Zellic

July 8, 2024

Prepared for
Kee Jefferys
Session team

Prepared by
Katerina Belotskaia
Hojung Han
Sylvain Pelissier
Zellic

Session Token

Smart Contract Security Assessment

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 assessment for Session team from May 21st to May 24th, 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 questions:

- Is the BLS signature verification implemented correctly?
- Are all payouts calculated correctly?
- Is there a way to bypass BLS signature verification?
- Can any actions be performed without the appropriate signature permit?
- Can the data about the service node ID be affected using the permit for the other service node ID?

1.3. Non-goals and Limitations

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

- The TokenVestingNoStaking.sol and Shared.sol contracts
- 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.

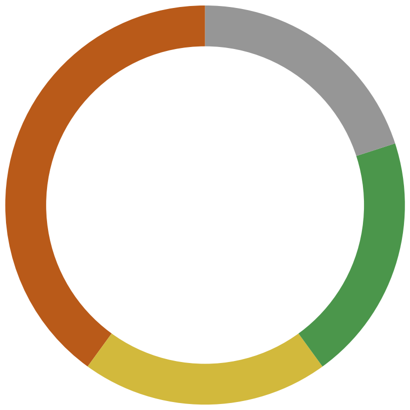
1.4. Results

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

Additionally, Zellic recorded its notes and observations from the assessment for Session team's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	2
🟨 Medium	1
🟩 Low	1
⬜ Informational	1



2. Introduction

2.1. About Session Token

Session team contributed the following description of Session Token:

> Session Token is the cryptocurrency driving the Session communications ecosystem. This EVM-compatible token can be used to unlock premium features within the Session private messaging application. Additionally, it serves as a security and incentivisation layer for the decentralised physical infrastructure network of Session Nodes that powers the app.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Session Token Contracts

Repository	https://github.com/oxen-io/eth-sn-contracts ↗
Version	eth-sn-contracts: `291f315b0978b0a580793d1d82996ab901ee3ba8`
Programs	• ServiceNodeRewards • SENT • ServiceNodeContribution • ServiceNodeContributionFactory • RewardRatePool • BN256G1 • BN256G2 • Pairing • TokenVestingStaking • TokenConverter
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of one and a quarter person-weeks. The assessment was conducted over the course of four calendar days.

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

Hojung Han
Engineer
hojung@zellic.io

Sylvain Pelissier
Engineer
sylvain@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 21, 2024	Kick-off call
May 21, 2024	Start of primary review period
May 24, 2024	End of primary review period

3. Detailed Findings

3.1. Potential vulnerability in `_aggregatePubkey` update mechanism

Target	ServiceNodeRewards		
Category	Protocol Risks	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

Upon examining the code where the `_aggregatePubkey` is updated through the `addBLSPublicKey` function, we have identified a potentially significant scenario that requires attention. Assuming that the DEX swap pair that includes SENT tokens has a large liquidity in the future, there exists a potential vulnerability.

Specifically, if a flash loan is used to acquire enough tokens to constitute more than a 2/3 majority, and these tokens are then staked, the `_aggregatePubKey` could be updated immediately.

The code involved is shown below.

```
function serviceNodeAdd(BN256G1.G1Point memory pubkey)
    internal returns (uint64 result) {
    [...]
    if (totalNodes == 1) {
        _aggregatePubkey = pubkey;
    } else {
        _aggregatePubkey = BN256G1.add(_aggregatePubkey, pubkey);
    }
    return result;
}
```

This scenario could enable arbitrary reward updates using the `updateRewardsBalance` function.

Impact

If exploited, this vulnerability could allow an attacker to perform arbitrary updates to the rewards balance. This could lead to manipulation of rewards and financial losses.

Recommendations

To mitigate this risk, it would be prudent to implement a minimum waiting period before updating the `_aggregatePubKey` after the execution of the `serviceNodeAdd` function, rather than allowing an immediate update. This would prevent the described exploit and ensure greater security and integrity of the system.

While the project team may monitor the difference between the liquidity of the DEX pair and the liquidity of the `ServiceNodeRewards` contract and take appropriate actions to avoid such scenarios, Session team should also consider the risks that could be prevented by human monitoring.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit `8cbd1fb7`↗. It was addressed by adding a limit that allows up to `max(5, 2% of totalNodes)` public keys to be aggregated into the `_aggregatePubkey` within a single block.

3.2. Lack of proof-of-possession verification

Target	ServiceNodeRewards		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

Before aggregating a BLS public key, a proof-of-possession is required by the contract. It consists in a signature of the domain separation tag, the public key together with the caller address, and the service-node public key. This allows proving to the contract that the service node knows the correct private key matching the public key submitted. It means that a node knows the private key x_i of its public key Q_i such that $Q_i = x_i \cdot G_1$, where G_1 is the generator of the group $\mathbb{G}_1$.

However, the function `seedPublicKeyList` does not use a proof-of-possession before adding a list of public keys to the aggregate key:

```
/// @notice Seeds the public key list with an initial set of service nodes.
///
/// @dev This should be called before the hardfork by the foundation to
/// ensure the public key list is ready to operate.
///
/// @param pkX Array of X-coordinates for the public keys.
/// @param pkY Array of Y-coordinates for the public keys.
/// @param amounts Array of amounts that the service node has staked,
/// associated with each public key.
function seedPublicKeyList(
    uint256[] calldata pkX,
    uint256[] calldata pkY,
    uint256[] calldata amounts
) external onlyOwner {
    if (pkX.length != pkY.length || pkX.length != amounts.length) {
        revert ArrayLengthMismatch();
    }

    for (uint256 i = 0; i < pkX.length; i++) {
        BN256G1.G1Point memory pubkey = BN256G1.G1Point(pkX[i], pkY[i]);
        uint64 allocID = serviceNodeAdd(pubkey);
        _serviceNodes[allocID].deposit = amounts[i];
        emit NewSeededServiceNode(allocID, pubkey);
    }
    updateBLSNonSignerThreshold();
}
```

Therefore, a rogue key attack is possible. For an aggregate key $Q = \sum_i Q_i$, the owner of the contract can add a new rogue key $Q_r = x_r \cdot G - \sum_i Q_i$ to the aggregate key. Then the new aggregate key verifies signatures signed only by their key x_r but not the other previous keys. This means that messages signed only by their key will be seen as valid even if they were not signed by the other public keys.

