HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Soilfarm
Date: 03 August, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Soilfarm
Approved By	Noah Jelich | Lead Solidity SC Auditor at Hacken OU
Tags	Fungible Token; Vesting; Staking; Yield Farming; Centralization; Signatures
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://soil.co/
Changelog	02.06.2023 - Initial Review 03.07.2023 - Second Review 03.08.2023 - Third Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by Soilfarm (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Soilfarm is a DeFi system aiming to provide users the ability to lend money to selected traditional funds, then yield SOIL tokens and USDC. Apart from that users will be able to buy SOIL tokens and stake them to receive more. To be able to join a pool or stake, users must complete KYC approval from the backend app. Then, for every action backend signature must be obtained. Smart contracts are not responsible for calculating the rewards.

The files in the scope:

- *SoilToken.sol* – ERC-20 token that features burn and snapshot functionalities. All initial supplies are minted to predefined addresses during the construction phase for initial holders. It does not permit any additional minting.
- *Staking.sol* – the Staking contract is a smart contract designed for staking and unstaking SOIL tokens, as well as claiming rewards and performing administrative functions. The contract allows users to stake their SOIL tokens, earn rewards, and withdraw their staked tokens along with the rewards. For every non-administrative function, the user has to provide a signature as a parameter in the function. The signature is generated in the backend, and smart contract checks if the provided signature is signed by the backend signer.
- *Vesting.sol* – token holder contract that releases tokens based on parameters provided by the admin that created it (the address for which vesting is created, amount of locked tokens, unix timestamp from which tokens are releasing, time between release intervals, amount of tokens released per interval, boolean indicating whether withdrawing on demand by admin is enabled)
- *PoolsContract.sol* – the pools contract is designed to allow users to gain rewards in USDC tokens and potentially in SOIL tokens by depositing to the pool. Pools are updatetable for the admin role. In order to withdraw from the pool users must wait a specified amount of time by the admin, unless early withdrawals are enabled. Just like in staking contracts, users have to provide signatures generated by the backend server as the parameter in the function call.
- *ProtocolSettings.sol* – provides an access control system to the protocol.
- *VerifySignatureSystem.sol* – provides signature verification to the protocol.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Privileged roles

- DEFAULT_ADMIN_ROLE: super admin that can grant and revoke roles. This role can call withdrawFunds function and withdraw all deposited funds by users in the Pools Contract.
- ADMIN_ROLE: this role is able to perform any action with an onlyAdmin modifier (creating pools, updating pools, withdraw collected fees from pools, update backend signer, disable and enable staking, unlock vested tokens in vesting on demand, send rewards to pool and staking contract).
- TokenSpender: address that can create vaults in vesting and from which vested tokens come from.
- Backend Signer: the address which the signatures provided by the user should come from.
- Rewards Holder Wallet: address that holds the rewards for Pools and Staking users.
- Soil Token Owner: The only role that can update a list of snapshooters.
- Snapshooter: Has the permission to make a snapshot on soil token.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements have some gaps:
  - Project overview is provided with roles and use cases.
  - Tokenomics is provided.
- Technical description is robust:
  - Dev env instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- Solidity Style Guide violations.
- Project contains redundant functionality.

Test coverage

Code coverage of the project is **97.71%** (branch coverage), with a mutation score of **53.31%.**

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is partially missing.
- Interactions with several users are tested thoroughly.
- Low test mutation score indicates the code logic is not validated well by tests, and that the tests would not catch code changes.

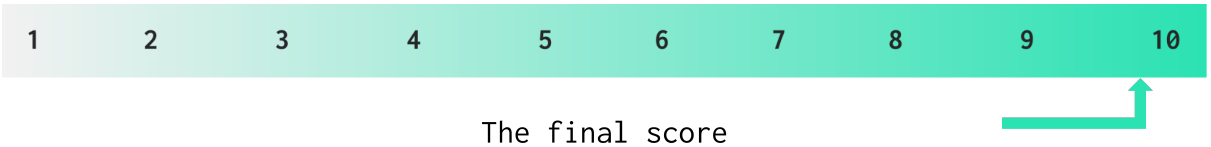
Security score

As a result of the audit, the code contains **no** issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.9**.The system users should acknowledge all the risks summed up in the risks section of the report.



