Zellic

Biconomy PasskeyRegistry and SessionKeyManager

Smart Contract Security Assessment

August 14, 2023

Prepared for:

Sachin Tomar, Chirag, Aman

Biconomy Labs

Prepared by:

Katerina Belotskaia and Ulrich Myhre

Zellic Inc.

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 infosec 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 Executive Summary

Zellic conducted a security assessment for Biconomy Labs from August 8th to August 14th, 2023. During this engagement, Zellic reviewed Biconomy PasskeyRegistry and SessionKeyManager's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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 there a possibility that an invalid signature could be incorrectly accepted as valid?
- Could a malicious user sucessfully execute the `validateUserOp` function with an unauthorized signature?

1.2 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.

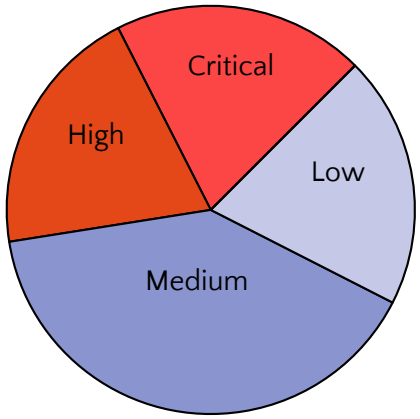
1.3 Results

During our assessment on the scoped Biconomy PasskeyRegistry and SessionKeyManager contracts, we discovered five findings. One critical issue was found. One was of high impact, two were of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Biconomy Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	1
Medium	2
Low	1
Informational	0



2 Introduction

2.1 About Biconomy PasskeyRegistry and SessionKeyManager

Biconomy PasskeyRegistry and SessionKeyManager are modules for Biconomy Smart Account.

The PasskeyRegistry module is an authorization module that enables users to deploy a smart contract wallet without an externally owned account (EOA) while relying on passkeys instead. Users can effortlessly generate wallets leveraging their biometric data, thereby eliminating the need to recall intricate private keys or passphrases.

Session keys are a powerful concept of temporary user-issued cryptographic keys that are authorized to sign only a predefined set of operations. Biconomy introduces a modular approach to session keys to unlock as much use cases as possible in an efficient and reliable way.

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 code base 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, upgradeability 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:

Biconomy PasskeyRegistry and SessionKeyManager Contracts

Repository	https://github.com/bcnmy/scw-contracts/tree/SCW-V2-Modular-SA
Version	scw-contracts: `f49764e4ff1e8e025bc2eb116a91be7a58a6c4d7`

Programs	Secp256r1 PasskeyRegistryModule SessionKeyManagerModule ERC20SessionValidationModule
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of one and a half person-weeks. The assessment was conducted over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 8, 2023	Start of primary review period
August 14, 2023	End of primary review period

3 Detailed Findings

3.1 ECDSA signatures can be trivially bypassed

- **Target**: Secp256r1.sol
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The final verification step in the PasskeyRegistryModule plug-in is to call the `Verify()` function in Secp256r1.sol. The latter does not adequately check the parameters before validation. The `passKey` parameter is picked directly from the internal mapping of public keys for `msg.sender` and is not directly controllable for someone else. Being of type `uint`, both `r` and `s` are guaranteed to be positive and the function also verifies that they are less than the order of the curve (the variable `nn` below). However, it is crucial to also verify that both $r \neq 0$ and $s \neq 0$ to avoid trivial signature bypasses.

```
function Verify(
    PassKeyId memory passKey,
    uint r,
    uint s,
    uint e
) internal view returns (bool) {
    if (r ≥ nn || s ≥ nn) {
        return false;
    }

    JPoint[16] memory points = _preComputeJacobianPoints(passKey);
    return VerifyWithPrecompute(points, r, s, e);
}
```

When the ECDSA verifies that a signature is signed by some public key, it takes in the tuple (**r**,**s**) (the signature pair) together with a public key and a hash. The hash is generated by hashing some representation of the operation that should be executed, and proving the signature for that hash means the owner of the public key approved the operation. The main calculation for verification in ECDSA is

R' = (**h** * **s_inv**) * **G** + (**r** * **s_inv**) * **pubKey**

where **s_inv** is the inverse of scalar **s** on the curve (i.e. inverse of **s** modulo the curve order) and **h** is the hash. The signature is said to be verified if the x-coordinate of the resulting point is equal to **r**, as in

(**R'**).x == **r**

Replacing **r** and **s** with 0, we get that **s_inv** is also 0, and the calculation becomes

R' = (**h** * `0`) * **G** + (`0` * `0`) * **pubKey**

R' = `0` * **G** + `0` * **pubKey**

R' = `0` == **r**

and the signature verification is **always** successful.

