SarosFarm

Smart Contract Audit Report
Prepared for SarosFinance

Date Issued:	Jun 15, 2022
Project ID:	AUDIT2022030
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022030
Version	v1.0
Client	SarosFinance
Project	SarosFarm
Auditor(s)	Peeraphut Punsuwan Ronnachai chaipha
Author(s)	Ronnachai chaipha
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Jun 15, 2022	Full report	Ronnachai chaipha

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by SarosFinance, Inspex team conducted an audit to verify the security posture of the SarosFarm smart contracts between May 10, 2022 and May 12, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of SarosFarm smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 3 high, 1 medium, 2 low, and 1 very low-severity issues. With the project team's prompt response, 3 high, 1 medium, 1 low, and 1 very-low-severity issues were resolved or mitigated in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that SarosFarm smart contracts have sufficient protections to be safe for public use. However, as the source code is not publicly available, the bytecode of the smart contracts deployed should be compared with the bytecode of the smart contracts audited before interacting with them. In the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Saros Finance is a Defi super network built on Solana. With three fundamental building blocks-SarosSwap (AMM), SarosFarm (Pool), and SarosPad (LaunchPad), Saros aims to attract more builders and users to the Solana Ecosystem.

SarosFarm is a program that allows platform users to stake tokens and earn rewards. The reward tokens will be stored in the reward pool and transferred to the users when they decide to claim the rewards.

Scope Information:

Project Name	SarosFarm
Website	https://saros.finance/swap
Smart Contract Type	Solana Program
Chain	Solana
Programming Language	Rust
Category	Yield Farming

Audit Information:

Audit Method	Whitebox
Audit Date	May 10, 2022 - May 12, 2022
Reassessment Date	May 20, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit:

Program	Bytecode SHA256 Hash
saros_farm	8b50480c4da9d149f55a5aeb813ac0d43b60237a3985c0a30ebbade2d295e1be

Reassessment:

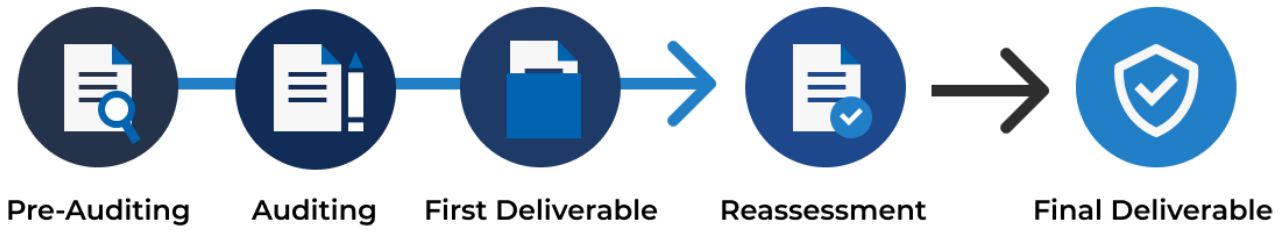
Program	Bytecode SHA256 Hash
saros_farm	6afb89388d6ec2edfd4d0da0362ebef5ed63dbb2aa4ebaf08a5eacb072c71029