The final score

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
02 June 2023	3	8	9	1
03 July 2023	1	0	4	1
03 August 2023	0	0	0	0

Risks

- The contract design provides administrators with a significant degree of control over contract operation. **If the keys of these admin accounts are compromised, the operation of the contract, including user funds and important parameters, could be affected**.
- The project's contract design shows a **high degree of centralization**. This centralization means that control over key operations and parameters is vested in a few accounts or roles. The risks associated with such centralization include **potential misuse of power, single points of failure.**
- Part of the project's functionality, particularly the **creation of signatures and rewards calculation, is handled off-chain** and was not part of the audit. As a result, the security of these off-chain components cannot be verified.
- The contract architecture currently **lacks a reentrancy guard** for operations that interact with tokens. While this might not pose a direct threat with the current tokens supported, it could potentially be a **security concern if the contract is updated to support other token standards, such as ERC777**.
- The security of the funds in the system depends on the **out-of-scope legal process**.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Not Relevant	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Passed	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Not Relevant	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Passed	
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

■■■■ Critical

C01. Signed Message Replay Attack; Irrevocable Signed Message

Impact	High
Likelihood	High

The protocol's pool and staking contracts, which inherit from VerifySignatureSystem, are not utilizing EIP712 for signatures and deadlines. The current implementation of signed messages lacks mechanisms to prevent replay attacks and does not offer a mechanism for message expiration.

Without a standard such as EIP712, signed messages can potentially be reused on other chains (replay attacks).

This can lead to unauthorized execution of smart contract functions, manipulation of the contract's state and data, or compromise of the integrity and security of the contract's operations.

The signature revoke issue arises when a contract lacks functionality that allows revoking a signed message.

This can result in the execution of a signed transaction when it is not desired by a signer.

Paths: ./contracts/Staking.sol : claimRewards(), stakeSOIL(), restakeRewards(), unstakeSOIL()

./contracts/PoolsContract.sol : deposit(), withdraw(), claimRewards()

./contracts/VerifySignatureSystem.sol : verifySignature()

Recommendation: implement EIP712 for signatures in the protocol. EIP712 offers a standard way to structure data and generate signatures, which would significantly enhance the security of the contract by protecting it from replay attacks. Also, consider adding an expiration timestamp (deadline) to the signed messages to ensure that they cannot be used indefinitely, further increasing the robustness of the protocol.

Found in: a3d764b

Status: Fixed (Revised commit: 7ec15f8)

C02. Funds Lock; Data Consistency;

Impact	High
Likelihood	High

The unstakeSOIL function of the Staking.sol contract includes an erroneous double deduction of the amountOfRewards variable. This

leads to a significant over-decrement of reward tokens, which in turn, could potentially lock the incorrectly deducted rewards within the contract.

If a user unstakes and the reward amount is deducted twice from amountOfRewards, the remaining balance of reward tokens in the contract may not be accurate. This could lead to a situation where the "double-spent" rewards remain locked within the contract, leaving them inaccessible to users who should be entitled to claim them. This might cause significant disruption to the contract's operation.

Paths: ./contracts/Staking.sol : unstakeSOIL()

Recommendation: revise the unstakeSOIL function to ensure that the rewards are deducted from amountOfRewards only once during each unstaking operation. This correction ensures the accurate tracking of reward token balances within the contract, preventing potential locking of reward tokens.

Found in: 7ec15f8

Status: Fixed (Revised commit: d71b55e)

■■■ High

H01. Highly Permissive Role Access

Impact	High
Likelihood	Medium

The function adminWithdraw grants the administrator the ability to withdraw up to 30% of SOIL tokens from the staking contract, excluding the collected fee. This capability does not update user amounts. Consequently, users who unstake after an admin withdrawal may not be able to retrieve their staked tokens fully or at all, depending on the sequence and timing of unstaking transactions. As all tokens (excluding fee) belong to users who staked them, this functionality is unclear.

Admin misuse of this function can lead to financial loss for users who may not be able to retrieve their staked tokens. The potential for loss could result in eroded trust and reputational damage to the platform.

Path: ./contracts/Staking.sol : adminWithdraw()

Recommendation: remove mentioned functionality, as its potential for misuse poses unnecessary risk. If administrative withdrawals are necessary, such functionality must transparently reflect and update user balances, ensuring users can retrieve their staked tokens fully.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

H02. Highly Permissive Role Access

Impact	**High**
Likelihood	Medium

The withdrawFunds function is designed to allow an admin role to withdraw all funds deposited by users. This functionality is highly sensitive as it deals with the transfer of assets that belong to the users of the contract. Having an admin role with this level of access can pose a significant risk if the account associated with the role is compromised.

If the account with admin privileges is compromised, malicious actors could drain the contract of its assets, causing significant financial damage to users and potentially irreparable harm to the trust and reputation of the project.

