Zellic

Prepared for
Kee Jefferys
Session

Prepared by
Katerina Belotskaia
Nipun Gupta
Zellic

October 23, 2024

Session Token

Smart Contract Patch Review

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security patch review for Session team from October 9th to October 14th, 2024. During this engagement, Zellic reviewed Session Token's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following question:

- Were any bugs introduced during recent optimizations, refactoring, or updates?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

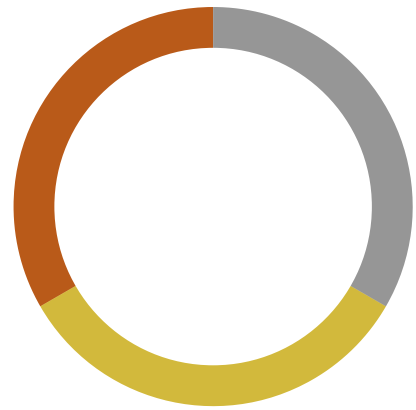
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Session Token contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	1



2. Introduction

2.1. About Session Token

The Session team contributed the following description of Session and Session Token:

> Session is an end-to-end encrypted, decentralised messaging application with **over 1,000,000 monthly active users**. It's a messaging app like no other —– with no phone number required on sign-up, onion-routed messaging, and a decentralised server structure. Session Token powers the Session application and is used to subscribe to Session Pro, make registrations using the Session Name Service and staked to run Session Nodes.

We were asked to review several patches to Session Token contracts for optimization, updates, and refactoring purposes. The purpose of this review was to focus exclusively on changes to the Session Token contracts, since our last review, evaluating them for any potential security vulnerabilities or inconsistencies. In section 4. ↗, we have provided an overview of the changes.

2.2. Scope

The engagement involved a review of the following targets:

Session Token Contracts

Type	Solidity
Platform	EVM-compatible

Target	eth-sn-contracts
Repository	https://github.com/oxen-io/eth-sn-contracts ↗
Version	`484a3231c7080c4b724776cc8cbb50db2e96eaa2`

Programs	`ServiceNodeContribution`
	`ServiceNodeContributionFactory`
	`ServiceNodeRewards`
	`TokenVestingStaking`

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io

Nipun Gupta
Engineer
nipun@zellic.io

3. Detailed Findings

3.1. Inconsistent status after withdrawing a contribution

Target	ServiceNodeContribution		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In the `_contributeFunds` function, when the staking requirements are met and `manualFinalize` is not set, the status is set to `Status.WaitForFinalized`, awaiting the operator to call the `finalize` function.

```
function _contributeFunds(address caller, address beneficiary, uint256 amount)
    private {
    [...]
    if (currTotalContribution == stakingRequirement) {
        emit Filled(_serviceNodeParams.serviceNodePubkey, operator);
        status = Status.WaitForFinalized;
    }

    // NOTE: Transfer funds from sender to contract
    emit NewContribution(caller, amount);
    SENT.safeTransferFrom(caller, address(this), amount);

    // NOTE: Auto finalize the node if valid
    if (status == Status.WaitForFinalized && !manualFinalize) {
        _finalize();
    }
}
```

However, during this waiting period, a user can still remove their contributions, as the `withdrawContribution` function has no restrictions during this stage. This leads to a situation where the contract remains in the `Status.WaitForFinalized` status, even though the staking requirement is no longer met after the withdrawal.

```
function withdrawContribution() external {
    if (msg.sender == operator) {
        _reset();
        return;
    }
```

```
    uint256 timeSinceLastContribution = block.timestamp
    - contributionTimestamp[msg.sender];
    if (timeSinceLastContribution < WITHDRAWAL_DELAY)
        revert WithdrawTooEarly(contributionTimestamp[msg.sender],
    block.timestamp, WITHDRAWAL_DELAY);

    uint256 refundAmount = removeAndRefundContributor(msg.sender);
    if (refundAmount > 0)
        emit WithdrawContribution(msg.sender, refundAmount);
}
```

