Audited by CRED SHIELDS

CredShields

Smart Contract Audit

Oct 30th, 2024

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of SavePlanetEarth between 23rd Oct 2024, and 27th Oct 2024. A retest was performed on 30th Oct 2024.

Author

Shashank (Co-founder, CredShields) shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead),Shreyas Koli(Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor)

Prepared for

SavePlanetEarth

Table of Contents

1. Executive Summary ----------

SavePlanetEarth engaged CredShields to perform a smart contract audit from 23rd Oct 2024, to 27th Oct 2024. During this timeframe, Nineteen (19) vulnerabilities were identified. **A retest was performed on 30th Oct 2024, and all the bugs have been addressed.**

During the audit, Three (3) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "SavePlanetEarth" and should be prioritized for remediation. **All such vulnerabilities were addressed and confirmed to have been fixed during the retest.**

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
SPE Smart Contract	2	1	2	4	3	17	**19**
	2	**1**	**2**	**4**	**3**	**17**	**19**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in SPE Smart Contract 's scope during the testing window while abiding by the policies set forth by SavePlanetEarth's team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both SavePlanetEarth's internal security and development teams to not only identify specific vulnerabilities but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at SavePlanetEarth can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, SavePlanetEarth can future-proof its security posture and protect its assets.

2. The Methodology ----------

SavePlanetEarth engaged CredShields to perform a SPE Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from 23rd Oct 2024, to 27th Oct 2024, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed upon:

IN SCOPE ASSETS
SPE.sol

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting phase

SavePlanetEarth is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	None	Low	Medium
		LOW	MEDIUM	HIGH
Likelihood				

Overall, the categories can be defined as described below -

1. **Informational**

   We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

2. **Low**

   Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

3. **Medium**

   Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. **High**

   High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. **Critical**

   Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. **Gas**

   To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- Shashank, Co-founder CredShields shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have about the engagement or this document.

3. Findings Summary ----------

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, Nineteen (19) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Lack of Mint and Burn Functionality	Critical	Functional Limitation
Fee Circumvention in batchTransfer() Function	Critical	Business Logic
WETH to ETH is not unwrapped in _swapAndLiquify()	High	Incorrect Token Handling
Zero Slippage value	Medium	Slippage Risk
Underflow in _transferStandard() Function	Medium	Arithmetic Underflow
Missing Zero Address Validations	Low	Missing Input Validation
Missing Events in Important Functions	Low	Missing Best Practices
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)

Use Ownable2Step	Low	Missing Best Practices
Require with Empty Message	Informational	Code Optimization
Use Call instead of Transfer	Informational	Missing Best Practices
Gas Optimization in Require/Revert Statements	Gas	Gas Optimization
Dead Code	Informational	Missing Best Practices
Cheaper Inequalities in require()	Gas	Gas Optimization
Gas Optimization in Increments	Gas	Gas Optimization
Cheaper Inequalities in if()	Gas	Gas Optimization
Cheaper Conditional Operators	Gas	Gas Optimization
Multiplication/Division by 2 should use Bit-Shifting	Gas	Gas Optimization
Custom error to save gas	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Not Vulnerable	Contract uses floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0
SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use

SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found
SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.

SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status ----------

SavePlanetEarth is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on 30th Oct 2024, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Lack of Mint and Burn Functionality	Critical	**Won't Fix** [30/10/2024]
Fee Circumvention in batchTransfer() Function	Critical	**Fixed** [30/10/2024]
WETH to ETH is not unwrapped in _swapAndLiquify()	High	**Fixed** [30/10/2024]
Zero Slippage value	Medium	**Fixed** [30/10/2024]
Underflow in _transferStandard() Function	Medium	**Fixed** [30/10/2024]
Missing Zero Address Validations	Low	**Fixed** [30/10/2024]
Missing Events in Important Functions	Low	**Fixed** [30/10/2024]
Floating and Outdated Pragma	Low	**Fixed** [30/10/2024]
Use Ownable2Step	Low	**Fixed** [30/10/2024]
Require with Empty Message	Informational	**Fixed** [30/10/2024]
Use Call instead of Transfer	Informational	**Fixed** [30/10/2024]