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

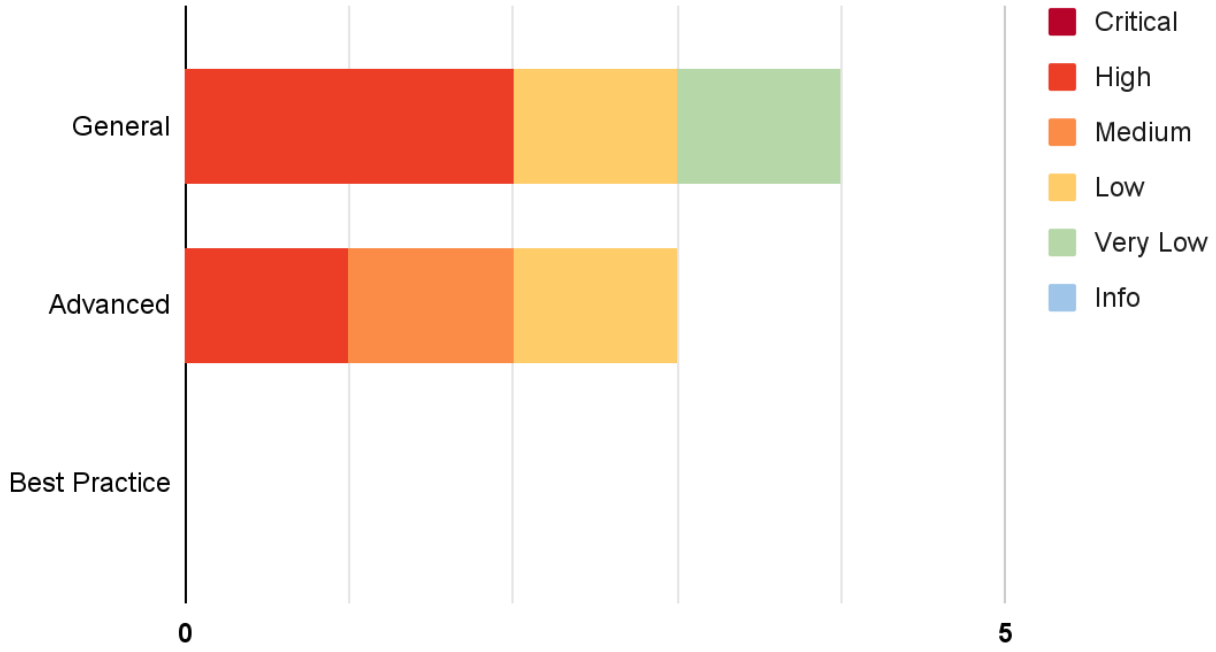
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

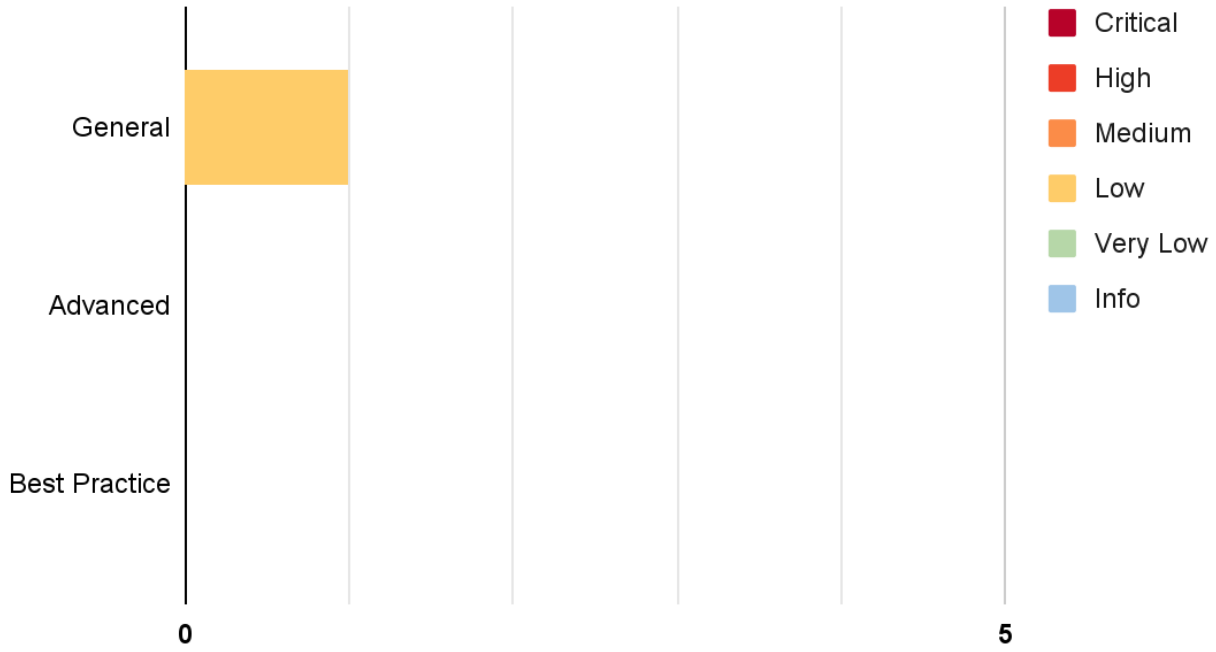
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Token Draining by the withdraw_token() Function	Advanced	High	Resolved *
IDX-002	Upgradability of Solana Program	General	High	Resolved *
IDX-003	Centralized Control of State Variable	General	High	Resolved *
IDX-004	Reward Miscalculation (reward_per_block)	Advanced	Medium	Resolved
IDX-005	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-006	Improper Reward Amount Verification	Advanced	Low	Resolved
IDX-007	Insufficient Logging for Privileged Functions	General	Very Low	Resolved