Path: ./contracts/PoolsContract.sol : withdrawFunds()

Recommendation: remove the withdrawFunds function. This approach reduces potential security risks and aligns with the principle of least privilege, considering users have already paid fees during contract interactions.

Found in: a3d764b

Status: Mitigated (the withdrawFunds function can only be executed by a Super Admin role, which is secured through a multi-signature. Furthermore, all user funds are safeguarded under a Loan Agreement. Each user signs this agreement with the Soil protocol, providing an additional layer of legal protection for the deposited assets.)

H03. Hidden Fees

Impact	**High**
Likelihood	Medium

The current system calculates staking, unstaking, and reward claim fees off-chain, which are then validated using a backend-generated signature. Users can potentially be charged multiple fees during a single stake operation, creating unpredictability in the overall cost of using the platform.

This design can lead to unexpectedly high cumulative fees for users, significantly reducing their earnings from staking and undermining the profitability of using the platform. This could harm the project's reputation and user trust.

Path: ./contracts/Staking.sol : stakeSOIL(), unstakeSOIL(), claimRewards()

Recommendation: provide a publicly available document detailing the fee system and off-chain aspects of the project to ensure transparency and trustworthiness. Implement on-chain fee limits

within the fee handling function to prevent excessive charges. Also, consider consolidating fees into a single charge either at the staking or withdrawal stage to enhance user-friendliness and cost predictability.

Found in: a3d764b

Status: Mitigated (The fee now applies only in the deposit function of the pool contract and stakeSOIL contract, providing a more predictable fee structure. According to the updated documentation for staking fees - “Fees in the Staking contract are taken in the Soil tokens. Administrator sets the amount of USDC, which will be taken from user worth of SOIL tokens. Every time a user wants to stake, backend checks the current USDC to SOIL exchange ratio and, based on that, calculates the amount of soil tokens taken from user as a fee.”)

H04. Data Consistency; Highly Permissive Role Access

Impact	High
Likelihood	Medium

The Staking contract does not record data related to the amount of tokens deposited by a user. This leads to a situation where it is possible to withdraw the entire staking balance to any address given a valid signature provided by the backend. As a result, this could potentially lead to loss of user funds.

Path: ./contracts/Staking.sol : stakeSOIL(), unstakeSOIL(), restakeRewards()

Recommendation: create a struct that holds user data. This data should be validated and updated with user interactions with the contract, such as staking, unstaking, and restaking rewards. This would provide an additional layer of security and help to prevent unauthorized withdrawals.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

H05. Funds Lock; Highly Permissive Role Access

Impact	High
Likelihood	Medium

The current design allows an admin to block or unblock any user with the toggleBlockUser function. If a user is blocked, they are unable to withdraw their vested tokens from all vaults assigned to them, potentially leading to an indefinite lock of their funds.

If the admin blocks a user, the user loses access to their vested tokens, leading to potential fund loss. Given that the contract lacks

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

a mechanism for forcibly withdrawing tokens from a blocked vault, this could mean irreversible loss of funds.

Path: ./contracts/Vesting.sol : toggleBlockUser()

Recommendation: a robust governance mechanism should be in place for admin actions. It could be beneficial to establish a well-documented policy outlining the reasons and scenarios for blocking users. In addition, the ability to block a user could be set at the vault creation level, so only specific vaults can be blocked, rather than an entire user account. This would offer more granularity and control over the blocking process, while still providing necessary administrative controls.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

H06. Hidden Fees

Impact	High
Likelihood	Medium

The current design of the pool contract applies fees at various transaction stages: deposit, withdrawal, reward claim, and upgrading to a pool with higher yield. These fees are predefined within the contract. The cumulative nature of these fees throughout multiple transactions can lead to an increased cost burden on the users.

The multiplicative fee structure may result in unexpectedly high total fees for the users. This can significantly reduce the profits they receive from participating in the pools, impacting the overall appeal of the pool system. The unpredictability of the total fees can potentially harm user trust and tarnish the project's reputation.