Impact

The `finalize` function remains available for calling, even if the staking requirements are not met. This allows the operator to attempt using the current collected contributions to add a new service node. However, even if the contract balance is sufficient, the `addBLSPublicKey` of the ServiceNodeRewards contract, which is called during node registration, will revert. This is because the `addBLSPublicKey` function performs a check to ensure that the total contributions from the current contributors match the `stakingRequirement`.

```
function addBLSPublicKey(
    BN256G1.G1Point memory blsPubkey,
    BLSSignatureParams memory blsSignature,
    ServiceNodeParams memory serviceNodeParams,
    Contributor[] memory contributors
) external whenNotPaused whenStarted {
    [...]
    for (uint256 i = 0; i < contributors.length; i++)
        totalAmount += contributors[i].stakedAmount;
    if (totalAmount != stakingRequirement)
        revert ContributionTotalMismatch(stakingRequirement, totalAmount);
    [...]
}
```

As the status is still `Status.WaitForFinalized`, new contributors cannot add funds via the `contributeFunds` call, because this function reverts if the current status is not `Status.WaitForOperatorContrib` or `Status.OpenForPublicContrib`. Thus, the only way to fix the status would be to reset all the current contributions and request all the contributors to contribute again. A malicious contributor could thus carry out the attack numerous times, leading to bad user experience for the other contributors and the contribution not ever being staked.

Recommendations

We recommend changing the status back to `Status.OpenForPublicContrib` in the `withdrawContribution` call if the current status is `Status.WaitForFinalized` and the withdraw amount decreases the contribution from `stakingRequirement`.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit 1d72b183 ↗.

3.2. Misleading beneficiary update after finalization

Target	ServiceNodeContribution		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `updateBeneficiary` function allows a contributor to change the current beneficiary address that receives the reward. This function updates the `Staker` object in the `_contributorAddresses` mapping, which is associated with the contributor.

```
function updateBeneficiary(address newBeneficiary) external {
    _updateBeneficiary(msg.sender, newBeneficiary); }

function _updateBeneficiary(address stakerAddr, address newBeneficiary)
    private {
    address desiredBeneficiary = newBeneficiary == address(0) ? stakerAddr :
    newBeneficiary;
    address oldBeneficiary = address(0);
    bool updated = false;
    uint256 length = _contributorAddresses.length;
    for (uint256 i = 0; i < length; i++) {
        IServiceNodeRewards.Staker storage staker = _contributorAddresses[i];
    [...]
        oldBeneficiary = staker.beneficiary;
        staker.beneficiary = desiredBeneficiary;
        break;
    }
    [...]
}
```

Impact

However, the issue arises if the `updateBeneficiary` function is called during `Status.Finalized`, because all information about contributors, including the beneficiary addresses, has already been provided to the ServiceNodeRewards contract for new node registration. Any beneficiary address updated after finalization will not be applied to the node, as the ServiceNodeRewards contract does not permit updates to the beneficiary information after registration.

Recommendations

We recommend restricting the `updateBeneficiary` function during the `Status.Finalized` phase to avoid misleading stakers that the beneficiary addresses has been updated.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit 4d87e5cc ↗.

3.3. Some functions can be implemented more efficiently

Target	Multiple contracts		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Below we provide our recommendations for gas optimizations.

1) In several functions, the length of an array can be cached to save gas in loops. The list of functions where this optimization can be applied is given below:

In the ServiceNodeContribution contract

- `reserved.length` in the `_updateReservedContributors` function

In the ServiceNodeRewards contract

- `contributors.length` in the `addBLSPublicKey` function
- `_serviceNodes[serviceNodeID].contributors.length` in the `_initiateRemoveBLSPublicKey` function
- `nodes.length` and `node.contributors.length` in the `seedPublicKeyList` function

Also, the index incrementing for the loop can be made unchecked to optimize gas usage.

