Session Token Bug Bounty Program - Submission #1

Session Response

Hi [Name Redacted],

Thank you for taking the time to review our smart contracts and for submitting a detailed report with PoCs for the three issues you identified. We have now reviewed these reported issues with our developers. Based on our assessment, none of the issues submitted can be classed as vulnerabilities, and we are unfortunately unable to offer you a USDC reward. However, we are very grateful for your submission nonetheless, and would love to send you some Session merch to thank you for your efforts. Please let us know if you would be interested in your reply!

Please find our evaluation of these issues below, including our findings and assessment of the severity of the reported issues.

Issue 1

Your PoC code for Issue 1 contains a compilation error. Once this error is corrected and the PoC is run, the EVM reverts with the following message:

```
Unset
Failing tests:
Encountered 1 failing test in test/TxOriginPhishing.t.sol:TxOriginPhishingTest
[FAIL. Reason: revert: Only the operator can perform this action.] testTxOriginPhishing() (gas: 16206)
```

This shows that our protections prevent the attack as implemented in the PoC.
More broadly, in our design, we never instantiate the multi-contribution contract directly. Anyone creating a ServiceNodeContribution contract (as done in the PoC) directly won't appear in our staking portal's open SN list because they are not discoverable.

To discover a multi-contribution contract, you must create one through our factory, to which our staking portal subscribes for emitted events:
https://github.com/oxen-io/eth-sn-contracts/blob/2382076ac5da1cb3efea756088b14b9ff8e60d27/contracts/ServiceNodeContributionFactory.sol#L26

Our staking portal only allows the creation of multi-contribution contracts through the factory. When this is done, the wallet that initiated the transaction becomes the operator of the node, i.e., tx.origin. If

we used msg.sender, the owner of the contract would be the factory since the factory creates the multi-contribution contract for us.

Assessment

Therefore, we assess that this is not a vulnerability unless a working PoC can be provided that shows a case where this could be exploited in practice with contracts launched via the factory. Please inform us if you intend to follow up with a working PoC for such a scenario.

Issue 2

The first described attack requires collusion between the revoker and the block producer. In our deployment, the revoker would be the token issuer, who has signed legal agreements with all vesting parties, binding them to revoke tokens only in very specific legal cases. The block producer would be the Arbitrum sequencer.

Since we use Arbitrum, timestamps are derived from the sequencer, which is centralized. To attack this contract, one would need to take over the Arbitrum sequencer. Arbitrum Documentation https://docs.arbitrum.io/build-decentralized-apps/arbitrum-vs-ethereum/block-numbers-and-time#block-timestamps-arbitrum-vs-ethereum states that the sequencer can set timestamps up to 1 hour into the future. This means an attacker could end vesting early by N hours if they repeatedly get Arbitrum to sequence blocks with +1 hour timestamps. However, this requires manipulating the sequencer. An attacker could also go back 24 hours, but only if the previous block was delayed by 24 hours. If the sequencer is blocked from producing blocks, time and block production slow down, delaying vesting if block numbers or oracles are used.

Given that this attack requires compromising both the Arbitrum sequencer and the revoker (a legally bound entity), and that in the worst-case scenario, this attack only allows the revoker to revoke tokens 24 hours after they are claimable or delay vesting by a maximum of 24 hours, we consider this to be an unrealistic exploit. If it did occur, the effects would be minimal and would expose the revoker to significant legal consequences.

Assessment

Therefore, we assess that this is not a vulnerability due to the very high requirements to exploit and very low severity outcomes if exploited. We do not plan to deploy any fixes related to the reported issue.

