceeding the maxPayout.

	By implementing this resolution, the client ensures that the referral rewards are calculated accurately and in line with the desired maxPayout limit, enhancing the security and correctness of the contract.

Issue	startTime can only be decreased
Severity	**Low**
Description	Currently, the changeStartTime function in the contract only allows for a decrease in the startTime value. However, it could be beneficial for the developer to consider allowing the option to shift the startTime to a future date as well. By enabling the ability to shift the startTime to a future date, the developer provides more flexibility in adjusting the contracts schedule. This can be useful in scenarios where it is necessary to postpone the start of a contract or align it with specific events or conditions. Introducing this capability would require modifying the changeStartTime function to include appropriate validation checks to ensure the new startTime value is valid and aligns with the contracts intended functionality and requirements. Ultimately, the decision to allow shifting the startTime to the future lies with the developer, who should carefully consider the implications and requirements of the contract before implementing this change.
Recommendations	*Consider the request to allow shifting the startTime to a future date, it is recommended to add the necessary logic to enable this functionality in the changeStartTime function.* *Heres a modified version of the function that includes the ability to shift the startTime to a future date:*

	function changeStartTime(uint256 newStartTime) external onlyOwner *{* *require(newStartTime > block.timestamp, "New start time must be in the future");* *startTime = newStartTime;* *emit StartTimeChanged(newStartTime);* *}* *With this modification, the function now checks if the provided newStartTime is greater than the current block timestamp, indicating a future date. If the condition is met, the startTime is updated with the new value.* *By implementing this logic, the contract allows for the adjustment of the startTime to a future date, providing the developer with the flexibility to shift the contracts start time as needed.* *Please note that this is a suggested implementation, and the developer should review and adapt it to fit the specific requirements and structure of the contract.*
Comments	Client has decided to remove the changeStartTime function entirely, it means they no longer require the ability to adjust the start time of the contract. The function can be safely removed from the contract code. By removing the changeStartTime function, any references to it in the contract should also be removed to ensure there are no potential issues or unused code. Please ensure that you update the contract code accordingly to remove the changeStartTime function and any associated references to it.

Issue	Gas optimization
Severity	**Informational**
Description	The following optimizations can be implemented in the contract to reduce gas consumption: 1) Variables that never change can be marked as constant to save gas by directly including their values in the bytecode. Specifically, the variables maxPayout, DEPOSITE_TAX_RATE, and MIN_DEPOSIT_AMOUNT can be marked as constant if they are not modified after initialization. 2) In view functions where variables are only used as return values, caching those variables in memory can save gas. For example, in the line address[] storage array = level7[userAddresss].referred;, the array variable can be cached in memory instead of storage. This optimization can be applied to all the level view functions. By implementing these optimizations, the contract can reduce gas consumption and improve overall efficiency. Please ensure that you update the contract code accordingly to apply these optimizations.
Recommendations	To optimize gas consumption and improve efficiency, please consider following the recommendations provided: 1) Mark the variables maxPayout, DEPOSITE_TAX_RATE, and MIN_DEPOSIT_AMOUNT as constant if they never change after initialization. This will allow their values to be directly included in the bytecode, reducing gas costs. 2) In view functions where variables are used only as return values, cache those variables in memory instead of storage. For example, in the line address[] storage array = level7[userAddresss].referred;, cache the array variable in memory. By implementing these recommendations, you can achieve a more gas-efficient and optimized contract.

Comments	Resolved

Issue	Typographical errors
Severity	**Informational**
Description	1) Remove the unused SafeMath library and update all operations to use the updated syntax. 2) Consider removing the unused variable BASE_WITHDRAW_FEE. 3) Remove the unused variable userAddress from all level structs. 4) Utilize or remove the unused events in the contract. 5) Remove the redundant assignment in the line that sets firstAddress. 6) Declare the changeStartTime function as external since it is only externally callable. 7) Declare the deposit function as external since it is only externally callable. 8) Remove the redundant check for a valid referrer address. 9) Remove redundant allowance check in the deposit and doubleUp functions. 10) Fix the parameter order in the ReferralRewarded event and emit it only when a reward is granted. 11) Update loop condition to iterate only for the correct number of referrers and remove redundant condition. 12) Declare the getMaxPayout function as external since it is only

	externally callable. 13) Declare the claimRewardsFirstAddress function as external since it is only externally callable. 14) Declare the getDay function as external since it is only externally callable. 15) Declare the claimRewards function as external since it is only externally callable. 16) Update the check for available interest in the claimRewards function to ensure it is greater than zero. 17) Declare the doubleUp function as external since it is only externally callable. By addressing these issues, the contract will be improved in terms of readability, efficiency, and adherence to best practices.
Recommendations	Implement above recommendations
Comments	Resolved. Most of the identified issues have been addressed, and the logic for firstAddress has been removed. It is recommended to remove any remaining instances where firstAddress is used or mentioned since it no longer serves any purpose in the contract and can be considered a dead variable. This cleanup will help improve code clarity and eliminate unnecessary references to unused variables.