2) The `_contributeFunds` function can be updated to avoid calling `_updateBeneficiary` when the staker is being added for the first time, as shown in the example below:

```
if (contributions[caller] == 0)
    _contributorAddresses.push(IServiceNodeRewards.Staker(caller, caller));

    _contributorAddresses.push(IServiceNodeRewards.Staker(caller, beneficiary)
        );
}
else {
    _updateBeneficiary(caller, beneficiary);
}
```

3) The `_reset()` function can be optimized. When `status == Status.Finalized`, it is unnecessary to call `removeAndRefundContributor` for each address in the `_contributorAddresses` array, since in this case, the entire array should simply be deleted and the contributions and `contribu-`

`tionTimestamp` should be set to zero. If the state is not `Finalized`, there is also no need to call `removeAndRefundContributor` for every element in the `_contributorAddresses` just to delete a single element. Currently, the `removeAndRefundContributor` function searches for the staker's address in the `_contributorAddresses` array, moves elements, removes the last one, and then transfers funds. In case of `status == Status.Finalized`, we suggest iterating through the `_contributorAddresses` array to refund each staker, set the `contributions[staker]` and `contributionTimestamp[staker]` to zero, and finally delete the `_contributorAddresses` array. Finally, call the function `_updateReservedContributors` with zero contributors.

Impact

The contract is slightly less efficient.

Recommendations

Consider modifying these functions to optimize gas usage.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit c6f0c2bc ↗.

4. Patch Review

This section documents notable and minor changes applied to the in-scope code from commit e66df039 ↗ to commit 13893770 ↗ (the latest commit of branch `master` at the time this engagement commenced).

4.1. Notable changes

The following were notable changes made to the codebase.

ServiceNodeContribution contract

The operator can assign a list of the reserved contributors, which includes their addresses and the amount they should contribute. However, this feature is optional, and an empty list is also acceptable. A newly added `reservedContributions` array contains this data, along with a flag indicating whether the funds have been received. Additionally, a new `reservedContributionsAddresses` array contains only the addresses of contributors. Reserved contributors are prioritized over other stakers, meaning that other stakers cannot contribute if the total contribution, considering both the awaiting contribution from reserved contributors and already provided contribution, exceeds the staking requirement.

New functions have been introduced for an account with the operator role to allow setup of the contract state. The `updateManualFinalize` function enables updating the `manualFinalize` boolean flag. The `updateFee` function allows to modify the `fee`; the new `fee` cannot exceed the constant `MAX_FEE`, which is set to `10000`. The `updatePubkeys` function allows to update `blsPubkey`, `_blsSignature`, `_serviceNodeParams.serviceNodePubkey`, `_serviceNodeParams.serviceNodeSignature1`, and `_serviceNodeParams.serviceNodeSignature2`, but only after proof-of-possession validation. The `updateReservedContributors` function allows to update the reserved contributors list. The number of new reserved contributors cannot exceed the immutable `maxContributors`. All these functions, except `updateManualFinalize`, are available to the operator only during the `WaitForOperatorContrib` phase.

The contribution process was modified to introduce the reserved contributors, specified by the operator. As described earlier, the stakers in the `reservedContributions` array have priority over other stakers. This means that the nonreserved stakers can only contribute the remaining portion, excluding the amount that is still awaiting from the reserved contributors.

```
if ((currTotalContribution + currReservedContribution) > stakingRequirement)
            revert
    ContributionExceedsStakingRequirement(currTotalContribution,
    currReservedContribution, stakingRequirement);
```

Additionally, there are specific requirements for the reserved contributors. First, the contributor must be the operator. Second, the contribution from any staker in the `reservedContributions` array must not be less than the amount specified for them.

```
function _contributeFunds(address caller, address beneficiary,
    uint256 amount) private {
    [...]
        if (status == Status.WaitForOperatorContrib) {
            if (caller != operator)
                revert FirstContributionMustBeOperator(caller, operator);
            status = Status.OpenForPublicContrib;
            emit
    OpenForPublicContribution(_serviceNodeParams.serviceNodePubkey, operator,
    _serviceNodeParams.fee);
        }
    [...]
        ReservedContribution storage reserved = reservedContributions[caller];
        if (reserved.amount > 0 && !reserved.received) {
            // NOTE: Check amount is sufficient
            if (amount < reserved.amount)
                revert ContributionBelowReservedAmount(amount,
    reserved.amount);
    [...]
}
```