Issue 3

You are correct in your assessment that there is no replay attack protection for the replay of BLS signatures. However, upon examining the proposed attacks on the following three functions, we find the following:

updateRewardsBalance

It is possible to replay signatures to update the rewards balance of a recipient. However, this cannot lead to a recipient claiming more tokens than they are owed. This is because the BLS signature signs the total amount of rewards owed over the lifetime of the recipient's node operation or contribution activities, as signed by the collective Service Node network. The amount claimed from their total lifetime rewards is tracked in the contract separately as an incrementing total, increasing each time they submit a claim. Thus, replaying an updateRewardsBalance transaction can only update the same value or an increased value as signed by the Service Node network. If a user were to claim all rewards, then replay an updateRewardsBalance transaction and attempt to claim more rewards, they would be prevented by contract checks that track lifetime claimed rewards and ensure that claims do not exceed the difference between lifetime claimed rewards and lifetime rewards earned.

removeBLSPublicKeyWithSignature & liquidateBLSPublicKeyWithSignature

You are also correct in your assessment that calls to liquidate and remove BLS keys can be replayed. Hence, the Service Node network includes a timestamp in these signatures, with expiry currently set to 10 minutes.

Within this 10-minute window, an attacker can replay the liquidate/remove BLS key transaction. However, this will only have a denial of service effect if the person the attacker is targeting was removed from the contract and then restaked or contributed to the same contract within 10 minutes of their original removal. In this case, the person would be removed from the contract errantly, but could simply wait a few more minutes and recontribute or restake.

Assessment
Thus, we assess that unless a PoC is provided which shows how a user can claim more than their lifetime owed rewards as signed by the Service Node network (the source of truth), we do not consider there to be a vulnerability in the replay of updateRewardsBalance transactions.

Our assessment of vulnerabilities reported relating to the replay of removeBLSPublicKeyWithSignature and liquidateBLSPublicKeyWithSignature functions is that we do consider these to be areas for improvement. However, the severity of these replay attacks is, in our view, very low, and the likelihood of them being used in practice, considering the lack of financial incentives, is also very low, thus we do not consider this a vulnerability. We may deploy updates to further limit the time in which signatures for these functions can be replayed, or implement frontend limitations to enhance user experience.

Summary

Thank you again for the time and effort you invested in investigating our contracts. A copy of the original full report from [Name redacted] can be found below

Bug bounty rewards are at the sole discretion of OPTF, as outlined in the Bug Bounty Terms, which you have agreed to by participating in the Bug Bounty Program.

Original Submission

Hello session team please my name is [Name redacted] and I would like to report a security vulnerability according to the bug bounty program.

Issue 1

tx.origin Phishing Vulnerability.

Vulnerable Contract:

ServiceNodeContribution.sol

Vulnerable Line of Code:

operator = tx.origin;

Detailed Description:

The ServiceNodeContribution contract contains a critical vulnerability related to the use of tx.origin in the constructor to set the operator address. This practice exposes the contract to phishing attacks, where an attacker can trick the operator into executing a transaction through a malicious contract. In such a scenario, tx.origin will be the operator's address, but msg.sender will be the attacker's contract, leading to unauthorized control over the contract.

Impact:

An attacker can exploit this vulnerability to gain unauthorized control over the contract by tricking the operator into interacting with a malicious contract.

This can result in:

Unauthorized access to critical functions restricted to the operator.

Potential financial loss if the attacker can manipulate contributions or withdrawals.

Severity: High

Given the potential for unauthorized control and financial loss, this vulnerability is classified as high severity.

Proof of Concept (PoC) Using Foundry:

Setup Foundry

Environment: Ensure you have Foundry installed. If not, install it using:

curl -L https://foundry.paradigm.xyz | bash

foundryup

Create a New Foundry Project:

forge init

txorigin-phishing-poc
cd txorigin-phishing-poc

Add the Vulnerable

Contract: Create a file src/ServiceNodeContribution.sol

and add the vulnerable contract code.