Path: ./contracts/PoolsContract.sol: deposit(), withdraw(), claimRewards(), upgradeToPoolWithHigherYield()

Recommendation: to enhance transparency and improve user experience, the consolidation of fees into a single charge is recommended. This could be implemented either at the deposit stage, withdrawal stage. Additionally, a fee can be defined for every specific pool and be settled during pool creation without possibility to update it or with max limit of fee for fee. Further, a comprehensive documentation detailing the fee structure and the rationale behind it could add value to the project.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

H07. Data Consistency; Fund Lock; Highly Permissive Role Access

Impact	High

Likelihood	Medium

The smart contract currently allows an admin to alter parameters of a pool, such as periods and early withdrawal settings, through the updatePool function. If these adjustments occur after users have deposited funds, it may lead to unintended outcomes:

- Fee Misapplication: If the period length is reduced, users who initially deposited with an expectation of a fixed lock-in duration could now face an early withdrawal fee.
- Fund Lock: If the isEarlyWithdrawalEnabled parameter is changed to false, users may be prevented from withdrawing their funds until the newly set period ends.

These unpredicted changes can lead to users losing funds due to unanticipated fees or being unable to access their deposits. Such situations could undermine trust in the platform and lead to potential financial losses for users.

Additionally, there is no check in the current signature for pool updates and deadlines, and as a result, any signature created for the pool before an update will be valid.

Path: ./contracts/PoolsContract.sol : updatePool()

Recommendation: disallow updates to a pool once it is open for deposits. This can be achieved by adding a condition in the updatePool function to check whether the pool's start time has already passed, as in require(currentPoolInfo.startTime > block.timestamp, "Pool has already started and cannot be updated"). This would prevent any changes to the pool that could negatively affect the users once they have started interacting with it.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

H08. Coarse-Grained Access Control

Impact	High
Likelihood	Medium

The staking, pools, and vesting contracts are all controlled by a single administrative role. This means that the same account has the ability to change critical parameters, conduct administrative withdrawals, and manage other sensitive operations across all these contracts. This level of access concentration can present significant risks if the admin account is compromised.

In the event that the admin account is compromised, an attacker could potentially exploit this administrative power across multiple contracts, leading to severe disruptions in operations, alteration of critical contract parameters, and potential loss of user funds across the entire platform. Furthermore, this level of access concentration

presents a single point of failure, further increasing the potential damage that could be inflicted by a malicious actor.

Paths: ./contracts/PoolsContract.sol

./contracts/Vesting.sol

./contracts/Staking.sol

Recommendation: consider adopting a more fine-grained access control strategy. By segregating roles and responsibilities, the potential damage from a compromised account can be significantly reduced. For example, having separate administrative roles for each contract (staking, pools, and vesting) can help limit the scope of actions that a single account can perform. Moreover, implementing multi-signature controls for sensitive operations can also provide an additional layer of security.

Found in: a3d764b

Status: Mitigated (All of the admins will be using multisig wallet)

H09. Coarse-Grained Access Control

Impact	**High**
Likelihood	Medium

The claimRewards function in both the Staking contract and the Pool contract uses an off-chain generated signature for authorization, which is verified by the verifySignature function. This function transfers the rewards from rewardsHolderWallet to the user. In the event that the backend server is compromised, the attacker can drain all rewards that rewardsHolderWallet has approved for the contract by creating valid signatures and claiming rewards.

A successful compromise of the backend server could lead to a total loss of all approved rewards in the rewardsHolderWallet. This could significantly undermine the profitability of using the platform and harm the project's reputation and user trust. It could also disrupt the normal operation of the staking and pooling contracts and lead to a potential halt in these functionalities.

Paths: ./contracts/PoolsContract.sol : _claimRewards()

./contracts/Staking.sol : claimRewards(), restakeRewards(), unstakeSOIL()

Recommendation: consider implementing a limit on the maximum amount of rewards that can be claimed in a single transaction. This would act as a safeguard by limiting the potential loss in the event of a backend compromise.

Additionally, consider a design where the staking and pool contracts hold a predefined amount of rewards. The contracts could have a

mechanism to add more rewards as needed, but any excess would be held outside of the contract. This would further limit potential losses.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

H10. Data Consistency

Impact	High
Likelihood	Medium

The 'claimRewards' function in the staking contract does not update the 'amountOfRewards' state variable, which could lead to discrepancies in reward tracking.

Since 'amountOfRewards' is a central variable to keep track of the available rewards in the contract, not updating it during the 'claimRewards' operation might result in inaccurate reward distribution. This can lead to scenarios where the contract thinks it has more rewards available than it actually does

Path: ./contracts/Staking.sol : claimRewards()

Recommendation: modify the 'claimRewards' function to decrement the 'amountOfRewards' state variable by the claimed amount. This would ensure that the contract's state accurately reflects the actual number of rewards available.