Another modification involves the manual finalization step. Previously, when the total contribution reached the `stakingRequirement` amount, the finalization was executed automatically. Now, the operator can choose whether the contract automatically adds the service node to the ServiceNodeRewards or initiates this process manually using the `finalize` function. In the case of manual finalization, when the staking requirement is met, the contract's status changes to `WaitForFinalized`, and only then the `finalize` function becomes available for the operator to execute. Additionally, the operator has the option to reset the contract before finalization, refunding the contributors' funds.

```
function _contributeFunds(address caller, address beneficiary, uint256 amount)
    private {
   [...]

       if (currTotalContribution == stakingRequirement) {
           emit Filled(_serviceNodeParams.serviceNodePubkey, operator);
           status = Status.WaitForFinalized;
       }
   [...]
       // NOTE: Auto finalize the node if valid
       if (status == Status.WaitForFinalized && !manualFinalize) {
           _finalize();
       }
   }
```

In the previous version of the code, the `resetContract` function allowed the operator to reset the contract's state to reregister the node and provide new contributions. This function is no longer supported. Instead, the two new functions `resetUpdateAndContribute` and `resetUpdateFeeReservedAndContribute` were introduced — `resetUpdateAndContribute` enables the operator to reset all contract parameters and, if necessary, provide an initial contribution, while `resetUpdateFeeReservedAndContribute` enables the operator to reset the contract's state and update the fee, reserved contributors, and the `manualFinalize` flag. The initial contribution can also be provided.

```
function _resetUpdateAndContribute([...]) private {
    _reset();
    _updatePubkeys(key, sig, params.serviceNodePubkey,
    params.serviceNodeSignature1, params.serviceNodeSignature2);
    _updateFee(params.fee);
    _updateReservedContributors(reserved);
    _updateManualFinalize(_manualFinalize);
    if (amount > 0)
        _contributeFunds(operator, beneficiary, amount);
}

function resetUpdateFeeReservedAndContribute([...]) external onlyOperator {
    _reset();
    _updateFee(fee);
    _updateReservedContributors(reserved);
    _updateManualFinalize(_manualFinalize);
    if (amount > 0)
        _contributeFunds(operator, beneficiary, amount);
}
```

Additionally, a new `reset` function was introduced, which only resets the contract state without allowing an initial contribution. All these functions are available at any stage. If the current stage is not final, all contributed funds will be refunded to the stakers.

```
function _reset() private {
{
    IServiceNodeRewards.Staker[] memory copy = _contributorAddresses;
    uint256 length = copy.length;
    for (uint256 i = 0; i < length; i++)
        removeAndRefundContributor(copy[i].addr);
    delete _contributorAddresses;
}
status = Status.WaitForOperatorContrib;
{
    IServiceNodeRewards.ReservedContributor[] memory zero
    = new IServiceNodeRewards.ReservedContributor[](0);
    _updateReservedContributors(zero);
}
```

```
}
```

Stakers now have the ability to update their specified `beneficiary` address using the `updateBeneficiary` function. If the new `beneficiary` address is set to zero, the staker's own address will be used as the `beneficiary` address.

```
function _updateBeneficiary(address stakerAddr, address newBeneficiary)
    private {
    address desiredBeneficiary = newBeneficiary == address(0) ? stakerAddr :
    newBeneficiary;
    [...]
    for (uint256 i = 0; i < length; i++) {
        IServiceNodeRewards.Staker storage staker = _contributorAddresses[i];
        if (staker.addr != stakerAddr)
            continue;

        if (staker.beneficiary == desiredBeneficiary)
            return;

        updated = true;
        oldBeneficiary = staker.beneficiary;
        staker.beneficiary = desiredBeneficiary;
        break;
    }
    [...]
}
```