Create the Malicious Contract:

Create a file src/MaliciousContract.sol:

```
Unset
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "./ServiceNodeContribution.sol";

contract MaliciousContract {
    ServiceNodeContribution public target;

    constructor(address _target) {
        target = ServiceNodeContribution(_target);
    }

    function attack() public {
        // This will set the operator to the attacker's address
        target.contributeOperatorFunds(1, IServiceNodeRewards.BLSSignatureParams(0, 0, 0, 0));
    }
}
```

Write the Test: Create a file test/TxOriginPhishing.t.sol:

```
Unset
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../src/ServiceNodeContribution.sol";
import "../src/MaliciousContract.sol";

contract TxOriginPhishingTest is Test {
    ServiceNodeContribution public target;
    MaliciousContract public malicious;

    address operator = address(0x1);
    address attacker = address(0x2);

    function setUp() public {
        vm.startPrank(operator);
        target = new ServiceNodeContribution(
            address(0x3), // _stakingRewardsContract
            10, // _maxContributors
            BN256G1.G1Point(0, 0), // _blsPubkey
            IServiceNodeRewards.ServiceNodeParams(0, 0, 0) // _serviceNodeParams
        );
```

```
        vm.stopPrank();

        vm.startPrank(attacker);
        malicious = new MaliciousContract(address(target));
        vm.stopPrank();
    }

    function testTxOriginPhishing() public {
        vm.startPrank(attacker);
        malicious.attack();
        vm.stopPrank();

        // Check if the operator is now the attacker
        assertEq(target.operator(), attacker);
    }
}
```

Run the Test:

forge test

The test should fail, indicating that the operator has been set to the attacker's address, demonstrating the vulnerability.

Mitigation:

Replace tx.origin with msg.sender in the constructor to ensure that the operator is correctly set to the address that directly deployed the contract.

Updated Constructor:

```
Unset
constructor(
```

```
    address _stakingRewardsContract,
    uint256 _maxContributors,
    BN256G1.G1Point memory _blsPubkey,
    IServiceNodeRewards.ServiceNodeParams memory _serviceNodeParams
) nzAddr(_stakingRewardsContract) nzUint(_maxContributors) {
    stakingRewardsContract = IServiceNodeRewards(_stakingRewardsContract);
    stakingRequirement = stakingRewardsContract.stakingRequirement();
    SENT = IERC20(stakingRewardsContract.designatedToken());

    maxContributors = _maxContributors;
    operator = msg.sender; // Use msg.sender instead of tx.origin
    blsPubkey = _blsPubkey;
    serviceNodeParams = _serviceNodeParams;
}
```

Issue 2.

Timestamp Manipulation Vulnerability in Vesting Schedule

Severity: high-medium

Description:

The TokenVestingStaking contract relies on block.timestamp to determine the start and end of the vesting period.

This introduces a vulnerability where miners can manipulate the block timestamp within a certain range (typically up to 15 seconds).

This manipulation can affect the precise timing of vesting-related actions, such as release, revoke, and claimRewards.

Vulnerable Code:

```
Unset
uint256 public immutable start;
uint256 public immutable end;

// Constructor
constructor(
    address beneficiary_,
    address revoker_,
    uint256 start_,
    uint256 end_,
    bool transferableBeneficiary_,
    IServiceNodeRewards stakingRewardsContract_,
    IERC20 sent_
) nzAddr(beneficiary_) nzAddr(address(stakingRewardsContract_)) nzAddr(address(sent_)) {
    require(start_ <= end_, "block.timestamp: start_ after end_");
    require(block.timestamp < start_, "Vesting: start before current time");

    beneficiary = beneficiary_;
    revoker = revoker_;
    start = start_;
    end = end_;
    transferableBeneficiary = transferableBeneficiary_;
    stakingRewardsContract = stakingRewardsContract_;
    SENT = sent_;
}

// Revoke function
function revoke(IERC20 token) external override onlyRevoker notRevoked {
    require(block.timestamp <= end, "Vesting: vesting expired");

    uint256 balance = token.balanceOf(address(this));
    uint256 unreleased = _releasableAmount(token);
    uint256 refund = balance - unreleased;
    revoked = true;

    emit TokenVestingRevoked(token, refund);
    token.safeTransfer(revoker, refund);
}
```

Impact:

Revoke Exploitation: A miner acting as the revoker can manipulate the block timestamp to revoke the vesting just before the vesting period ends, reclaiming funds that should have been vested to the beneficiary.