Gas Optimization in Require/Revert Statements	Gas	**Fixed** [30/10/2024]
Dead Code	Informational	**Fixed** [30/10/2024]
Cheaper Inequalities in require()	Gas	**Won't Fix** [30/10/2024]
Gas Optimization in Increments	Gas	**Fixed** [30/10/2024]
Cheaper Inequalities in if()	Gas	**Won't Fix** [30/10/2024]
Cheaper Conditional Operators	Gas	**Fixed** [30/10/2024]
Multiplication/Division by 2 should use Bit-Shifting	Gas	**Fixed** [30/10/2024]
Custom error to save gas	Gas	**Fixed** [30/10/2024]

Table: Summary of findings and status of remediation

5. Bug Reports ----------

Bug ID #1[Won't Fix]

Lack of Mint and Burn Functionality

Vulnerability Type
Functional Limitation

Severity
Critical

Description
The contract implements various functionalities typical of an token, such as transferring, approving, and allowance. However, it lacks mint and burn functions, which are essential in many token standards and use cases.

Minting: Without a mint function, the contract cannot increase the token supply, which may be necessary for several reasons:
Burning: Without a burn function, users or the contract cannot reduce the token supply

The lack of these functions limits the contract's functionality and may reduce its flexibility and appeal in scenarios where such mechanisms are expected or required.

Affected Code
- SPE.sol

Impacts
This limits the contract's ability to distribute additional tokens in the future for various purposes, such as rewards, incentives, or new allocations. sers cannot burn their tokens, which could be essential for deflationary mechanics or for users wanting to remove tokens permanently.

Remediation
It is recommended to implement a mint function that allows tokens to be created and assigned to an address. Ensure access control, so only authorized accounts can call it. And also Implement a burn function that allows token holders to burn (destroy) their tokens, reducing the total supply.

Retest

Client comment : The SPE tokenomics and whitepaper state that our token contract is unable to mint anymore tokens. This is attractive to our investor base and a selling point for SPE. Also, since launch in 2021, we have used the 0x00....DEAD as the burn address and will continue to do so, this way it is easily visible on blockchain explorers where all of the token supply is and how it got there. For these reasons, we feel it is unnecessary to implement mint and burn functions.

Bug ID #2[Fixed]

Fee Circumvention in batchTransfer() Function

Vulnerability Type
Business Logical

Severity
Critical

Description
In this contract, the transfer() function is used to transfer tokens, and it internally calls the _transfer() function. The _transfer() function calculates fees, specifically _stakingRewardsFee and _liquidityFee, and then calls _transferStandard() to update balances after accounting for these fees.

However, the batchTransfer() function, which is used to transfer tokens in bulk, directly calls _transferStandard() in a loop without calculating or applying any fees. If a user invokes batchTransfer() when fees are set to zero, they can avoid paying any fees on their transfers, effectively circumventing the intended fee mechanism for multiple token transfers.

Affected Code
- SPE.sol#batchTransfer()

Impacts
Since batchTransfer() does not apply fees, users can transfer tokens in bulk without incurring fees, resulting in a loss of fee revenue for the system.

Remediation
To fix this issue it is recommended to call _transfer() in batchTransfer() instead of _transferStandard() function.

Retest
This vulnerability has been fixed by implementing above remediation.

Bug ID #3 [Fixed]

WETH to ETH is not unwrapped in _swapAndLiquify()

Vulnerability Type
Incorrect Token Handling

Severity
High

Description
The _swapAndLiquify() function in this contract is designed to convert a portion of tokens to ETH, then add both tokens and ETH as liquidity to the liquidity pool. However, there is a critical flaw in how the contract handles the swapped token:

1. The _swapTokensForEth() function swaps tokens for WETH, not ETH.
2. The contract does not unwrap WETH to ETH after the swap, resulting in the contract holding WETH instead of ETH.
3. Consequently, newBalance in _swapAndLiquify() will always be zero because address(this).balance only tracks ETH, not WETH.

This results in a failed call to _addLiquidity(), as it attempts to add liquidity with an ethAmount of zero. The function will therefore fail every time liquidity is added, preventing the contract from maintaining proper liquidity and causing functionality issues with token swaps and liquidity additions.

Affected Code
- SPE.sol#_swapAndLiquify()

Impacts
Since ethAmount is always zero, the _addLiquidity() function will fail, causing the contract to be unable to add liquidity to the liquidity pool as intended.

Remediation
It is recommended to Modify _swapTokensForEth() or _swapAndLiquify() to unwrap WETH into ETH after performing the swap. This way, address(this).balance will reflect the correct ETH balance.

Retest

This issue has been fixed by unwrapping tokens in `_swapTokensForEth()` function.