Impact

Even if this function is callable only by the owner of the contract, it allows the owner to add a rogue key afterwards and have the signature that is signed only by them seen as valid.

Recommendations

Ensure proof-of-possession before all aggregate key modification.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit `9c97d798` ↗.

Session team's official response is paraphrased below:

> The function `seedPublicKeyList` reverts if the contract has already started, when the variable `isStarted` is set to `true`. The keys passed as parameter before the contract started, will be handled out-of-band via our C++ code that will be responsible for checking the proof-of-possession.

3.3. Incorrect curve mapping

Target	BN256G2		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In the original BLS signature paper ↗, the authors construct a hash function called **MapToGroup**$_{h'}$ in order to hash a message to an arbitrary point of $\mathbb{G}_2$. This point is later used by the signature algorithms to create the signature. The function construction from the paper ensures that the security proof in the random oracle model holds. It is built such that the hash function is indistinguishable from a random hash function mapping a message to a point.

The function `mapToG2` is implementing such algorithm in the code. However, the construction slightly differs from the original paper. Here is an implementation:

```
// hashes to G2 using the try and increment method
function mapToG2(uint256 h) internal view returns (G2Point memory) {
    // Define the G2Point coordinates
    uint256 x1 = h;
    uint256 x2 = 0;
    uint256 y1 = 0;
    uint256 y2 = 0;

    bool foundValidPoint = false;

    // Iterate until we find a valid G2 point
    while (!foundValidPoint) {
        // Try to get y^2
        (uint256 yx, uint256 yy) = Get_yy_coordinate(x1, x2);

        // Calculate square root
        (uint256 sqrt_x, uint256 sqrt_y) = FQ2Sqrt(yx, yy);

        // Check if this is a point
        if (sqrt_x != 0 && sqrt_y != 0) {
            y1 = sqrt_x;
            y2 = sqrt_y;
            if (IsOnCurve(x1, x2, y1, y2)) {
                foundValidPoint = true;
            } else {
                x1 += 1;
            }
        } else {
            // Increment x coordinate and try again.
            x1 += 1;
        }
    }

    return (G2Point([x2, x1], [y2, y1]));
}
```

Here, if the point $(h, \sqrt{h^3+3})$ does not exist on the curve, the value of h is incremented by 1 until such point exists. It means that the values $h_1 = h$ and $h_2 = h+1$ will map to the same point on the curve if the function `mapToG2` exits after one iteration for h_1.

Here is a proof of concept demonstrating a collision between $h_1 = 3$ and $h_2 = 4$:

```python
from bn128_curve import b2
from bn128_field import FQ2, FQ

def mapToG2(h):
    has_root = False

    while not has_root:
        x = FQ2([h, 0])
        y = x**3 + b2
        t, has_root = y.sqrt()
        if has_root:
            return (x,t)
        h +=1

if __name__ == "__main__":
    print(mapToG2(3) == mapToG2(4))
```

This contradicts the initial paper requirements of such function since this is a construction of a collision for the function above.

There is another flaw in the above construction. The imaginary part of the x-coordinate of the resulting point is constant. It means that the function `mapToG2` maps to a very small number of points, which is not uniform at all.

This problem is also present in the library mcl ↗ in the function `tryAndIncMapTo`.

Impact

In the library, the function is always applied on an input resulting from the `keccak256` function. Thus, the practical exploitability of such behavior is not obvious. However, the BLS-signature security proof does not apply anymore with such construction.

Recommendations

The function `mapToG2` should not deviate from the paper construction or be the implementation of a more recent construction as defined in the RFC 9380 ↗.

The previous behaviors should be implemented as unitary tests to avoid regression in the future.

Remediation

This issue has been acknowledged by Session team, and fixes were implemented in the following commits:

- `d84e9744` ↗
- `59aeae32` ↗

3.4. Bias in `hashToField` function

Target	BN256G2		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `hashToField` is used to hash a `uint256` value to $\mathbb{F}_p$. The resulting value is later mapped to a point on an elliptic curve with the function `mapToG2`. The message is first hashed with the `keccak256` function and then transformed to a value in the finite field with the function `maskBits`:

```
// This matches mcl maskN, this only takes the 254 bits for the field, if it is
    still greater than the field then take the 253 bits
function maskBits(uint256 input) internal pure returns (uint256) {
    uint256 mask = ~uint256(0) - 0xC0;
    if (byteSwap(input & mask) >= FIELD_MODULUS) {
        mask = ~uint256(0) - 0xE0;
    }
    return input & mask;
}
```

The two first bits of the value are set to zero, and then if the value is still bigger than p, the next bit is masked to zero. It means that the values between p and $2^{254} - 1$ are mapped to a value between $p - 2^{253}$ and $2^{253} - 1$. Values between zero and p are left unchanged, resulting in values in the range $[p - 2^{253}, 2^{253} - 1]$ having twice the probability to be chosen as output.

Impact

In the original BLS paper, the author assumed a function seen as a random oracle, which is not the case with such a construction. It seems difficult to exploit such bias in practice; however, the BLS-signature security proof does not apply anymore with such construction.

Recommendations

The construction should output uniformly distributed values with negligible bias similarly to the `hash_to_field` function defined in the RFC 9380 ↗.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit 58d5c68b ↗.

3.5. Wrong input length to static call

Target	BN256G1		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `add` function of BN256G1 is calling the precompiled contract performing an elliptic curve point addition. It uses a `staticcall` opcode directly in assembly. The opcode expects an input length as a parameter. In this function, it is set to `0xc0` but the input array is only 4 * 32 bytes, so it should be `0x80` instead.