Release Timing: The beneficiary may be unable to release tokens at the expected time if a miner manipulates the timestamp to delay the block.

Claim Rewards: The beneficiary's ability to claim rewards can be affected by timestamp manipulation, potentially delaying or preventing the claim.

Severity: Medium

Likelihood: high -Medium

- Timestamp manipulation by miners is feasible but requires control over block production.

Impact: Medium - The ability to revoke funds or delay vesting actions can significantly affect the beneficiary's expected outcomes.

Proof of Concept (PoC):

Below is a Foundry test script to demonstrate the timestamp manipulation vulnerability.

```
Unset
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../contracts/TokenVestingStaking.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockERC20 is ERC20 {
    constructor() ERC20("Mock Token", "MTK") {
        _mint(msg.sender, 1000000 * 10 ** 18);
    }
}

contract TokenVestingStakingTest is Test {
    TokenVestingStaking public vesting;
```

```
    MockERC20 public token;
    address public beneficiary = address(1);
    address public revoker = address(2);
    uint256 public start;
    uint256 public end;

    function setUp() public {
        token = new MockERC20();
        start = block.timestamp + 1 days;
        end = start + 30 days;

        vesting = new TokenVestingStaking(
            beneficiary,
            revoker,
            start,
            end,
            true,
            IServiceNodeRewards(address(0)),
            IERC20(address(token))
        );

        token.transfer(address(vesting), 1000 * 10 ** 18);
    }

    function testTimestampManipulation() public {// Fast forward to just before the end of the vesting period
        vm.warp(end - 1);

        // Attempt to revoke just before the vesting period ends
        vm.prank(revoker);
        vesting.revoke(IERC20(address(token)));

        // Check if the revoker successfully reclaimed the funds
        uint256 revokerBalance = token.balanceOf(revoker);
        assertEq(revokerBalance, 1000 * 10 ** 18, "Revoker should have reclaimed the funds");
    }
}
```

Steps to Reproduce:

Deploy the TokenVestingStaking contract with a start and end time.

Transfer tokens to the vesting contract.

Fast forward the block timestamp to just before the end of the vesting period.

Call the revoke function as the revoker.

Verify that the revoker successfully reclaims the funds.

Mitigation:

Use Block Numbers:

Instead of relying on block.timestamp, use block numbers for time-based calculations.

This reduces the risk of manipulation since block numbers are sequential and not subject to miner control.

Time Buffer: Introduce a buffer period to account for potential timestamp manipulation.

For example, add a grace period before the end of the vesting period during which revocation is not allowed.

Oracle: Use a trusted time oracle to provide more accurate time data.

This adds an additional layer of security by relying on an external source for time information.

Issue 3.

Signature Replay Vulnerability.