The `withdrawContribution` function has also been modified. If the caller is the operator, the contract state will be reset. Additionally, this function can now be called after the finalization stage, even when a new node has already been added. However, if the caller is not the operator, the function will return successfully without changing the contract state.

```
function withdrawContribution() external {
    if (msg.sender == operator) {
        _reset();
        return;
    }

    uint256 timeSinceLastContribution = block.timestamp
    - contributionTimestamp[msg.sender];
    if (timeSinceLastContribution < WITHDRAWAL_DELAY)
        revert WithdrawTooEarly(contributionTimestamp[msg.sender],
    block.timestamp, WITHDRAWAL_DELAY);

    uint256 refundAmount = removeAndRefundContributor(msg.sender);
```

```
    if (refundAmount > 0)
        emit WithdrawContribution(msg.sender, refundAmount);
}
```

Several new view functions have been introduced, including `blsSignature`, `serviceNodeParams`, `contributorAddresses`, `getContributions`, `getReserved`, and `totalReservedContribution`.

TokenVestingStaking contract

The `revoke` function has been updated, introducing a two-step revocation process. Previously, a caller with the revoker role could immediately withdraw all unreleased tokens before the vesting period ended and prevent the beneficiary from claiming tokens. Now, during the first `revoke` function call before the end of the vesting period, the revoker can only mark the contract as `revoked` and must wait until the vesting period ends to withdraw tokens. Once the vesting period has ended, the revoker can call the `revoke` function again to withdraw the full token balance. After the contract is marked as `revoked`, the release action becomes unavailable to the beneficiary.

```
function release(IERC20 token) external override onlyBeneficiary notRevoked {
    uint256 amount = releasableAmount(token);
    require(amount > 0, "Vesting: no tokens are due");
    emit TokensReleased(token, amount);
    token.safeTransfer(beneficiary, amount);
}

function revoke(IERC20 token) external override onlyRevoker {
    if (!revoked) { // Only allowed to revoke whilst in vesting period
        require(block.timestamp <= end, "Vesting: vesting expired");
        revoked = true;
        emit TokenVestingRevoked(token);
    }
    uint256 amount = releasableAmount(token);
    if (amount > 0) {
        emit TokensRevokedReleased(token, amount);
        token.safeTransfer(revoker, amount);
    }
}
```

The multicontribution feature has been introduced with the `contributeFunds` function, allowing contributions of partial token amounts to the trusted ServiceNodeContribution contract. The provided `ServiceNodeContribution` address is validated using the new `getContributionContract` function, which checks if the provided contract address was deployed by the current snContribFactory contract. If valid, the function returns address of the `ServiceNodeContribution`; otherwise, it reverts.

```
function contributeFunds(address snContribAddr,
                         uint256 amount,
                         address snContribBeneficiary
) external onlyRevokerIfRevokedElseBeneficiary afterStart
    nzAddr(snContribBeneficiary)
    {
    IServiceNodeContribution snContrib
    = getContributionContract(snContribAddr);
    SENT.approve(snContribAddr, amount);
    snContrib.contributeFunds(amount, snContribBeneficiary);
}
```

Since the revoker can halt the vesting process but cannot withdraw funds until the vesting period ends, several functions are now available to the revoker when the contract is in a `revoked` state. This access control is enforced via the `onlyRevokerIfRevokedElseBeneficiary` modifier, which is applied to the following functions, `initiateRemoveBLSPublicKey`, `addBLSPublicKey`, `claimRewards`, `contributeFunds`, `withdrawContribution`, and `updateBeneficiary`.

The `updateBeneficiary` function allows assigning a beneficiary to the ServiceNodeContribution contract.

The `withdrawContribution` function has been introduced, allowing the withdrawal of contributions made to the ServiceNodeContribution contract.

The `claimRewards` function has been updated to call the `claimRewards` function of the ServiceNodeRewards contract.

The `addBLSPublicKey` function has also been modified. The `investorServiceNodes` is no longer updated and has been removed from the contract. The first element of the `contributors` array is filled out by the information about `Staker`, using the contract address and the provided `snBeneficiary` address. Additionally, the necessary `stakingRequirement` amount will be approved from this contract to the ServiceNodeRewards contract to transfer tokens and add new nodes.