The `mul` function has a similar mismatch with the input length set to `0x80`, whereas there are only three elements in the input array.

Impact

According to EIP-196 ↗, the input is shorter than expected; the input is padded with zero, which in this case would not change the result.

Recommendations

To avoid unexpected behaviors, the correct length should be set correctly. Some unitary tests with test vectors coming from a source other than the library would improve the confidence in the implementation.

Remediation

This issue has been acknowledged by Session team, and a fix was implemented in commit `9f3d379b` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Unused and sometimes misleading functions

Some functions are unused by the contracts. Here is a list found by the Slither ↗ tool:

```
BN256G1.hashToG1
BN256G1.modPow
BN256G1.mul
BN256G2.ECTwistAdd
BN256G2.ECTwistMul
BN256G2.GetFieldModulus
BN256G2.P2
BN256G2._FQ2Div
BN256G2._FQ2Pow
BN256G2.calcField
BN256G2.getWeierstrass
```

Some of them have misleading names. For example, `hashToG1` may be interpreted as computing a hash from an integer to the curve. However, this function simply performs a scalar multiplication, which has trivial collisions.

To avoid future misusage, unused functions should be removed.

Session team implemented the suggested changes in `4a59fcdc` ↗.

4.2. Gas savings

In the `FQ2Sqrt` function of BN256G2, a square root computation in the base field $\mathbb{F}_p$ can be saved. If the number has no imaginary part (i.e., `x2 == 0`) and the number is not a square, then the second square root computation `_sqrt(FIELD_MODULUS - x1)` is not necessary as it will return the same value `t1` as for the previous square root computation.

In $\mathbb{F}_p$, the square root a of x is computed with

$$a = x^{\frac{p+1}{4}} \mod p$$

and we have this:

$$a^2 = x^{\frac{p+1}{2}} = x^{\frac{p-1}{2}} x \mod p$$

If a is not a square, it means from the Legendre symbol that

$$a^{\frac{p-1}{2}} = -1 \mod p$$

So in $\mathbb{F}_p^2$, we have this:

$$(i \cdot a)^2 = -x^{\frac{p-1}{2}} x$$

And as explained before, the square root of x in $\mathbb{F}_p^2$ is $i \cdot a$ if a is not a square in $\mathbb{F}_p$.

Thus, the code can be replaced by the following:

```
// if x.b is zero
if (x2 == 0) {
    // Fp::squareRoot(t1, x.a)
    (t1, has_root) = _sqrt(x1);

    // if sqrt exists
    if (has_root) {
        return (t1, 0); // y.a = t1, y.b = 0
    } else {
        return (0, t1); // y.a = 0, y.b = t1
    }
}
```

It saves the gas of computing a second square root in the base field.

Another possible gas-saving measure is in `ECTwistMul`. There is no need to compute the scalar multiplication in case of the point at infinity; the point at infinity can be returned directly. Similarly, if the scalar is zero, there is no need to compute the multiplication.

Session team implemented the suggested changes for the square root computation in `3cef85e7` ↗.

4.3. Misleading functions' name

Functions named with suffix `Jacobian` in BN254G2 library are in fact not working with Jacobian coordinates but with projective coordinates. The point-addition algorithm for $\mathbb{G}_2$ implemented by the function `_ECTwistAddJacobian` is described by **formula 3** of this paper ↗, "Efficient Elliptic Curve Exponentiation Using Mixed Coordinates". The point-doubling algorithm implemented by function `_ECTwistDoubleJacobian` is the implementation of **formula 4** of the same paper. The point-doubling algorithm was later optimized in hyperelliptic.org ↗. The library may benefit from those improvements. In the same way, the function `_fromJacobian` transforms projective into affine coordinates.

The function `_ECTwistMulByCofactorJacobian` is an implementation of the algorithm from the paper "Faster Hashing to $\mathbb{G}_2$ ↗" and is also not working on Jacobian coordinates.

The naming is misleading, and the lack of reference to the original algorithms may be error-prone for later developments.

Adding tests with test vectors coming from a source other than the library would improve the confidence in the implementation.

Session team implemented the suggested changes in bd5657f2 ↗ with references to the implemented algorithms in the comments.

4.4. Data desynchronization

The `stakingRequirement` variable is set to the corresponding value from the ServiceNodeRewards contract in the constructor and cannot be changed in the future because the `stakingRequirement` is immutable. The `_stakingRequirement` variable from the ServiceNodeRewards contract corresponds to this value.

The `stakingRequirement` is used in the `contributeFunds` function to determine the sufficient amount of contributions collected — after which, funds will be transferred to the ServiceNodeRewards contract during the registration of the service node.

The `addBLSPublicKey` function from the ServiceNodeRewards contract expects that the exact `_stakingRequirement` amount of tokens will be transferred. But unlike the ServiceNodeContribution contract, the `_stakingRequirement` can be changed by the owner of the contract.

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

If `_stakingRequirement` from the ServiceNodeRewards contract is changed by the owner, the `stakingRequirement` value will become obsolete. As a result, for example, if `_stakingRequirement` was increased, the `addBLSPublicKey` function will be reverted due to the insufficient number of tokens sent. And because the `stakingRequirement` cannot be updated, this ServiceNodeContribution contract therefore cannot be used in the future.

This issue has been acknowledged by Session team. Their official response is paraphrased below:

Our solution to this is to immediately reject any further contributions if the staking requirement (or contributors) is detected to have changed. The reason for this is that:

1. The staking requirement and contributors is not expected to change very frequently and forcing operators to shutdown the contract via cancelling and reinitiating is an acceptable cost.
2. If we were to allow the requirements to change, this opens up a series of edge cases where contributors that previously had the correct minimum collaterialisation may suddenly become invalidly collateralised or over collateralised in which case all surrounding contracts have to handle this situation correctly. This significantly increases the permutation of possible cases to be handled.