Vulnerable Functions.

```
Unset
updateRewardsBalance:

function updateRewardsBalance(
    address recipientAddress,
    uint256 recipientRewards,
    BLSSignatureParams calldata blsSignature,
```

```
    uint64[] memory ids
) external whenNotPaused whenStarted hasEnoughSigners(ids.length) {
    if (recipientAddress == address(0)) {
        revert NullRecipient();
    }

    if (recipients[recipientAddress].rewards >= recipientRewards) {
        revert RecipientRewardsTooLow();
    }

    // NOTE: Validate signature
    {
        bytes memory encodedMessage = abi.encodePacked(rewardTag, recipientAddress, recipientRewards);
        BN256G2.G2Point memory Hm = BN256G2.hashToG2(encodedMessage, hashToG2Tag);
        validateSignatureOrRevert(ids, blsSignature, Hm);
    }

    uint256 previousBalance = recipients[recipientAddress].rewards;
    recipients[recipientAddress].rewards = recipientRewards;
    emit RewardsBalanceUpdated(recipientAddress, recipientRewards, previousBalance);
}
removeBLSPublicKeyWithSignature:

function removeBLSPublicKeyWithSignature(
    BN256G1.G1Point calldata blsPubkey,
    uint256 timestamp,
    BLSSignatureParams calldata blsSignature,
    uint64[] memory ids
) external whenNotPaused whenStarted hasEnoughSigners(ids.length) {
    bytes memory pubkeyBytes = BN256G1.getKeyForG1Point(blsPubkey);
    uint64 serviceNodeID = serviceNodeIDs[pubkeyBytes];
    if (signatureTimestampHasExpired(timestamp)) {
        revert SignatureExpired(serviceNodeID, timestamp, block.timestamp);
    }

    if (
        blsPubkey.X != _serviceNodes[serviceNodeID].pubkey.X || blsPubkey.Y != _serviceNodes[serviceNodeID].pubkey.Y
    ) revert BLSPubkeyDoesNotMatch(serviceNodeID, blsPubkey);

    // NOTE: Validate signature
    {
        bytes memory encodedMessage = abi.encodePacked(removalTag, blsPubkey.X, blsPubkey.Y, timestamp);
        BN256G2.G2Point memory Hm = BN256G2.hashToG2(encodedMessage, hashToG2Tag);
        validateSignatureOrRevert(ids, blsSignature, Hm);
    }

    _removeBLSPublicKey(serviceNodeID, _serviceNodes[serviceNodeID].deposit);
}
liquidateBLSPublicKeyWithSignature:
```

```
function liquidateBLSPublicKeyWithSignature(
    BN256G1.G1Point calldata blsPubkey,
    uint256 timestamp,
    BLSSignatureParams calldata blsSignature,
    uint64[] memory ids
) external whenNotPaused whenStarted hasEnoughSigners(ids.length) {
    bytes memory pubkeyBytes = BN256G1.getKeyForG1Point(blsPubkey);
    uint64 serviceNodeID = serviceNodeIDs[pubkeyBytes];
    if (signatureTimestampHasExpired(timestamp)) {
        revert SignatureExpired(serviceNodeID, timestamp, block.timestamp);
    }

    ServiceNode memory node = _serviceNodes[serviceNodeID];
    if (blsPubkey.X != node.pubkey.X || blsPubkey.Y != node.pubkey.Y) {
        revert BLSPubkeyDoesNotMatch(serviceNodeID, blsPubkey);
    }

    // NOTE: Validate signature
    {
        bytes memory encodedMessage = abi.encodePacked(liquidateTag, blsPubkey.X, blsPubkey.Y, timestamp);
        BN256G2.G2Point memory Hm = BN256G2.hashToG2(encodedMessage, hashToG2Tag);
        validateSignatureOrRevert(ids, blsSignature, Hm);
      }

      // Calculating how much liquidator is paid out
      emit ServiceNodeLiquidated(serviceNodeID, node.operator, node.pubkey);
      uint256 ratioSum = _poolShareOfLiquidationRatio + _liquidatorRewardRatio + _recipientRatio;
      uint256 deposit = node.deposit;
      uint256 liquidatorAmount = (deposit * _liquidatorRewardRatio) / ratioSum;
      uint256 poolAmount = deposit * _poolShareOfLiquidationRatio == 0
          ? 0
          : (_poolShareOfLiquidationRatio - 1) / ratioSum + 1;

      _removeBLSPublicKey(serviceNodeID, deposit - liquidatorAmount - poolAmount);

      // Transfer funds to pool and liquidator
      if (_liquidatorRewardRatio > 0) SafeERC20.safeTransfer(designatedToken, msg.sender, liquidatorAmount);
      if (_poolShareOfLiquidationRatio > 0)
          SafeERC20.safeTransfer(designatedToken, address(foundationPool), poolAmount);
  }
```

Detailed Description.