Found in: cd03b51

Status: Fixed (Revised commit: 7ec15f8)

Medium

M01. Highly Permissive Role Access

Impact	Medium
Likelihood	Medium

The function updateLockoutTime provides the ability to modify the lockout time for unstaking. This change is applied not only to future stakers but also to those who have already staked their tokens. Importantly, this function does not impose any restrictions on the newly assigned lockout time.

This lack of restriction allows for potential manipulation of the lockout period. Existing stakers could unexpectedly find themselves subject to extended lockout periods, restricting their ability to unstake their tokens as planned.

Path: ./contracts/Staking.sol : updateLockoutTime()

Recommendation: restrict the application of updateLockoutTime solely to future stakes and establish a maximum limit for the lockout time

period. This would provide protection for existing stakers and prevent potential misuse of the function.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

M02. Data Consistency

Impact	Medium
Likelihood	Medium

Each time a user stakes, the lastDepositTime for that user is updated to the current block timestamp. Consequently, even for the user's initial stake that was made a significant duration ago, the user must wait until the end of the unlock period to make a withdrawal. This system may discourage users from making new deposits if they want to maintain the possibility of unstaking within a predefined time frame.

If users find the current staking system inconvenient, they may choose not to use the system, leading to a decrease in user engagement and possibly affecting the platform's overall usage and liquidity.

Path: ./contracts/Staking.sol : stakeSOIL()

Recommendation: consider implementing a feature that allows users to have multiple stakes with distinct unlock periods. This change could improve the user experience by offering more flexibility in managing their stakes. It would permit users to unstake their initial deposits according to their original schedules, even if they make new stakes later.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

M03. Requirements Violation

Impact	Medium
Likelihood	Medium

If the block.timestamp is within the first release interval, i.e., block.timestamp - userVault.releaseFrom is less than userVault.releaseInterval, the code incorrectly allows tokens to be unlocked. This is contrary to the documentation, which states that tokens should not be unlockable before the first interval of token release has passed.

This is due to the following line in the code:

spendableIntervals = (block.timestamp - userVault.releaseFrom + userVault.releaseInterval) / userVault.releaseInterval;

A user can unlock tokens immediately when the releaseFrom timestamp is reached, and throughout the first interval, which is contrary to the expected behavior. This could potentially disrupt the planned token release schedule.

Path: ./contracts/Vesting.sol : _vaultInfo()

Recommendation: to maintain clarity and consistency, ensure that the documentation aligns with the current implementation. If the tokens are intended to be unlocked as soon as the releaseFrom timestamp is reached, update the documentation accordingly to reflect this behavior. Alternatively, if the documentation's indication that tokens should not be unlockable until a full releaseInterval has passed is correct, then the code should be updated to match this description.

Found in: a3d764b

Status: Mitigated (The token release mechanism is designed to begin unlocking tokens as soon as the 'releaseFrom' timestamp is reached.)

M04. Race Condition

Impact	**High**
Likelihood	**Low**

The deposit, withdraw, claimRewards and upgradeToPoolWithHigherYield functions in the pool contract deduct a fee (poolsContractFee) from the deposited amount. The admin can change this fee at any time via the changePoolsContractFee function.

When a user initiates a deposit transaction, they consider the current fee. However, the admin could change the fee before the transaction processes. This could lead to the user depositing a different amount than intended or even transaction failure due to checks within the deposit function.

Path: ./contracts/PoolsContract.so : changePoolsContractFee()

Recommendation: consider a mechanism that prevents sudden fee changes and max fee limit, ensuring users have sufficient notice before a new fee applies.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

M05. Integer Overflow/Underflow

Impact	**High**
Likelihood	**Low**

The current logic in the withdraw function may incorrectly allow a user to initiate a withdrawal where the total fee (base and early withdrawal fees) exceeds the withdrawal amount.

If a user tries to withdraw their funds early, and the total fees exceed the withdrawal amount, the operation would fail due to a SafeMath underflow error (Solidity version 0.8.0 and above). This can lead to a situation where the user may not be able to withdraw their funds early.

Path: ./contracts/PoolsContract.so : withdraw()

Recommendation: update the condition to account for the total fee (chargedFee) rather than just the base fee (poolsContractFee), as shown below:

```
require(amountToWithdraw > chargedFee, "Amount must be greater than fees");
```

This modification will ensure the correct fee is considered when checking if the withdrawal amount is greater than the applicable fees, whether it is an early withdrawal or not.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

M06. Data Consistency

Impact	High
Likelihood	Low

The contract exhibits inconsistencies in fee deductions across the withdraw, claimRewards and upgradeToPoolWithHigherYield functions. For withdraw and upgradeToPoolWithHigherYield, fees are deducted from the amount being withdrawn or moved, but not from the rewards being claimed. Conversely, in the claimRewards function, the fee is deducted directly from the rewards.