In either case we will be rejecting any open contracts that have not yet been finalised during the time in which a staking or contributor limit has changed and informing the operator to recreate the contract with the new parameters. Note that changing these limits does not affect any contracts which have been finalised whereby the responsibility of managing their funds is done in the rewards contract where as-of-current is handled independently from the contribution contract and correctly.

4.5. Redundant checks

The `finalizeNode` function is exclusively called by the `contributeFunds` function. Both functions incorporate the checks shown below. Furthermore, the `finalizeNode` function is triggered solely when `totalContribution()` equals `stakingRequirement` but additionally verifies this condition. Therefore, duplicate checks can be deleted.

```
require(!finalized, "Node has already been finalized.");
require(!cancelled, "Node has been cancelled.");
```

This issue has been acknowledged by Session team, and a fix was implemented in commit 07460298 ↗.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: BN256G1.sol

Function: `add()`

This performs point addition in $\mathbb{G}_1$. It is a wrapper to the precompile contract, computing the addition over `alt_bn128` curves as defined in EIP-196 ↗.

Inputs

- `p1`
  - **Validation**: Points are validated to lie on the curve during the call of the precompiled contract and revert if not.
  - **Impact**: Avoid a resulting point not lying on the curve or having an erroneous resulting point.
- `p2`
  - **Validation**: Points are validated to lie on the curve during the call of the precompiled contract and revert if not.
  - **Impact**: Avoid a resulting point not lying on the curve or having an erroneous resulting point.

Branches and code coverage (including function calls)

Intended branches

- [x] The precompiled contract returns the addition result.

Negative behavior

- [ ] The precompiled contract reverts because a point is not on the curve.

Function call analysis

- `staticcall(sub(gas(), 2000), 7, input, 0x80, r, 0x60)`
  - **External/Internal?** External.
  - **Argument control**: `input` and sizes are controlled.
  - **Impact**: Addition computation of two points.

5.2. Module: BN256G2.sol

Function: `hashToG2()`

The function maps a `uint256` value to a point in the group $\mathbb{G}_2$. To fulfill BLS security, this function should be collision-resistant and not reveal the discrete logarithm of the resulting point.

Inputs

- `h`
    - **Validation**: No validation.
    - **Impact**: Map a value to $\mathbb{G}_2$.

Branches and code coverage (including function calls)

Intended branches

- ☑ The hash is mapped to a point.

Function call analysis

- `mapToG2(h)`
    - **External/Internal?** Internal.
    - **Argument control**: The arguments are controllable by the caller, but the value is usually a hash.
    - **Impact**: Maps a value to a point on the curve.
- `ECTwistMulByCofactor(map.X[1], map.X[0], map.Y[1], map.Y[0])`
    - **External/Internal?** Internal.
    - **Argument control**: No.
    - **Impact**: Clear the cofactor to have a point in $\mathbb{G}_2$.

5.3. Module: Pairing.sol

Function: `pairing()`

The `pairing` function is a wrapper to the precompile contract, computing the ate pairing operation over `alt_bn128` curves as defined in EIP-197 ↗. The pairing operation is used to verify the BLS signatures.

Inputs

- `p1`

    - **Validation**: If the points coordinates are bigger than or equal to p, the pairing operation fails and the contract reverts.
    - **Impact**: Points in $\mathbb{G}_1$ representing the generator and the aggregate public key.
- `p2`
    - **Validation**: If the points coordinates are bigger than or equal to p, the pairing operation fails and the contract reverts.
    - **Impact**: Points in $\mathbb{G}_2$ representing the signature and the message hash.

Branches and code coverage (including function calls)

Intended branches

- [x] The precompiled contract returns the pairing verification.

Negative behavior

- [ ] The precompiled contract reverts because of wrong encoding or a point on the curve but not in $\mathbb{G}_2$.

Function call analysis

- `call(sub(gas(), 2000), 8, 0, add(input, 0x20), mul(inputSize, 0x20), out, 0x20)`
    - **External/Internal?** External.
    - **Argument control**: `input` and sizes are controlled.
    - **Impact**: Paring check for the signature verification.

5.4. Module: RewardRatePool.sol

Function: `payoutReleased()`

This function calculates and releases the due interest payout to the beneficiary, updates the total paid out and the last payout time, and transfers the released funds to the beneficiary.

Branches and code coverage (including function calls)

Intended branches

- [ ] The reward amount is received, excluding the previously received rewards.

Function call analysis

- `payoutReleased() -> calculateReleasedAmount(block.timestamp) -> calculateInterestAmount(SENT.balanceOf(address(this)), timeElapsed)`
  - **External/Internal?** Internal.
  - **Argument control**: N/A.
  - **Impact**: Accurately calculates the amount of tokens given to the beneficiary over time.
- `SENT.safeTransfer(beneficiary, released)`
  - **External/Internal?** External.
  - **Argument control**: N/A.
  - **Impact**: Transfer `released` SENT tokens from the contract to the beneficiary.

5.5. Module: ServiceNodeContributionFactory.sol

Function: `deployContributionContract()`

The function deploys a new ServiceNodeContribution contract with a specified BLS public key and service node parameters, and it emits an event for the new contract deployment.

Inputs

- `blsPubkey`
  - **Validation**: N/A.
  - **Impact**: This variable is used to create the ServiceNodeContribution contract.
- `serviceNodeParams`
  - **Validation**: N/A.
  - **Impact**: This variable is used to create the ServiceNodeContribution contract.

Branches and code coverage (including function calls)

Intended branches

- ☐ Creates a ServiceNodeContribution contract using the given parameters.

Function call analysis

- `new ServiceNodeContribution(address(stakingRewardsContract), maxContributors, blsPubkey, serviceNodeParams)`
  - **External/Internal?** External.
  - **Argument control**: `blsPubkey` and `serviceNodeParams`.
  - **Impact**: Creates the ServiceNodeContribution contract.