* The mitigations or clarifications by SarosFinance can be found in Chapter 5.

5. Detailed Findings Information

5.1. Token Draining by the withdraw_token() Function

ID	IDX-001
Target	saros_farm
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The token in the pool and pool reward account can be drained by the platform owner. **Likelihood:** Medium The only platform owner can execute the `withdraw_token()` function. However, the platform owner has a lot of motives to carry out this attack.
Status	Resolved * SarosFinance team has resolved the issue by allowing the platform owner to only withdraw the reward token and can withdraw after the end of the pool's farm period. However, if the platform owner withdraws the reward token exceeds the amount that must be reserved for users who have not yet claimed, users will not be able to claim reward tokens immediately.

5.1.1. Description

The `withdraw_token()` function can be used to withdraw staking and reward tokens from the pool and pool reward account as shown below:

lib.rs

```
309 #[access_control(utils::verify_root(*ctx.accounts.root.key))]
310 pub fn withdraw_token(
311   ctx: Context<WithdrawTokenContext>,
312   amount: u64,
313   authority_nonce: u8,
314 ) -> Result<()> {
315   msg!("SarosFarm: Withdraw Token");
316
317   let base_account = &mut ctx.accounts.base_account;
318   let authority = &ctx.accounts.authority;
319   let from = &ctx.accounts.from;
320   let to = &ctx.accounts.to;
```

```
321
322   let seeds: &[&[_]] = &[
323     &constants::AUTHORITY_SEED[..],
324     base_account.to_account_info().key.as_ref(),
325     &[authority_nonce]
326   ];
327
328   utils::transfer_token(
329     authority,
330     from,
331     to,
332     amount,
333     &[&seeds]
334   )?;
335
336   Ok(())
337 }
```

As a result, there are insufficient rewards to distribute to users, and the user's capital (staking token) is lost when the platform owner uses the `withdraw_token()` function to withdraw those tokens.

5.1.2. Remediation

Inspex suggests that the platform owner should limit the usage of the `withdraw_token()` function as follow:

- The platform owner should be able to withdraw the reward token in the pool rewards account only. This means the platform owner must not be able to withdraw the deposit tokens from the users.

lib.rs

```
309 #[access_control(utils::verify_root(*ctx.accounts.root.key))]
310 pub fn withdraw_token(
311   ctx: Context<WithdrawTokenContext>,
312   amount: u64,
313   authority_nonce: u8,
314 ) -> Result<()> {
315   msg!("SarosFarm: Withdraw Token");
316
317   let base_account = &mut ctx.accounts.base_account;
318   let authority = &ctx.accounts.authority;
319   let from = &ctx.accounts.from;
320   let to = &ctx.accounts.to;
321
322   let current_block = Clock::get().unwrap().slot;
323   if current_block <= base_account.reward_end_block {
324     return Err(ErrorCode::CantWithdrawNow.into());
325   }
326
```

```
327   let seeds: &[&[_]] = &[
328     &constants::AUTHORITY_SEED[..],
329     base_account.to_account_info().key.as_ref(),
330     &[authority_nonce]
331   ];
332
333   utils::transfer_token(
334     authority,
335     from,
336     to,
337     amount,
338     &[&seeds]
339   )?;
340
341   Ok(())
342 }
343

    ...
345
689 #[derive(Accounts)]
690 pub struct WithdrawTokenContext<'info> {
691
692   /// CHECK: program owner, verified using #access_control
693   #[account(signer)]
694   pub root: AccountInfo<'info>,
695
696   /// CHECK: Pool or PoolReward address
697   pub base_account: Account<'info, PoolReward>,
698
699   /// CHECK: PDA to hold pool's assets
700   pub authority: AccountInfo<'info>,
701
702   /// CHECK: Pool token account
703   #[account(mut)]
704   pub from: AccountInfo<'info>,
705
706   /// CHECK: User token account
707   #[account(mut)]
708   pub to: AccountInfo<'info>,
709
710   /// CHECK: Solana native Token Program
711   #[account(
712     constraint = utils::is_token_program(&token_program)
713   )]
714   pub token_program: AccountInfo<'info>,
715 }
```