Bug ID #4 [Fixed]

Zero Slippage value

Vulnerability Type
Slippage Risk

Severity
Medium

Description
_swapTokensForEth() and _addLiquidity() function that is part of a contract handling token swaps and adding liquidity to pool. In this function, a token swap operation is performed with a hardcoded (slippage) value set to zero. Slippage refers to the maximum acceptable difference between the expected price of an asset and the actual executed price during a swap and adding liquidity. A slippage of zero means that the code expects the swap to return a value near 0.

Affected Code
- _swapTokensForEth()
- _addLiquidity()

Impacts
The impact of setting the slippage to zero is that the code assumes that the token swap will always occur at an exact price without any price fluctuations. In a decentralized environment, token prices can vary rapidly due to market conditions, resulting in the possibility of the swap failing or being executed at a significantly different rate than expected. This can lead to undesirable outcomes, including failed transactions or losses for users.

Remediation
To make the token swap function more robust and adaptable to market conditions, it is recommended to set a non-zero slippage tolerance (e.g., a small percentage) rather than a hardcoded zero value. This will allow the code to accommodate minor price fluctuations and ensure that the swap is more likely to succeed.

Retest
This vulnerability has been fixed by adding slippage in given functions.

Bug ID #5 [Fixed]

Underflow in _transferStandard() Function

Vulnerability Type
Arithmetic Underflow

Severity
Medium

Description
The _transferStandard() internal function in this contract is responsible for transferring tokens from the sender to the recipient and deducts the transfer amount, tAmount, from the sender's balance. However, the function lacks a validation check to ensure that the sender has a sufficient balance to cover tAmount.

As a result, if tAmount user input exceeds _tOwned[sender] (the sender's balance), an arithmetic underflow will occur when performing _tOwned[sender] = _tOwned[sender] - tAmount;. This can lead to unexpected behavior, as underflows may not generate explicit errors in certain environments and could set _tOwned[sender] to a large value, disrupting the contract's balance tracking. The function should include a check to verify that the sender's balance is sufficient to perform the transfer, and if not, revert with an appropriate error message (e.g., "Insufficient balance").

Affected Code
- SPE.sol#_transferStandard()

Impacts
Users attempting transfers with insufficient balances will not receive a clear error message, which could lead to confusion or an unintentional loss of funds.

Remediation
Ensure that _transferStandard() checks if the sender's balance is sufficient before performing the transfer by adding a require statement. This will prevent underflow and provide a clear error message if the sender lacks sufficient funds.

```
function _transferStandard(address sender, address recipient, uint256 tAmount) private {
————————code ———————-



// Check if sender has enough balance
require(_tOwned[sender] >= tAmount, "Insufficient balance");
_tOwned[sender] = _tOwned[sender] - tAmount; _tOwned[recipient] = _tOwned[recipient] +
tTransferAmount;

————————————code ——————————
}
```

Retest

This vulnerability has been fixed by implementing above validation.

Bug ID #6 [Fixed]

Missing Zero Address Validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
The contracts were found to be setting new addresses without proper validations for zero addresses.
Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever.
Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost forever.

Affected Code
- SPE.sol#constructor(liqWallet, stakingRewardsWallet)

Impacts
If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.

Remediation
Add a zero address validation to all the functions where addresses are being set.

Retest
This vulnerability has been fixed by applying zero address validation

Bug ID #7 [Fixed]

Missing Events in Important Functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code
The following functions were affected -

- SPE.sol#excludeFromFee(address account)
- SPE.sol#includeInFee(address account)
- SPE.sol#setBuyStakingRewardsFeePercent()
- SPE.sol#setBuyLiquidityFeePercent()
- SPE.sol#setSellStakingRewardsFeePercent()
- SPE.sol#setSellLiquidityFeePercent()
- SPE.sol#setMaxTxPercent()

Impacts
Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation
Consider emitting events for important functions to keep track of them.

Retest

This issue has been fixed by emitting right events in given functions.

Bug ID #8 [Fixed]

Floating and Outdated Pragma

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.
The contract allowed floating or unlocked pragma to be used, i.e., ^0.8.7. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Affected Code

- SPE.sol

Impacts
If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.
Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.
The likelihood of exploitation is low.

Remediation
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.25 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest
This vulnerability has been fixed.

Bug ID #9 [Fixed]

Use Ownable2Step

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Affected Code
- contract SavePlanetEarth is Context, IERC20, Ownable {

Impacts
Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

Remediation
It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

Retest
This vulnerability has been fixed.