5.6. Module: ServiceNodeContribution.sol

Function: `contributeFunds()`

This function allows any caller to contribute SENT tokens. The caller will be added to the `contributorAddresses` list. When the necessary amount of contribution is reached, the `stakingRewardsContract.addBLSPublicKey()` function will be called for service-node registration.

Inputs

- `amount`
    - **Validation**: The caller should have enough tokens to send.
    - **Impact**: The amount of SENT tokens to send to this contract.

Branches and code coverage (including function calls)

Intended branches

- ☐ `contributions` for the caller was updated.
- ☐ The caller was added to the `contributorAddresses`.

Function call analysis

- `SENT.safeTransferFrom(msg.sender, address(this), amount);`
    - **External/Internal?** External.
    - **Argument control**: `amount`.
    - **Impact**: Transfer SENT `amount` of tokens from the caller to the contract.
- `finalizeNode() -> SENT.approve(address(stakingRewardsContract), stakingRequirement);`
    - **External/Internal?** External.
    - **Argument control**: `stakingRequirement`.
    - **Impact**: Give the contract `stakingRewardsContract` the approval to spend `stakingRequirement` amount of SENT tokens.
- `finalizeNode() -> stakingRewardsContract.addBLSPublicKey(blsPubkey, blsSignature, serviceNodeParams, contributors)`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: The `stakingRequirement` of SENT tokens will be transferred to the `stakingRewardsContract` contract.

Function: `withdrawContribution()`

The function allows any contributor to withdraw their contribution.

Branches and code coverage (including function calls)

Intended branches

- ☐ The caller was deleted from the `contributorAddresses` list.
- ☐ The `contributions` for the caller address was set to zero.

Negative behavior

- ☐ Reverts if `finalized` is `true`.
- ☐ Reverts if caller is the `operator`.
- ☐ Caller does not have contribution.
- ☐ `cancelled` is true, but the delay in withdrawing funds has not passed yet.

Function call analysis

- `removeAndRefundContributor(msg.sender) -> SENT.safeTransfer(toRemove, result);`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: Transfer SENT token to the caller.

5.7. Module: ServiceNodeRewards.sol

Function: `addBLSPublicKey()`

The function adds a BLS public key to the list of service nodes, requiring a proof-of-possession signature, and optionally includes multiple contributors.

Inputs

- `blsPubKey`
    - **Validation**: Must match with `blsSignature` and must not be an already used `blsPubKey`.
    - **Impact**: A 64-byte BLS public key to identify the service node.
- `blsSignature`
    - **Validation**: Must match with `blsPubKey`.
    - **Impact**: A 128-byte BLS proof-of-possession signature that proves ownership of the `blsPubkey`.

- `serviceNodeParams`
    - **Validation**: N/A.
    - **Impact**: Includes the x25519 public key and signature to be added to the service node, as well as the fee charged by the operator.
- `contributors`
    - **Validation**: The number of contributors must be less than `_maxContributors`, and the first element of contributors must be the operator of the service node.
    - **Impact**: Represents a list of contributors for service nodes with multiple contributors.

Branches and code coverage (including function calls)

Intended branches

- ☐ Collects the designated token from the caller in the amount of `_stakingRequirement`.
- ☑ Updates the operator, contributors, and deposit of `_serviceNodes`.

Negative behavior

- ☐ Reverts if the proof-of-possession is invalid.
- ☐ Reverts if the number of contributors exceeds `maxContributors`.
- ☐ Reverts if the total tokens the contributors want to stake do not match `_stakingRequirement`.
- ☐ Reverts if there are existing `serviceNodeIDs` corresponding to the given `blsPubKey` parameter.

Function call analysis

- `validateProofOfPossession(blsPubKey, blsSignature, caller, serviceNodeParams.serviceNodePubkey)`
    - **External/Internal?** Internal.
    - **Argument control**: `blsPubKey`, `blsSignature`, `caller`, and `serviceNodeParams.serviceNodePubkey`.
    - **Impact**: Validate the BLS signature of the proof of possession to ensure the private key is known.
- `updateBLSNonSignerThreshold()`
    - **External/Internal?** Internal.
    - **Argument control**: N/A.
    - **Impact**: The function updates the internal threshold for the maximum number of non-signers allowed in an aggregate BLS signature before it can be validated.
- `SafeERC20.safeTransferFrom(designatedToken, caller, address(this), _stakingRequirement)`
    - **External/Internal?** External.

- **Argument control**: `caller`.
- **Impact**: Collects the `designatedToken` from the caller in the amount of the predefined `_stakingRequirement`.

Function: `initiateRemoveBLSPublicKey()`

The function initiates a request for a service node to leave the network by its service node ID, notifying the network of the node's intention to exit.

Inputs

- `serviceNodeID`
  - **Validation**: The `leaveRequestTimestamp` of `_serviceNodes` corresponding to the `serviceNodeID` must be zero. This means `initiateRemoveBLSPublicKey` cannot be called more than once for the same `serviceNodeID`.
  - **Impact**: Retrieve the `_serviceNodes` corresponding to the service ID.

Branches and code coverage (including function calls)

Intended branches

- [ ] The `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID` is updated to `block.timestamp`.

Negative behavior

- [ ] Reverts if `msg.sender` is not among the contributors of `_serviceNodes` matching the given `serviceNodeID`.
- [ ] Reverts if the `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID` is not zero.

Function call analysis

N/A.

Function: `liquidateBLSPublicKeyWithSignature()`

The function removes a service node by liquidating its node from the network, rewarding the caller and transferring the remaining funds according to predefined ratios, after validating the provided BLS signature and ensuring the network's approval for the liquidation.