- During the pool is active, when withdrawing the reward token, the platform owner should ensure that the available reserve amount is enough for all the users who have not claimed their rewards to prevent the case that there is not enough reward to distribute.

5.2. Upgradability of Solana Program

ID	IDX-002
Target	saros_farm
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of the affected programs can be arbitrarily changed. This allows the upgrade authority to change the logic of the program in favor of the platform, e.g., transferring the users' funds to the platform owner's account. **Likelihood: Medium** Only the program upgrade authority can redeploy the program to the same program address; however, there is no restriction to prevent the authority from inserting malicious logic.
Status	Resolved * SarosFinance team has confirmed that they will use multisig as an upgrade authority. This will be controlled by multiple trusted parties to ensure the transparency of the platform. However, the contract has not been deployed yet the users should be sure that the parties are trusted before using the platform.

5.2.1. Description

Programs on Solana can be deployed through the upgradable BPF loader to make them upgradable, allowing the program's upgrade authority to redeploy the program with the new logic, bug fixes, or upgrades to the same program address.

However, there is no restriction on how and when the program will be upgraded. This opens up an attack surface on the program, allowing the upgrade authority to redeploy the program with malicious logic and gain unfair benefits from the users, for example, transferring funds out from the users' accounts.

5.2.2. Remediation

Inspex suggests deploying the program as an immutable program to prevent the program logic from being modified.

However, if the upgradability is needed, Inspex suggests mitigating this issue by the following options:

- Using a multisig account controlled by multiple trusted parties as the upgrade authority
- Implementing a community-run governance to control the redeployment of the program

5.3. Centralized Control of State Variable

ID	IDX-003
Target	saros_farm
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood: Medium** There is nothing to restrict the changes from being done by the owner. However, only some owner roles can call these functions to change the states.
Status	Resolved * SarosFinance team has confirmed that they will use multisig as an administrative account. This will be controlled by multiple trusted parties to ensure that the critical changes will be publicly known first before it has an effect. However, the contract has not been deployed yet the users should be sure that the parties are trusted before using the platform.

5.3.1. Description

Critical state variables can be updated anytime by the controlling authorities. Changes in these variables can impact the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Program	Access Control	Function
programs/saros-farm/src/lib.rs (L:295)	saros_farm	verify_root	update_reward_per_block()
programs/saros-farm/src/lib.rs (L:309)	saros_farm	verify_root	withdraw_token()

5.3.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the program. However, if modifications are needed, Inspex suggests limiting the use of these functions by the following options:

- Using a multisig account controlled by multiple trusted parties to ensure that the changes of critical states are well prepared
- Implementing a community-run governance to control the use of these functions

However, if the `set_pause_pool()` and `set_pause_reward_pool()` functions are necessary for the emergency case, Inspex suggests changing the access control of the `set_pause_pool()` and `set_pause_reward_pool()` functions from `verify_root` to another access control.

5.4. Reward Miscalculation (reward_per_block)

ID	IDX-004
Target	saros_farm
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The reward miscalculation can lead to unfair reward token distribution, which may cause a loss of reputation. **Likelihood:** Medium The rewards miscalculation can occur after the `reward_per_block` has been updated. However, only the owner can update the `reward_per_block` state.
Status	Resolved SarosFinance team has resolved this issue as suggested.