Bug ID #10 [Fixed]

Require with Empty Message

Vulnerability Type
Code optimization

Severity
Informational

Description
During analysis; multiple require statements were detected with empty messages. The statement takes two parameters, and the message part is optional. This is shown to the user when and if the require statement evaluates to false. This message gives more information about the conditional and why it gave a false response.

Affected Code

- SPE.sol#setNumTokensSellToAddToLiquidity()
- SPE.sol#withdrawalEther()

Impacts
Having a short descriptive message in the require statement gives users and developers more details as to why the conditional statement failed and helps in debugging the transactions.

Remediation
It is recommended to add a descriptive message, no longer than 32 bytes, inside the require statement to give more detail to the user about why the condition failed.

Retest
This vulnerability has been fixed.

Bug ID #11 [Fixed]

Use Call instead of Transfer

Vulnerability Type
Missing Best Practices

Severity
Informational

Description:
Using Solidity's transfer function has some notable shortcomings when the withdrawer is a smart contract, which can render ETH deposits impossible to withdraw. Specifically, the withdrawal will inevitably fail when:

- The withdrawer smart contract does not implement a payable fallback function.
- The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units.
- The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

Affected Code

- SPE.sol#withdrawalEther()

Impacts
The transfer function has some restrictions when it comes to sending ETH to contracts in terms of gas which could lead to transfer failure in some cases.

Remediation
It is recommended to transfer ETH using the call() function, handle the return value using require statement, and use the nonreentrant modifier wherever necessary to prevent reentrancy.
Ref: https://solidity-by-example.org/sending-ether/

Retest
This vulnerability has been fixed.

Bug ID #12 [Fixed]

Gas Optimization in Require/Revert Statements

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The require/revert statement takes an input string to show errors if the validation fails.
The strings inside these functions that are longer than 32 bytes require at least one additional MSTORE, along with additional overhead for computing memory offset and other parameters. For this purpose, having strings lesser than 32 bytes saves a significant amount of gas.

Affected Code

- SPE.sol#setStakingRewardsWalletAddress(address _address)
- SPE.sol#setLiqWalletAddress(address _address)
- SPE.sol#_approve(address owner, address spender, uint256 amount)
- SPE.sol#_transfer(address from, address to, uint256 amount)
- SPE.sol#batchTransfer(address[ ] calldata recipients, uint256[ ] calldata amounts)

Impacts
Having longer require/revert strings than 32 bytes cost a significant amount of gas.

Remediation
It is recommended to shorten the strings passed inside require/revert statements to fit under 32 bytes. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

Retest
This vulnerability has been fixed.

Bug ID #13 [Fixed]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informational

Description
It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities. The functions and parameters, contracts, and interfaces that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

Affected Code

- address[ ] private _excluded

Impacts
This does not impact the security aspect of the Smart contract but prevents confusion when the code is sent to other developers or auditors to understand and implement.
This reduces the overall size of the contracts and also helps in saving gas.

Remediation
If the library functions are not supposed to be used anywhere, consider removing them from the contract.

Retest
This vulnerability has been fixed.

Bug ID #14 [Won't Fix]

Cheaper Inequalities in require()

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be performing comparisons using inequalities inside the require statement. When inside the require statements, non-strict inequalities (>=, <=) are usually costlier than strict equalities (>, <).

Affected Code

- SPE.sol#setBuyStakingRewardsFeePercent(uint256 stakingRewardsFee)
- SPE.sol#setSellStakingRewardsFeePercent(uint256 stakingRewardsFee)
- SPE.sol#_transfer(address from, address to, uint256 amount)

Impacts
Using non-strict inequalities inside "require" statements costs more gas.

Remediation
It is recommended to go through the code logic, and, **if possible**, modify the non-strict inequalities with the strict ones to save gas as long as the logic of the code is not affected.

Retest
Client comment : Unable to optimize further because fees must be equal to 10% or less, and also does not apply to _transfer in this case.

Bug ID #15 [Fixed]

Gas Optimization in Increments

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract uses two for loops, which use post increments for the variable “i”.
The contract can save some gas by changing this to ++i.
++i costs less gas compared to i++ or i += 1 for unsigned integers. In i++, the compiler has to create a temporary variable to store the initial value. This is not the case with ++i in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Affected Code

- SPE.sol#batchTransfer(address[ ] calldata recipients, uint256[ ] calldata amounts)

Impacts
Using i++ instead of ++i costs the contract deployment around 600 more gas units.