Inputs

- `blsPubkey`
  - **Validation**: The `blsPubKey` must be the BLS public key of a preregistered ServiceNode.
  - **Impact**: The BLS public key of the ServiceNode to be removed.
- `timestamp`
  - **Validation**: The timestamp must be within the seconds specified by `signatureExpiry` from the current `block.timestamp`.
  - **Impact**: A parameter necessary to check the validity period of the `blsSignature`.
- `blsSignature`
  - **Validation**: N/A.
  - **Impact**: A parameter necessary to verify the `encodedMessage` generated in `liquidateBLSPublicKeyWithSignature`.
- `ids`
  - **Validation**: The length of `ids` must not exceed 1/3 of the total number of signers, and under no circumstances can the length of `ids` exceed 300, which can be adjusted by `setBLSNonSignerThresholdMax`.
  - **Impact**: The IDs of the signers to be excluded from the BLS public-key aggregation.

Branches and code coverage (including function calls)

Intended branches

- [x] The `_serviceNode` and `serviceNodeIDs` data matching the specified `serviceNodeID` are deleted.
- [ ] A certain amount of `designatedToken` is awarded to the address that executed this function.
- [ ] A certain amount of `designatedToken` is sent to the `foundationPool`.

Negative behavior

- [ ] Reverts if the `blsPubKey` given as a parameter is not a preregistered `blsPubKey` in `_serviceNodes`.
- [x] Reverts if BLS signature verification fails.

Function call analysis

- `BN256G2.hashToG2(BN256G2.hashToField(string(encodedMessage)))`
  - **External/Internal?** External.
  - **Argument control**: Value is hashed with `hashToField` before passing as an argument.
  - **Impact**: Hash computation before BLS signature verification.

- `validateSignatureOrRevert(ids, blsSignature, Hm) -> Pairing.pairing2(BN256G1.P1(), signature, BN256G1.negate(pubkey), hashToVerify)`
    - **External/Internal?** Internal.
    - **Argument control**: `ids`, `blsSignature`, and `Hm`.
    - **Impact**: Aggregates the excluded signers (`ids`) and verifies the `encodedMessage` (`Hm`) with the `blsSignature`.
- `_removeBLSPublicKey(serviceNodeID, deposit - liquidatorAmount - poolAmount) -> serviceNodeDelete(serviceNodeID)`
    - **External/Internal?** Internal.
    - **Argument control**: `serviceNodeID`.
    - **Impact**: Deletes the specified data from `_serviceNodes` and `serviceNodeIDs` — also removes the specified public key from `_aggregatePubKey`.
- `_removeBLSPublicKey(serviceNodeID, deposit - liquidatorAmount - poolAmount) -> updateBLSNonSignerThreshold()`
    - **External/Internal?** Internal.
    - **Argument control**: `serviceNodeID`.
    - **Impact**: Updates the number of signers that can be excluded from BLS signature aggregation.
- `SafeERC20.safeTransfer(designatedToken, msg.sender, liquidatorAmount)`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: Transfers the `liquidatorAmount` of `designatedToken` to the address that executed the `liquidateBLSPublicKeyWithSignature` function.
- `SafeERC20.safeTransfer(designatedToken, address(foundationPool), poolAmount)`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: Transfers the `poolAmount` of `designatedToken` to the `foundationPool`.

Function: `removeBLSPublicKeyAfterWaitTime()`

The function removes a BLS public key after the required wait time for a leave request has passed, without needing a signature.

Inputs

- `serviceNodeID`
    - **Validation**: The `leaveRequestTimestamp` of `_serviceNodes` corresponding to `serviceNodeID` must not be zero, and the current `block.timestamp` must be at least 30 days past the `leaveRequestTimestamp`.

- **Impact**: The `serviceNodeID` is necessary to delete specific `_serviceNode` and `serviceNodeIDs` data.

Branches and code coverage (including function calls)

Intended branches

- ☐ If sufficient time has passed since the `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID`, the corresponding `_serviceNode` and `serviceNodeIDs` data will be deleted.

Negative behavior

- ☐ Reverts if the `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID` is zero, indicating that `initiateRemoveBLSPublicKey` was not called previously.
- ☐ Reverts if the current `block.timestamp` is not at least 30 days past the `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID`.

Function call analysis

- `_removeBLSPublicKey(serviceNodeID, _serviceNodes[serviceNodeID].deposit) → serviceNodeDelete(serviceNodeID)`
  - **External/Internal?** Internal.
  - **Argument control**: `serviceNodeID`.
  - **Impact**: Deletes the specified data from `_serviceNodes` and `serviceNodeIDs` — also removes the specified public key from `_aggregatePubKey`.
- `_removeBLSPublicKey(serviceNodeID, _serviceNodes[serviceNodeID].deposit) → updateBLSNonSignerThreshold()`
  - **External/Internal?** Internal.
  - **Argument control**: `serviceNodeID`.
  - **Impact**: Updates the number of signers that can be excluded from BLS signature aggregation.

Function: `removeBLSPublicKeyWithSignature()`

The function removes a BLS public key from the network using an aggregated BLS signature, allowing the service node to exit and release the staked amount.

The difference between this function and `removeBLSPublicKeyAfterWaitTime` is that it does not require setting the `leaveRequestTimestamp` in `_serviceNodes` beforehand through `initiateRemoveBLSPublicKey`.

Inputs

- `blsPubkey`
    - **Validation**: The `blsPubKey` must be the BLS public key of a preregistered ServiceNode.
    - **Impact**: The BLS public key of the ServiceNode to be removed.
- `timestamp`
    - **Validation**: The timestamp must be within the seconds specified by `signatureExpiry` from the current `block.timestamp`.
    - **Impact**: A parameter necessary to check the validity period of the `blsSignature`.
- `blsSignature`
    - **Validation**: N/A.
    - **Impact**: A parameter necessary to verify the `encodedMessage` generated in `removeBLSPublicKeyWithSignature`.
- `ids`
    - **Validation**: The length of `ids` must not exceed 1/3 of the total number of signers, and under no circumstances can the length of `ids` exceed 300, which can be adjusted by `setBLSNonSignerThresholdMax`.
    - **Impact**: The IDs of the signers to be excluded from the BLS public-key aggregation.