5.4.1. Description

The `reward_per_block` variable is used to determine the total amount of reward tokens to be transferred as a reward per block, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `reward_per_block` variable is modified without updating the pending rewards first, the reward of a pool will be incorrectly calculated.

In the `update_reward_per_block()` function shown below, the `reward_per_block` variable is modified without updating the rewards.

lib.rs

```
294 #[access_control(utils::verify_root(*ctx.accounts.root.key))]
295 pub fn update_reward_per_block(
296   ctx: Context<UpdateRewardPerBlockContext>,
297   reward_per_block: u128,
298 ) -> Result<()> {
299   msg!("SarosFarm: Update RewardPerBlock");
300
301   let pool = &mut ctx.accounts.pool;
302
303   pool.reward_per_block = reward_per_block;
304
305   Ok(())
306 }
```

For example:
Assuming that at block 1000, the `reward_per_block` is set to 10 tokens per block, pool `total_shares` is 100.

Block	Action
1000	pools' `last_updated_block` are updated
2000	The `reward_per_block` is updated to 20 tokens per block using the `update_reward_per_block()` function.
3000	pools' `last_updated_block` are updated once again

The total rewards distributed from block 1000 to block 3000 equal 20 tokens per block, from block 1000 to block 3000 (20 x (3000 - 1000) = 40000 tokens).

However, rewards should be calculated by accounting for the original `reward_per_block` value during the period when it is not yet updated as follows:

- 10 tokens per block, from block 1000 to block 2000 (10 * (2000 - 1000) = 10000 tokens)
- 20 tokens per block, from block 2000 to block 3000 (20 * (3000 - 2000) = 20000 tokens)
- Total tokens distributed (10000 + 20000 = 30000 tokens)

5.4.2. Remediation

Inspex suggests adding the `update_pool_reward()` function and executing it before updating the `reward_per_block` variable as shown in the following example:

lib.rs

```
294 #[access_control(utils::verify_root(*ctx.accounts.root.key))]
295 pub fn update_reward_per_block(
296   ctx: Context<UpdateRewardPerBlockContext>,
297   reward_per_block: u128,
298 ) -> Result<()> {
299   msg!("SarosFarm: Update RewardPerBlock");
300
301   let pool = &mut ctx.accounts.pool;
302   pool.update_pool_reward();
303   pool.reward_per_block = reward_per_block;
304
305   Ok(())
306 }
```

5.5. Smart Contract with Unpublished Source Code

ID	IDX-005
Target	saros_farm
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	**Severity:** Low **Impact:** Medium The logic of the smart contract may not align with the user's understanding, causing undesired actions to be taken when the user interacts with the smart contract. **Likelihood:** Low The possibility for the users to misunderstand the functionalities of the contract is not very high with the help of the documentation and user interface.
Status	**Acknowledged** SarosFinance team has acknowledged this issue and decided not to publish the source code because the team wants to protect their intellectual property.

5.5.1. Description

The smart contract source code is not publicly published, so the users will not be able to easily verify the correctness of the functionalities and the logic of the smart contract by themselves. Therefore, it is possible that the user's understanding of the smart contract does not align with the actual implementation, leading to undesired actions on interacting with the smart contract.

5.5.2. Recommendation

Inspex suggests publishing the contract source code through a public code repository or verifying the smart contract source code on the blockchain explorer so that the users can easily read and understand the logic of the smart contract by themselves.

5.6. Improper Reward Amount Verification

ID	IDX-006
Target	saros_farm
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The users can not withdraw the rewards token until the rewards are enough to claim. However, the users are still withdrawing their deposit tokens normally. **Likelihood:** Low Users can check the amount of reward tokens in the rewards pool first before they start depositing. However, it is unlikely that the reward amount will be insufficient for claiming since this would cause damage to the platform's reputation.
Status	Resolved SarosFinance team has resolved this issue as suggested by depositing reward tokens while calling the `create_pool_reward()` and `update_pool_reward_params()` functions.