This inconsistency in fee deduction can lead to confusion for users and could potentially affect the total amount they expect to receive from a transaction. The inconsistency also brings a level of unpredictability to the user experience, which can affect user trust and contract engagement.

Path: ./contracts/PoolsContract.so : withdraw(), claimRewards(), upgradeToPoolWithHigherYield()

Recommendation: given the impact on user expectations, it is recommended to implement a consistent approach to fee deductions across all functions. In this case, not deducting the fee from the rewards would be a beneficial approach, considering the stated fixed APRs. Alongside this, it is essential to provide detailed public documentation explaining the fee structures, calculations, and the impact on rewards and withdrawals.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

M07. Redundant Code

Impact	Low
Likelihood	High

The deposit function in the Pool contract, the createVault function in the Vesting contract, and the stakeSOIL function in the Staking contract all contain redundant lines of code that manually verify the sufficient token allowance and balance.

The unnecessary checks add computational complexity to the contract, leading to higher Gas costs for users. Additionally, the superfluous code reduces the overall clarity and efficiency of the contract.

Paths: ./contracts/PoolsContract.so : deposit()

./contracts/Vesting.sol : createVault()

./contracts/Staking.sol : stakeSOIL()

Recommendation: the redundant balance and allowance checks should be replaced with token.safeTransferFrom(tokenSpender, address(this), amount); line. The safeTransferFrom function inherently handles these checks, eliminating the need for additional manual verification and improving the overall efficiency of the contract.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

M08. Data Consistency

Impact	Medium
Likelihood	Medium

The createVault function in the smart contract accepts parameters amount and tokensAmountPerInterval but does not enforce that the amount is a multiple of tokensAmountPerInterval or that tokensAmountPerInterval is less than or equal to the amount.

This can lead to a vault being created where the total amount of tokens is not divisible by the amount released per interval, causing irregular distribution schedules and potential confusion.

Additionally, it can create situations where tokensAmountPerInterval is greater than the total amount of tokens in the vault. This would result in the entire vault being depleted in the first release, and no tokens would be available for subsequent releases.

Path: ./contracts/Vesting.sol : createVault()

Recommendation: add a requirement in the createVault function to ensure that amount is a multiple of tokensAmountPerInterval and that tokensAmountPerInterval is not greater than the amount. This would enforce a consistent and intuitive distribution schedule for all vaults.

Found in: a3d764b

Status: Mitigated (A check has been implemented to ensure that tokensAmountPerInterval cannot exceed the total amount. This adjustment still allows for the possibility of unlocking all tokens in a vesting period, if tokensAmountPerInterval equals the total amount to be vested.)

Low

L01. Data Consistency

Impact	Low
Likelihood	Low

The function unlockTokensOnDemand allows an admin to unlock any number of unclaimed tokens for a user without considering the releaseFrom time and the number of releaseInterval passed. This implies that tokens can be unlocked for a user before their appropriate release time, contradicting the vesting schedule.

Such an implementation may lead to a situation where tokens are released prematurely before the appropriate vesting period. It also breaks the consistency of the vesting schedule, where tokens are meant to be released in defined intervals.

Path: ./contracts/Vesting.sol : unlockTokensOnDemand()

Recommendation: modify the unlockTokensOnDemand function to only allow the unlocking of tokens that are due for release based on the releaseFrom time and releaseInterval.

This could be achieved by first calculating the amount of releaseInterval that has passed since the releaseFrom time and then allowing the admin to unlock only up to the calculated number of tokens. This way, the function respects the vesting schedule and maintains fairness across all users.

Found in: a3d764b

Status: Mitigated (This functionality will be reserved only for vaults that are meant to be created for liquidity group)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

L02. Missing Events on Critical State Updates

Impact	Low
Likelihood	Medium

Critical state changes should emit events for tracking things off-chain.

This can lead to inability for users to subscribe events and check what is going on with the project.

Path: ./contracts/ProtocolSettings.sol : updateBackendSigner()

Recommendation: emit events on critical state changes.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

L03. Missing Validation

Impact	Medium
Likelihood	Low

The project should be consistent and contain no self contradictions.

According to the implementation of *SoilToken constructor(),* it accepts arrays without length restriction. If the array has more than 255 elements, it will revert (counter in loop has max value of 255).