Remediation
It is recommended to switch to ++i and change the code accordingly so the function logic remains the same and meanwhile saves some gas.

Retest
This vulnerability has been fixed.

Bug ID #16 [Won't Fix]

Cheaper Inequalities in if()

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be doing comparisons using inequalities inside the "if" statement. When inside the "if" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- SPE.sol#_transferStandard(address sender, address recipient, uint256 tAmount)
- SPE.sol#_takeLiquidity(uint256 tLiquidity)
- SPE.sol#_takeStakingRewards(uint256 tStakingRewards)

Impacts
Using strict inequalities inside "if" statements costs more gas.

Remediation
It is recommended to go through the code logic, and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest
Client comment : No actual change is needed in these instances; all conditions are already logically and gas-efficiently optimized. The gas cost associated with executing these if-statements as they are currently written is minimal, and further adjustments would not improve gas consumption.

Bug ID #17 [Partially Fixed]

Cheaper Conditional Operators

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Upon reviewing the code, it has been observed that the contract uses conditional statements involving comparisons with unsigned integer variables. Specifically, the contract employs the conditional operators x != 0 and x > 0 interchangeably. However, it's important to note that during compilation, x != 0 is generally more cost-effective than x > 0 for unsigned integers within conditional statements.

Affected Code

- SPE.sol#setNumTokensSellToAddToLiquidity(uint256 _amount)
- SPE.sol#_transfer(address from, address to, uint256 amount)
- SPE.sol#_transferStandard(address sender, address recipient, uint256 tAmount)
- SPE.sol#_takeLiquidity(uint256 tLiquidity)
- SPE.sol#_takeStakingRewards(uint256 tStakingRewards)

Impacts
Employing x != 0 in conditional statements can result in reduced gas consumption compared to using x > 0. This optimization contributes to cost-effectiveness in contract interactions.

Remediation
Whenever possible, use the x != 0 conditional operator instead of x > 0 for unsigned integer variables in conditional statements.

Retest
Client comment : In _transferStandard no change is needed for this as it already uses >= . All other changes (4) have been implemented

Bug ID #18 [Fixed]

Multiplication/Division by 2 should use Bit-Shifting

Vulnerability Type
Gas Optimization

Severity
Gas

Description
In Solidity, the EVM (Ethereum Virtual Machine) executes operations in terms of gas consumption, where gas represents the computational cost of executing smart contract functions. Multiplication and division by two can be achieved using either traditional multiplication and division operations or bitwise left shift (<<) and right shift (>>) operations, respectively. However, using bit-shifting operations is more gas-efficient than using traditional multiplication and division operations.
x * 2 can be replaced with x << 1.
x / 2 can be replaced with x >> 1.

Affected Code

- SPE.sol#_swapAndLiquify(uint256 contractTokenBalance)

Impacts
Gas consumption directly affects the cost of executing smart contracts on the Ethereum blockchain. Using bit-shifting operations for multiplication and division by two reduces the gas cost from 5 to 3, leading to more cost-effective and efficient smart contract execution. This optimization is particularly relevant in scenarios where gas efficiency is crucial, such as high-frequency operations or resource-intensive contracts.

Remediation
It is recommended to use left and right shift instead of multiplying and dividing by 2 to save some gas.

Retest

This vulnerability has been fixed.

Bug ID #19 [Fixed]

Custom error to save gas

Vulnerability Type
Gas Optimization

Severity
Gas

Description
During code analysis, it was observed that the smart contract is using the revert() statements for error handling. However, since Solidity version 0.8.4, custom errors have been introduced, providing a better alternative to the traditional revert(). Custom errors allow developers to pass dynamic data along with the revert, making error handling more informative and efficient. Furthermore, using custom errors can result in lower gas costs compared to the revert() statements.

Affected Code

- SPE.sol#_transfer(address from, address to, uint256 amount)

Impacts
Custom errors allow developers to provide more descriptive error messages with dynamic data. This provides better insights into the cause of the error, making it easier for users and developers to understand and address issues.

Remediation
It is recommended to replace all the instances of revert() statements with error() to save gas..

Retest
This vulnerability has been fixed.

6. The Disclosure ----------

The Reports provided by CredShields are not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.

YOUR SECURE FUTURE STARTS HERE

CRED SHIELDS

At CredShields, we're more than just auditors. We're your strategic partner in ensuring a secure Web3 future. Our commitment to your success extends beyond the report, offering ongoing support and guidance to protect your digital assets

Audited by CRED SHIELDS

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com