5.6.1. Description

In the `saros_farm` program, it allows the users to stake the token with the `stake_pool()` and `stake_pool_reward()` functions respectively. After that, the users will be able to claim the reward token through the `claim_reward()` function.

During the `claim_reward()` function has been called, the `saros_farm` program will calculate the reward for the number of times users have staked, with the `reward_per_block` variable stored in the pool reward account.

Then, the reward token account (`pool_reward_token_account`) will transfer the token with the calculated amount (`pending_reward`) to the user's token account as in line 244.

lib.rs

```
222 pub fn claim_reward(
223   ctx: Context<ClaimRewardContext>
224 ) -> Result<()> {
225   msg!("SarosFarm: Claim Reward Instruction");
226
227   let pool_reward_authority = &ctx.accounts.pool_reward_authority;
228   let pool_reward_token_account = &ctx.accounts.pool_reward_token_account;
```

```
229   let user_reward_token_account = &ctx.accounts.user_reward_token_account;
230
231   let pool_reward = &mut ctx.accounts.pool_reward;
232   let user_pool_reward = &mut ctx.accounts.user_pool_reward;
233
234   pool_reward.update_pool_reward();
235
236   let pending_reward = pool_reward.calculate_reward(user_pool_reward.amount,
      user_pool_reward.reward_debt, user_pool_reward.reward_pending);
237   if pending_reward > 0 {
238     let seeds: &[&[_]] = &[
239       &constants::AUTHORITY_SEED[..],
240       pool_reward.to_account_info().key.as_ref(),
241       &[pool_reward.authority_nonce]
242     ];
243
244     utils::transfer_token(
245       pool_reward_authority,
246       pool_reward_token_account,
247       user_reward_token_account,
248       pending_reward,
249       &[&seeds]
250     )?;
251   }
```

At the same time, the `saros_farm` program doesn't check whether the amount of reward token stored in the reward token account (`pool_reward_token_account`) is enough for this user or not. Therefore, when the reward is insufficient, the user will not be able to claim the reward immediately until the platform owner transfers the reward into the reward token account.

5.6.2. Remediation

Inspex suggests adding a mechanism to ensure that the stored reward tokens in the reward token account (`reward_token_account`) is sufficient for distributing to all users until the pool is expired (end of the `reward_end_block` state). This can be achieved by adding a condition in the `create_pool_reward()` function as shown below:

lib.rs

```
38  #[access_control(utils::verify_root(*ctx.accounts.root.key))]
39  pub fn create_pool_reward(
40    ctx: Context<CreatePoolRewardContext>,
41    pool_reward_nonce: u8,
42    pool_reward_authority_nonce: u8,
43    reward_token_mint: Pubkey,
44    reward_per_block: u128,
45    reward_start_block: u64,
```

```
46    reward_end_block: u64,
47  ) -> Result<()> {
48    msg!("SarosFarm: Create Pool Reward Instruction");
49
50    if reward_start_block > reward_end_block {
51      return Err(ErrorCode::TimeOverlap.into());
52    }
53
54    let pool_reward = &mut ctx.accounts.pool_reward;
55    let root = &mut ctx.accounts.root;
56    let root_reward_token_account = &mut ctx.accounts.root_reward_token_account;
57    let reward_token_account = &mut ctx.accounts.reward_token_account;
58
59    let current_block = Clock::get().unwrap().slot;
60
61    let (pool_reward_authority, _): (Pubkey, u8) =
   utils::get_authority_account(pool_reward.to_account_info().key,
62  ctx.program_id);
      let reward_token_account_key =
63  utils::get_associated_token_address(&pool_reward_authority,
64  &reward_token_mint);

65    require!(reward_token_account.key() != reward_token_account_key,
66  ErrorCode::InvalidPoolRewardTokenAccount);
67
68    pool_reward.nonce = pool_reward_nonce;
69    pool_reward.authority_nonce = pool_reward_authority_nonce;
70    pool_reward.reward_token_mint = reward_token_mint;
71    pool_reward.reward_token_account = reward_token_account.key();
72    pool_reward.reward_per_block = reward_per_block;
73    pool_reward.reward_end_block = reward_end_block;
74    pool_reward.total_shares = 0;
      pool_reward.accumulated_reward_per_share = 0;
75    pool_reward.last_updated_block = std::cmp::max(reward_start_block,
76  current_block);
77    pool_reward.state = constants::PoolState::Unpaused;

78    let total =
79  u128::from(reward_end_block.checked_sub(pool_reward.last_updated_block).unwrap(
80  )).checked_mul(reward_per_block).unwrap() / 1_000;
81    let u64_max = u128::from(u64::MAX);
82    let amount = if total > u64_max {
83      u64::MAX
84    }
85    else {
86      u64::try_from(total).unwrap()
87    };
```

```
88
89    utils::transfer_token(
90      root,
91      root_reward_token_account,
92      reward_token_account,
93      amount,
94      &[]
95    )?;
96
97    Ok(())
98  }
99
    ...

361 #[derive(Accounts)]
362 #[instruction(pool_reward_nonce: u8, _pool_reward_authority_nonce: u8,
363 reward_token_mint: Pubkey)]
364 pub struct CreatePoolRewardContext<'info> {
365
366   /// CHECK: program owner, verified using #access_control
367   #[account(signer, mut)]
368   pub root: AccountInfo<'info>,
369
370   pub pool: Account<'info, Pool>,
371
372   #[account(
373     init,
374     seeds = [
375       pool.to_account_info().key.as_ref(),
376       reward_token_mint.key().as_ref(),
377     ],
378     bump,
379     payer = root,
380     space =  8 + PoolReward::LEN
381   )]
382   pub pool_reward: Account<'info, PoolReward>,
383
384   #[account(mut)]
385   pub root_reward_token_account: AccountInfo<'info>,
386
387   #[account(mut)]
388   pub reward_token_account: AccountInfo<'info>,
389
390   pub system_program: Program<'info, System>,
391 }
```

Please note that the deposit reward calculation in the `create_pool_reward()` function does not support

the case when the platform owner changes the `reward_per_block` through the `update_reward_per_block()` function. This also includes withdrawing the reward tokens through the `withdraw_token()` function. One of these concerns will invalidate the deposit reward calculation. This is due to the remediation of `IDX-003: Centralized Control of State Variables` since we ideally suggest not changing nor executing these privilege functions.

However, if the `update_reward_per_block()` function is necessary, we suggest adding a checking mechanism and re-calculate the needed reward amount again.

5.7. Insufficient Logging for Privileged Functions

ID	IDX-007
Target	saros_farm
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is unlikely that the execution of the privileged functions will be a malicious action.
Status	Resolved SarosFinance team has resolved this issue as suggested.

5.7.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

The privileged functions with insufficient logging are as follows:

File	Program	Function
programs/saros-farm/src/lib.rs (L:76)	saros_farm	set_pause_pool()
programs/saros-farm/src/lib.rs (L:93)	saros_farm	set_pause_reward_pool()
programs/saros-farm/src/lib.rs (L:295)	saros_farm	update_reward_per_block()
programs/saros-farm/src/lib.rs (L:309)	saros_farm	withdraw_token()

5.7.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

lib.rs

```
75  #[event]
76  pub struct SetPausePoolEvent {
77      pub state: bool,
78  }
79
80  #[access_control(utils::verify_root(*ctx.accounts.root.key))]
81  pub fn set_pause_pool(
82    ctx: Context<SetPausePoolContext>,
83    is_pause: bool
84  ) -> Result<()> {
85    msg!("SarosFarm: Set Pause Pool Instruction");
86    let pool = &mut ctx.accounts.pool;
87
88    if is_pause {
89      pool.state = constants::PoolState::Paused;
90    } else {
91      pool.state = constants::PoolState::Unpaused;
92    }
93
94    emit!(SetPausePoolEvent{
95      state: is_pause,
96    });
97
98    Ok(())
99  }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspeX

CYBERSECURITY PROFESSIONAL SERVICE