According to the implementation of *PoolsContract createPool()* and *updatePool()*, these functions accept arrays without length restriction. If the array *periods[]* has more than 255 elements, it will revert (counter in loop inside *_validatePoolInfo()* has max value of 255).

This may lead to unexpected value processed by the contract.

Paths:

./contracts/SoilToken.sol : constructor(),

./contracts/PoolsContract.sol : createPool(), updatePool(),

Recommendation: provide documentation, comments and identifiers in code consciously, implement the validations.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

L04. Non-Finalized Code

Impact	Low
Likelihood	Low

The production code should not contain any functions, variables or commented code parts that are being used solely in the test environment.

Path: ./contracts/Vesting.sol : withdraw()

Recommendation: remove commented code.

Found in: cd03b51

Status: Fixed (Revised commit: 7ec15f8)

Informational

I01. State Variables Can Be Declared Immutable

The protocolSettings variable's value is set in the constructor. This variable can be declared immutable

Path: ./contracts/Vesting.sol

Recommendation: declare mentioned variable as immutable.

Found in: a3d764b

Status: Fixed (Revised commit: 7ec15f8)

I02. Redundant Code

The _vaultInfo function presents a redundancy in the check for userVault.unclaimedAmount == 0. The first instance of this check appears at the function's beginning, prompting an immediate return if userVault.unclaimedAmount == 0 is true. However, a subsequent check for the same condition surfaces within an if block: if (block.timestamp >= userVault.releaseFrom). Given that the function would have already existed at the initial check, this second evaluation is unnecessary, constituting redundant code.

Users may incur slightly higher gas costs when interacting with the contract due to the additional computation required by the redundant check.

Path: /contracts/Vesting.sol : _vaultInfo()

Recommendation: remove the second check for userVault.unclaimedAmount == 0 from the function.

Found in: a3d764b

Status: Fixed (Revised commit: 7ec15f8)

I03. Out-Of-Bounds Array Access

The function _vaultInfo can potentially be called with an index that is out of bounds for the vaults[userAddress] array. This will result in a "revert" error which is thrown by Solidity when attempting to access an array index that does not exist. This can disrupt the flow of contract operations and lead to a failed transaction.

This issue occurs because there is no check in the _vaultInfo function to ensure that the given index is within the bounds of the vaults[userAddress] array.

Path: /contracts/Vesting.sol : _vaultInfo()

Recommendation: add a check in the _vaultInfo function to verify the index is within the length of the vaults[userAddress] array. If the check fails, the function can revert with an appropriate error message. This will ensure that all function calls to _vaultInfo are valid and will not result in a failed transaction.

Found in: a3d764b

Status: Fixed (Revised commit: 7ec15f8)

I04. Style Guide Violation

The provided projects should follow the official guidelines.

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Within a grouping, place the view and pure functions last.

It's best practice to cover all functions with NatSpec annotation and to follow the Solidity naming convention. This will increase overall code quality and readability.

Paths: ./contracts/PoolsContract.sol

./contracts/Vesting.sol

./contracts/Staking.sol

Recommendation: follow the official Solidity guidelines.

Found in: a3d764b

Status: Fixed (Revised commit: 7ec15f8)

I05. Inefficient Gas Model - Counter Increment

To save some gas, increment operation of the counter inside *for* loop can be done at the end of the loop in an *unchecked{}* code block. It omits overflow checks and saves Gas. This can be safely used when overflow is not possible.

Paths: ./contracts/SoilToken.sol : constructor(), updateWhitelistOfSnapshooters(),

./contracts/Vesting.sol : withdraw(),

Recommendation: put the post-iteration increment operation at the end of the loop inside an *unchecked{}* code block.

Found in: a3d764b

Status: Reported

I06. Missing Zero Address Validation

Address parameters are being used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Path:

./contracts/PoolsContract.sol : constructor(),

Recommendation: implement zero address checks.

Found in: a3d764b

Status: Fixed (Revised commit: cd03b51)

I07. Unused Identifier

The variable *softCap* inside the *PoolInfo* struct is never used.

Path:
./contracts/SoilToken.sol,

Recommendation: remove unused variable from struct.

Found in: a3d764b

Status: Mitigated (*softCap* is used for off-chain operations)

I08. Redundant Import

The PoolsContract and Staking contracts are importing the deprecated draft-EIP712.sol instead of the final version of EIP712.sol.