Branches and code coverage (including function calls)

Intended branches

- [x] The `_serviceNode` and `serviceNodeIDs` data matching the specified `serviceNodeID` are deleted.

Negative behavior

- [x] Reverts if the signature is invalid.
- [ ] Reverts if the `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID` is zero, indicating that `initiateRemoveBLSPublicKey` was not called previously.
- [x] Reverts if the current `block.timestamp` is not at least 30 days past the `leaveRequestTimestamp` of `_serviceNodes` matching the given `serviceNodeID`.

Function call analysis

- `BN256G2.hashToG2(BN256G2.hashToField(string(encodedMessage)))`
    - **External/Internal?** External.
    - **Argument control**: Value is hashed with `hashToField` before passing as an argument.
    - **Impact**: Hash computation before BLS signature verification.
- `validateSignatureOrRevert(ids, blsSignature, Hm) -> Pair-`

`ing.pairing2(BN256G1.P1(), signature, BN256G1.negate(pubkey), hashToVerify)`
    - **External/Internal?** Internal.
    - **Argument control**: `ids`, `blsSignature`, and `Hm`.
    - **Impact**: Aggregates the excluded signers (`ids`), updates the verification key, and verifies the `encodedMessage` (`Hm`) with the `blsSignature`.
- `_removeBLSPublicKey(serviceNodeID, _serviceNodes[serviceNodeID].deposit) -> serviceNodeDelete(serviceNodeID)`
    - **External/Internal?** Internal.
    - **Argument control**: `serviceNodeID`.
    - **Impact**: Deletes the specified data from `_serviceNodes` and `serviceNodeIDs` — also removes the specified public key from `_aggregatePubKey`.
- `_removeBLSPublicKey(serviceNodeID, _serviceNodes[serviceNodeID].deposit) -> updateBLSNonSignerThreshold()`
    - **External/Internal?** Internal.
    - **Argument control**: `serviceNodeID`.
    - **Impact**: Updates the number of signers that can be excluded from BLS signature aggregation.

Function: `updateRewardsBalance()`

The function updates the rewards balance for a given recipient based on a verified BLS signature from the network.

Inputs

- `recipientAddress`
    - **Validation**: `recipientAddress` must not be zero.
    - **Impact**: The address of the wallet that will receive the rewards.
- `recipientRewards`
    - **Validation**: `recipientRewards` must be greater than the rewards that the `recipientAddress` has already received.
    - **Impact**: Token rewards of amount `recipientRewards` are given to `recipientAddress`.
- `blsSignature`
    - **Validation** N/A.
    - **Impact** A parameter necessary to verify the `encodedMessage` generated in `updateRewardsBalance`.
- `ids`
    - **Validation** The length of `ids` must not exceed 1/3 of the total number of signers, and under no circumstances can the length of `ids` exceed 300, which can be adjusted by `setBLSNonSignerThresholdMax`.
    - **Impact** The IDs of the signers to be excluded from the BLS public-key aggre-

gation.

Branches and code coverage (including function calls)

Intended branches

- ☑ Sets the `recipientAddress` to be able to claim rewards of amount `recipientRewards`.

Negative behavior

- ☐ Reverts if `recipientAddress` is `address(0)`.
- ☐ Reverts if `recipientRewards` is less than the rewards the `recipientAddress` has already received.
- ☑ Reverts if BLS signature verification fails.

Function call analysis

- `validateSignatureOrRevert(ids, blsSignature, Hm) -> Pairing.pairing2(BN256G1.P1(), signature, BN256G1.negate(pubkey), hashToVerify)`
  - **External/Internal?** Internal.
  - **Argument control**: `ids`, `blsSignature`, and `Hm`.
  - **Impact**: Aggregates the excluded signers (`ids`), updates the verification key, and verifies the `encodedMessage` (`Hm`) with the `blsSignature`.

5.8. Module: TokenConverter.sol

Function: `convertTokens()`

The function converts a specified amount of tokenA to tokenB at a predefined conversion rate, transferring tokenA from the user to the contract and sending the equivalent amount of tokenB to the user.

Inputs

- `_amountA`
  - **Validation**: The caller should have enough tokens to convert.
  - **Impact**: The amount of tokenA to send to this contract.

Branches and code coverage (including function calls)

Intended branches

- ☐ Collects tokenA from the user.

- ☐ Converts tokenA to tokenB at the specified rate and sends it to the user.

Negative behavior

- ☐ Reverts if `_amountA` is zero.
- ☐ Reverts if the user does not have enough tokenA for `_amountA`.
- ☐ Reverts if the contract does not have enough tokenB for the amount converted at the specified rate.

Function call analysis

- `tokenA.safeTransferFrom(msg.sender, address(this), _amountA)`
  - **External/Internal?** External.
  - **Argument control**: `_amountA`.
  - **Impact**: Collects `_amountA` of tokenA needed for the token conversion.
- `tokenB.safeTransfer(msg.sender, amountB);`
  - **External/Internal?** External.
  - **Argument control**: `amountB`.
  - **Impact**: Sends `amountB` of tokenB, which is the equivalent amount after conversion, to the user.

5.9. Module: TokenVestingStaking.sol

Function: `addBLSPublicKey()`

The function allows multiple `blsPubKey` to be added to the `stakingRewardsContract` and managed within the `investorServiceNodes` array.

Inputs

- `blsPubkey`
  - **Validation**: N/A.
  - **Impact**: A parameter used to add service-node information to the `stakingRewardsContract` contract.
- `blsSignature`
  - **Validation**: N/A.
  - **Impact**: A parameter used to add service-node information to the `stakingRewardsContract` contract.
- `serviceNodeParams`
  - **Validation**: N/A.
  - **Impact**: A parameter used to add service-node information to the `stakingRewardsContract` contract.