The contract relies on BLS signatures for critical operations such as updating rewards, removing service nodes, and liquidating service nodes.

However, it does not use nonces or other mechanisms to ensure that each signature is unique and can only be used once.

This oversight allows an attacker to replay a valid signature within the expiry window, causing unintended and potentially harmful actions.

Impact.

updateRewardsBalance:

Impact: An attacker could repeatedly update the rewards balance, leading to unauthorized reward claims.

Severity: High

removeBLSPublicKeyWithSignature:

Impact: An attacker could repeatedly remove a service node, disrupting the network and causing denial of service.

Severity: High

liquidateBLSPublicKeyWithSignature:

Impact: An attacker could repeatedly liquidate a service node, causing financial loss to the node operator and potentially destabilizing the network.

Severity: Critical

Overall Severity: Critical

The ability to replay signatures can lead to significant financial loss, network disruption, and unauthorized actions.

Unset

Proof of Concept (PoC) using Forge/Foundry

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "./ServiceNodeRewards.sol";

contract SignatureReplayTest is Test {
    ServiceNodeRewards serviceNodeRewards;
    address token = address(0x123);
    address foundationPool = address(0x456);
    uint256 stakingRequirement = 1000;
    uint256 maxContributors = 10;
    uint256 liquidatorRewardRatio = 10;
    uint256 poolShareOfLiquidationRatio = 10;
    uint256 recipientRatio = 80;

    function setUp() public {
        serviceNodeRewards = new ServiceNodeRewards();
        serviceNodeRewards.initialize(
            token,
            foundationPool,
            stakingRequirement,
            maxContributors,
            liquidatorRewardRatio,
            poolShareOfLiquidationRatio,
            recipientRatio
        );
    }

    function testSignatureReplay() public {
        // Assume we have a valid BLS signature and other parameters
        BN256G1.G1Point memory blsPubkey = BN256G1.G1Point(1, 2);
        BLSSignatureParams memory blsSignature = BLSSignatureParams(1, 2, 3, 4);
        uint64[] memory ids = new uint64[](0);
        uint256 timestamp = block.timestamp;

        // First call to removeBLSPublicKeyWithSignature
        serviceNodeRewards.removeBLSPublicKeyWithSignature(blsPubkey, timestamp, blsSignature, ids);

        // Replay the samesignature within the expiry window
        vm.expectRevert("BLSPubkeyDoesNotMatch");
        serviceNodeRewards.removeBLSPublicKeyWithSignature(blsPubkey, timestamp, blsSignature, ids);
    }
}
```

Detailed Description

The provided PoC demonstrates the vulnerability by calling the removeBLSPublicKeyWithSignature function twice with the same signature and timestamp.

The second call should ideally fail due to a nonce or similar mechanism, but in the current implementation, it does not, highlighting the replay vulnerability.

Impact

Unauthorized Reward Claims:

Attackers can repeatedly claim rewards, leading to financial losses for the contract and its participants.

Network Disruption:

Repeated removal of service nodes can disrupt the network's operation, causing denial of service.

Financial Loss:

Repeated liquidation of service nodes can cause significant financial losses to node operators and destabilize the network.

Severity

Overall Severity: Critical
The replay vulnerability can lead to significant financial loss, unauthorized actions, and network disruption.

Recommendations.

Implement Nonce Mechanism:

Introduce a nonce for each signature to ensure uniqueness and prevent replay attacks.

Enhance Signature Expiry:

Consider reducing the expiry window or adding additional checks to mitigate the risk of replay within the window.

Audit and Testing:

Conduct thorough audits and testing to identify and address similar vulnerabilities in other parts of the contract.