Paths: ./contracts/PoolsContract.sol

./contracts/Staking.sol

Recommendation: replace the deprecated draft-EIP712.sol with the final EIP712.sol

Found in: cd03b51

Status: Fixed (Revised commit: 7ec15f8)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://gitlab.nextrope.com/client/soil/soil-blockchain/-/tree/develop/
Commit	a3d764b2c01dcafcbf8fde5b248d76920779cd48
Whitepaper	
Requirements	NatSpec
Technical Requirements	File: Soil functional and technical requirements.docx SHA3: 69e2a989583458146e89a43055ba1db6054184ac799d9884f01dc17048ff9c56
Contracts	File: PoolsContract.sol SHA3: af0d8233c6eab53fec9e10055905ba3471861c6f99ffd23977e8b3eadd1ba6df File: ProtocolSettings.sol SHA3: f12680f7e113b9dbb1328f86a93d00111ee8d2145b1376cf98c1ea85b67b6313 File: SoilToken.sol SHA3: c457c78c50ad2282dfcc2abcd3ddde08b6e0f527e18e24930998752d388b6a51 File: Staking.sol SHA3: e402ff8958b339c7e8760798bad399cdb7dba8f270038cc40328e6c18fe064ca File: VerifySignatureSystem.sol SHA3: 5a3e0ebf721c87c4061e351fe66efc1e5bd60f9e3d3dc260ceff6a2995ffb0be File: Vesting.sol SHA3: e3ee37c0246452d99b0524d942642376663060d8a228e1f00288b384925752e9

Second review scope

Repository	https://gitlab.nextrope.com/client/soil/soil-blockchain/-/tree/develop/
Commit	cd03b51f3ac3be0937d7a5f3e761890b7a3c0b2d
Whitepaper	Link
Requirements	NatSpec
Technical Requirements	File: Soil functional and technical requirements.docx SHA3: fdf6b93a65a6934fd3b58ca30c53b7a10f93854056c9e04251e959f285d39a3a
Contracts	File: PoolsContract.sol SHA3: edc3c9bb46dc3c79e293949d7bb219a68d385215a8c380e4b4ce6a835073f81f File: ProtocolSettings.sol SHA3: f3fff2f8ec374a57bd233fdb2d403be33a161529d5ed301b17fc164d05d30d31 File: SoilToken.sol

	SHA3: aa3349690e31aef6df2fcc04e4b1b71b0e2ea605d77141f103e2a6ae1380ac5b File: Staking.sol SHA3: 742ef1ae19727e8f5b0893b55bc64b2b395b5799127a4d5d464fd4f8cdc5ce4d File: VerifySignatureSystem.sol SHA3: 1718e130a068fb562e10f5f00d5649e69bf1a3dc3eb2564c49e3c6bbb368433a File: Vesting.sol SHA3: bb233d1a455401bad0ab3ea8449c6dbe012121a4402ebc43b9f6e02842ae6c60

Third review scope

Repository	https://gitlab.nextrope.com/client/soil/soil-blockchain/-/tree/develop/
Commit	d71b55edbf34fb3e1e94f3a991aa5b8440cbc299
Whitepaper	Link
Requirements	NatSpec
Technical Requirements	File: Soil Staking Fee Docs.pdf SHA3: dec8247dfcbc0457208bdb3f13980403172636df460841ca8266bb066f143602 File: SOIL_General Loan Terms_230516.docx SHA3: 3b67e226a66dd2d9153de97684a8c9be9e929221d83fcab3011ebee116f974e2 File: Soil functional and technical requirements.pdf SHA3: f3eff92ecac619cdfbbd4f52119498f06c04c95296a203ad929327f64ca8adc4 File: Tokenomics.pdf SHA3: 6f251c44af46ca101e752b004f1ee58e162c11dc08cd3a33680d723299a0faf8
Contracts	File: PoolsContract.sol SHA3: bdf1e2fad973991d970dc3f08b2488cc04a4556022c1dff918f3015941f0c627 File: ProtocolSettings.sol SHA3: 23596d95f90b7929da875c761fd08b95b0917b4a193a1d09f7ef30596cecc0d7 File: SoilToken.sol SHA3: 2da58aeea54c59e02a269c37a5ae3732c5e771fc6c06ccb62fef4aad18e45082 File: Staking.sol SHA3: e2777c1b27567e276ca82ca9cdb15c33f9bac86967849ab24bbee8409eb7627a File: VerifySignatureSystem.sol SHA3: 756cb67d55ab82e4ac8ea45464d534c6f588c971963f8fea23ce5c8eac2bbf13 File: Vesting.sol SHA3: 3e0a2b6e7e21c2a4d392eb7f687ccc3741c7bec3ba99e4ea3f5df94262ab2a80