Branches and code coverage (including function calls)

Intended branches

- ☐ Adds service-node–related information to `investorServiceNodes`.
- ☐ Activates the specified service node by calling the `addBLSPublicKey` function in the `stakingRewardsContract`.

Negative behavior

- ☐ Reverts if the caller is not the `beneficiary`.
- ☐ Reverts if the contract is `revoked`.
- ☐ Reverts if the current `block.timestamp` is before the `start`.

Function call analysis

- `investorServiceNodes.push(ServiceNode(serviceNodeID, stakingRequirement))`
  - **External/Internal?** Internal.
  - **Argument control**: N/A.
  - **Impact**: Adds the information of the service node to be added to `investorServiceNodes`.
- `SENT.approve(address(stakingRewardsContract), stakingRequirement);`
  - **External/Internal?** External.
  - **Argument control**: N/A.
  - **Impact**: Approves the amount of SENT tokens to be staked in the `stakingRewardsContract`.
- `stakingRewardsContract.addBLSPublicKey(blsPubkey, blsSignature, serviceNodeParams, contributors);`
  - **External/Internal?** External.
  - **Argument control**: `blsPubkey` and `serviceNodeParams`.
  - **Impact**: Adds BLS public key and other data to the `stakingRewardsContract`.

Function: `claimRewards()`

The function claims staking rewards for the beneficiary, removes any unstaked service nodes from the `investorServiceNodes`, and transfers the claimed rewards to the beneficiary.

Branches and code coverage (including function calls)

Intended branches

- ☐ Removes the information of unstaked nodes from `investorServiceNodes`.
- ☐ After receiving rewards from `stakingRewardsContract`, transfers them to the `beneficiary`.

Function call analysis

- `stakingRewardsContract.serviceNodes(investorServiceNodes[i - 1].serviceNodeID)`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: The function retrieves service-node information from the `stakingRewardsContract`. After this function call, it checks the retrieved service-node information, and if it is confirmed that the service node has already been unstaked, it removes it from `investorServiceNodes`.
- `stakingRewardsContract.claimRewards()`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: Claims rewards from the `stakingRewardsContract`.
- `SENT.safeTransfer(beneficiary, amount)`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: Transfers the claimed rewards to the `beneficiary`.

Function: `initiateRemoveBLSPublicKey()`

The function starts the process for removing a service node by calling the `initiateRemoveBLSPublicKey` function of the `stakingRewardsContract`.

Branches and code coverage (including function calls)

Intended branches

- ☐ Calls the `initiateRemoveBLSPublicKey` function on the `stakingRewardsContract` with the given `serviceNodeID`.

Negative behavior

- ☐ Reverts if the caller is not the `beneficiary`.
- ☐ Reverts if the contract is `revoked`.
- ☐ Reverts if the current `block.timestamp` is before the `start`.

Function call analysis

- `stakingRewardsContract.initiateRemoveBLSPublicKey(serviceNodeID)`
    - **External/Internal?** External.
    - **Argument control**: `serviceNodeID`.
    - **Impact**: Initiates the removal process for the specified service node in the `stakingRewardsContract`.

Function: `release()`

The function transfers vested tokens to the beneficiary if the vesting period has been completed.

Branches and code coverage (including function calls)

Intended branches

- ☐ Calculates the releasable amount of tokens using `_releasableAmount`.
- ☐ Transfers the releasable tokens to the beneficiary.

Negative behavior

- ☐ Reverts if the caller is not the `beneficiary`.
- ☐ Reverts if the contract is `revoked`.
- ☐ Reverts if there are no tokens to be released.

Function call analysis

- `_releasableAmount(token)`
  - **External/Internal?** Internal.
  - **Argument control**: `token`.
  - **Impact**: Calculates the amount of tokens that can be released to the beneficiary.
- `token.safeTransfer(beneficiary, unreleased)`
  - **External/Internal?** External.
  - **Argument control**: N/A.
  - **Impact**: Transfers the releasable amount of tokens to the beneficiary.

Function: `retrieveRevokedFunds()`

The function allows the `revoker` to retrieve tokens that have been unstaked after the vesting has been revoked, ensuring that funds are not locked in the contract.

Branches and code coverage (including function calls)

Intended branches

- ☐ Checks if the vesting has been revoked.
- ☐ Transfers the balance of tokens to the `revoker`.

Negative behavior

- ☐ Reverts if the caller is not the `revoker`.

- ☐ Reverts if the contract has not been revoked.

Function call analysis

- `token.balanceOf(address(this))`
    - **External/Internal?** External.
    - **Argument control**: N/A.
    - **Impact**: Retrieves the balance of tokens held by the contract.
- `token.safeTransfer(revoker, balance)`
    - **External/Internal?** External.
    - **Argument control**: `balance`.
    - **Impact**: Transfers the balance of tokens to the `revoker`.

Function: `revoke()`

The function allows the revoker to revoke the vesting and stop the beneficiary from releasing any tokens if the vesting period has not been completed.

Branches and code coverage (including function calls)

Intended branches

- ☐ Checks if the vesting period has not expired.
- ☐ Calculates the balance and releasable amount of tokens.
- ☐ Calculates the refundable amount and sets the contract to revoked.
- ☐ Transfers the refundable amount to the revoker.

Negative behavior

- ☐ Reverts if the caller is not the `revoker`.
- ☐ Reverts if the contract is `revoked`.
- ☐ Reverts if the current `block.timestamp` is after the end of the vesting period.

Function call analysis

- `_releasableAmount(token)`
    - **External/Internal?** Internal.
    - **Argument control**: `token`.
    - **Impact**: Calculates the amount of tokens that can be released to the beneficiary.
- `token.safeTransfer(revoker, refund)`
    - **External/Internal?** External.

- **Argument control**: `refund`.
- **Impact**: Transfers the refundable amount of tokens to the `revoker`.

6. 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 five findings. No critical issues were found. Two findings were of high impact, one was of medium impact, one was of low impact, and the remaining finding was informational in nature.

6.1. 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.