ServiceNodeRewards contract

The reward-claiming process has been modified with the introduction of a claim cycle. The duration of the cycle is controlled by the `claimCycle` variable. The maximum reward amount that can be claimed during each cycle is capped by the `claimThreshold` value. Once this value is exceeded, the `claimRewards` function becomes unavailable until the start of the next cycle. The `currentClaimTotal` variable tracks the total claims made in the current cycle and is reset at the beginning of each new cycle.

```
function _claimRewards(address claimingAddress, uint256 amount) internal {
    [...]
    // NOTE: Reset the total claims if we have entered a new cycle
    uint256 nextClaimCycle = block.timestamp / claimCycle;
```

```
    if (nextClaimCycle > currentClaimCycle) {
        currentClaimCycle = nextClaimCycle;
        currentClaimTotal = 0;
    }

    // NOTE: Accumulate the claims for the current cycle
    currentClaimTotal += amount;
    if (currentClaimTotal > claimThreshold) revert ClaimThresholdExceeded();
    [...]
}
```

The `claimCycle` duration can be updated by the contract owner via the `setClaimCycle` function, with no upper limit, although it cannot be set to zero. The initial duration is set to 12 hours. Similarly, the `claimThreshold` could be updated by the owner of the contract using the `setClaimThreshold` function, with no upper limit, but it also cannot be zero. The initial `claimThreshold` is set to `1_000_000 * 1e9`.

Additionally, the process for removing a BLSPublicKey using the `initiateRemoveBLSPublicKey` can be initiated by any staker of the service node. However, stakers with a deposit of less than 25% of the `stakingRequirement` can initiate the removal after a 30-day delay from when the node was added.

```
function _initiateRemoveBLSPublicKey(uint64 serviceNodeID, address caller)
    internal whenStarted {
    [...]
    for (uint256 i = 0; i < _serviceNodes[serviceNodeID].contributors.length;
    i++) {
        if (_serviceNodes[serviceNodeID].contributors[i].staker.addr ==
    caller) {
            isContributor = true;
            isSmall =
                SMALL_CONTRIBUTOR_DIVISOR
    * _serviceNodes[serviceNodeID].contributors[i].stakedAmount
                    < _serviceNodes[serviceNodeID].deposit;
            break;
        }
    }
    [...]
    if (isSmall && block.timestamp <
    _serviceNodes[serviceNodeID].addedTimestamp
    + SMALL_CONTRIBUTOR_LEAVE_DELAY)
        revert SmallContributorLeaveTooEarly(serviceNodeID, caller);
    [...]
}
```

ServiceNodeContributionFactory contract

The `deployedContracts` mapping was added to track the deployed ServiceNodeContribution contracts.

The `deployContributionContract` was renamed to `deploy`. Additionally, new arguments were introduced:

- `sig`, the BLS proof-of-possession signature
- `reserved`, the `ReservedContributor` array, which contains a list of reserved contributors and their staking amounts
- `manualFinalize`, a boolean flag that determines whether the contract will finalize automatically or not

All these parameters are passed to the `ServiceNodeContribution` constructor. The address of the created contract are added to the `deployedContracts`.

A new `owns` function was introduced, which checks if the provided contract was deployed by this factory.

4.2. Minor differences

The following were minor changes made to the codebase.

SENT contract

The ERC20Permit extension from OpenZeppelin has been added.

RewardRatePool contract

The `ANNUAL_INTEREST_RATE` was renamed to `ANNUAL_SIMPLE_PAYOUT_RATE`, and the value increased from 145 (14.5%) to 151 (15.1%).

The `calculateReleasedAmount` function was updated to no longer accept the `timestamp` as input as well as the `rewardRate` function. Instead, these functions now use the current timestamp `block.timestamp`.

The `calculateInterestAmount` function was renamed to `calculatePayoutAmount` with no other modifications.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Session Token contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.