33AUDITS & CO.

Seraph Token Audit Report

Introduction

A time-boxed security review of the Seraph Token protocol was done, focusing on the security aspects of the smart contracts.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About Seraph Token

Seraph Token is a token that allows for vesting of tokens to a beneficiary.

Severity Classifications

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Scope

- contracts/SERAPH.sol
- contracts/VestingWallet.sol

Findings Summary

ID	Title	Severity	Status

ID	Title	Severity	Status
[M-01]	Unsafe Beneficiary Change Mechanism	Medium	Open
[M-02]	Unsafe Vesting Parameter Validation	Medium	Open
[L-01]	Incorrect Token Decimals Handling	Low	Open
[L-02]	Floating Pragma	Low	Open
[L-03]	Missing Events for Critical Operations	Low	Open
[L-04]	Use of Custom Errors	Low	Open

[M-01] Unsafe Beneficiary Change Mechanism

Severity:

Medium

Description:

The beneficiary change is performed in a single step, which could result in the beneficiary being permanently lost if set to an incorrect address.

Impact:

If the new beneficiary address is incorrect, funds could be permanently locked in the contract.

Resolution:

Implement a two-step transfer process with propose and accept functions. Here's an example implementation:

```
contract VestingWallet {
    // ... existing contract code ...

    address public pendingBeneficiary;

    function proposeBeneficiary(address newBeneficiary) external {
        require(msg.sender == beneficiary, "Only beneficiary can
propose");
        require(newBeneficiary != address(0), "Cannot propose zero
address");
        pendingBeneficiary = newBeneficiary;
    }

    function acceptBeneficiary() external {
        require(msg.sender == pendingBeneficiary, "Only proposed
beneficiary can accept");
        beneficiary = pendingBeneficiary;
        pendingBeneficiary = address(0);
    }
}
```

[M-02] Unsafe Vesting Parameter Validation

Severity:

Medium

Description:

The contract lacks comprehensive validation for critical vesting parameters including:

- Start time can be set to past timestamps
- `zeroReleaseMonths` not validated against total vesting duration
- `firstMonthRelease` not validated against total amount
- No checks for logical consistency between time-based parameters

Impact:

Invalid parameter combinations could lead to:

- Immediate token releases due to past start times
- Zero release period exceeding total vesting period
- First month release exceeding total amount
- Broken vesting schedules

Resolution:

Implement comprehensive parameter validation in the initialize function:

- Ensure start time is in the future
- Validate zero release months against total months
- Verify first month release against total amount
- Add logical consistency checks between all time-based parameters

```
function initialize(
    string calldata name,
    address beneficiary,
    uint256 startTime,
    uint256 totalMonths,
    uint256 totalAmount,
    address token,
    uint256 zeroReleaseMonths,
    uint256 firstMonthRelease
) public initializer {
    // Validate start time is in future
    require(startTime >= block.timestamp, "Start time must be in the
future");
```

```
    // Validate zero release months against total duration
    require(zeroReleaseMonths < _totalMonths, "Zero release exceeds total
months");

    // Use proper token decimals
    uint8 decimals = IERC20Metadata(token).decimals();
    uint256 scaledAmount = totalAmount * (10**decimals);
    uint256 scaledFirstMonth = firstMonthRelease * (10**decimals);

    // Validate first month release against total
    require(scaledFirstMonth < scaledAmount, "First month exceeds total");

    // ... rest of initialization ...
}
```

[L-01] Incorrect Token Decimals Handling

Severity:

Low

Description:

The contract assumes all tokens have 18 decimals by hardcoding `10 ** 18` for `totalAmount` and `firstMonthRelease` calculations. While most common ERC20 tokens use 18 decimals, some tokens may use different decimal places.

Impact:

- For tokens with non-standard decimals: amounts would need to be adjusted accordingly
- Requires manual verification of token decimals before deployment
- No direct financial impact as amounts can be adjusted during initialization

Resolution:

My assumption is this is meant to just be used with the Seraph Token, which has 18 decimals so its not necessarily an issue. But if the contract is reused in the future with other tokens, this could be an issue if they don't use 18 decimals like USDC and USDT.

[L-02] Floating Pragma

Severity:

Low

Description:

The contract uses a floating pragma `^0.8.0`, allowing compilation with any 0.8.x version.

Impact:

Different versions could lead to different bytecode being deployed on different chains or environments, potentially causing inconsistencies in behavior.

Resolution:

Lock the pragma to a specific version to ensure consistent deployment behavior.

[L-03] Missing Events for Critical Operations

Severity:

Low

Description:

The contract doesn't emit events for critical operations like beneficiary changes and initialization parameters.

Impact:

- Reduced off-chain tracking capabilities
- Difficulty in monitoring contract state changes
- Limited transparency for users and integrators

Resolution:

Add appropriate events for all critical state changes and operations.

Some examples include:

```
// Events for critical operations
event VestingInitialized(
    address indexed beneficiary,
    uint256 startTime,
    uint256 totalMonths,
    uint256 totalAmount,
    address indexed token,
    uint256 zeroReleaseMonths,
    uint256 firstMonthRelease
);

event TokensClaimed(
    address indexed beneficiary,
    uint256 amount,
    uint256 timestamp
);

event VestingRevoked(
    address indexed beneficiary,
    uint256 unreleasedAmount,
    uint256 timestamp
);
```

```
event VestingParametersUpdated(
    uint256 newTotalMonths,
    uint256 newZeroReleaseMonths,
    uint256 newFirstMonthRelease
);
```

[L-04] Use of Custom Errors

Severity:

Low

Description:

The contract uses require statements with string messages instead of custom errors, which is less gas-efficient and provides less detailed error information.

```
function initialize(
    string calldata name,
    address beneficiary,
    uint256 startTime,
    uint256 totalMonths,
    uint256 totalAmount,
    address token,
    uint256 zeroReleaseMonths,
    uint256 firstMonthRelease
) public initializer {
    // Validate start time is in future
    require(startTime >= block.timestamp, "Start time must be in the
future");

    // Validate zero release months against total duration
    require(zeroReleaseMonths < _totalMonths, "Zero release exceeds total
months");

    // Use proper token decimals
    uint8 decimals = IERC20Metadata(token).decimals();
    uint256 scaledAmount = totalAmount * (10**decimals);
    uint256 scaledFirstMonth = firstMonthRelease * (10**decimals);

    // Validate first month release against total
    require(scaledFirstMonth < scaledAmount, "First month exceeds total");

    // ... rest of initialization ...
}
```

Impact:

- Higher gas costs for revert cases

- Less descriptive error handling
- Reduced debugging capabilities

Resolution:

Replace require statements with custom errors for better gas efficiency and error handling.