Impact

Anyone who can submit operations that will be validated by the PasskeyRegistryModule can impersonate other users and do anything that the account owner could do, leading to loss of funds.

Recommendations

Check that none of (**r**,**s**) are equal to zero.

```
function Verify(
    PassKeyId memory passKey,
    uint r,
    uint s,
    uint e
) internal view returns (bool) {
    if (r ≥ nn || s ≥ nn || r==0 || s==0) {
        return false;
    }

    JPoint[16] memory points = _preComputeJacobianPoints(passKey);
    return VerifyWithPrecompute(points, r, s, e);
}
```

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit 5c5a6bfe.

3.2 PasskeyRegistryModule reverts when validating user operations

- **Target**: PasskeyRegistryModule.sol
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: High
- **Impact**: High

Description

The PasskeyRegistryModule contract is called from SmartAccount.sol through the `validateUserOp` function,

```
function validateUserOp(
    UserOperation calldata userOp,
    bytes32 userOpHash,
    uint256 missingAccountFunds
) external virtual override returns (uint256 validationData) {
    if (msg.sender != address(entryPoint()))
        revert CallerIsNotAnEntryPoint(msg.sender);

    (, address validationModule) = abi.decode(
        userOp.signature,
        (bytes, address)
    );
    if (address(modules[validationModule]) != address(0)) {
        validationData = IAuthorizationModule(validationModule)
            .validateUserOp(userOp, userOpHash);
    } else {
        revert WrongValidationModule(validationModule);
    }
    _validateNonce(userOp.nonce);
    _payPrefund(missingAccountFunds);
}
```

where `userOp.signature` is decoded to figure out the address for the module that should do the actual validation. The `validateUserOp()` function takes in the raw, unprocessed `userOp` struct (of type `UserOperation`).

Inside PasskeyRegistryModule.sol, the `validateUserOp(userOp, userOpHash)` function is just a wrapper for `_validateSignature(userOp, userOpHash)`, which is a wrapper for `_verifySignature(userOpHash, userOp.signature)`. Do note that the `userOp.sign`

ature element was passed to the last function. This is the exact same value that was decoded in SmartAccount→validateUserOp(), and it contains both the signature data and the validation module address.

The final function, _verifySignature(), starts like this,

```
function _verifySignature(
    bytes32 userOpDataHash,
    bytes memory moduleSignature
) internal view returns (bool) {
    (
        bytes32 keyHash,
        uint256 sigx,
        uint256 sigy,
        bytes memory authenticatorData,
        string memory clientDataJSONPre,
        string memory clientDataJSONPost
    ) = abi.decode(
            moduleSignature,
            (bytes32, uint256, uint256, bytes, string, string)
        );
    ...
}
```

where it tries to decode the signature (including the address) as (bytes32, uint256, uint256, bytes, string, string). This will revert because the validation module address is still a part of the decoded blob.

In the SessionKeyManagerModule.sol contract, the validateUserOp() function is implemented correctly

```
function validateUserOp(
    UserOperation calldata userOp,
    bytes32 userOpHash
) external view virtual returns (uint256) {
    SessionStorage storage sessionKeyStorage
    = _getSessionData(msg.sender);
    (bytes memory moduleSignature, ) = abi.decode(
        userOp.signature,
        (bytes, address)
    );
    // Here it does `abi.decode(moduleSignature, ...)`
```

```
        ...
}
```

where the address is stripped off before decoding the remainder.

Impact

The module will always revert and is not usable. If it is the only available validation module, no user operations can happen.

Recommendations

Strip off the address like the SessionKeyManagerModule does, and write test cases for the module. Simple test cases can find mistakes such as these earlier. In general, it is good practice to build a rigorous test suite to ensure the system operates securely and as intended.

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit 5c5a6bfe.

3.3 Missing test coverage

- **Target**: Secp256r1.sol
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The Secp256r1 module implements critical functionality for signature validation, and it is implemented in a nonstandard and highly optimized way. To ensure that the library works in common cases, edge cases, and invalid cases, it is crucial to have proper test coverage for these types of primitives. There are currently no tests using this library, making it hard to see if it works at all.

Impact

Missing test cases could lead to critical bugs in the cryptographic primitives. These could lead to, for example,

- Signature forgery and total account takeover
- Surprising or very random gas costs
- Proper signatures not validating, leading to DOS
- Recovery of private keys in extreme cases.

Recommendations

Google has Project Wycheproof, which includes many test vectors for common cryptographic libraries and their operations. A good match for this module, which uses Secp256r1 (aka NIST P-256) and 256-bit hashes, is to use the `ecdsa_secp256r1_sha256_test.json` test vectors. Do note that many of these vectors target DER decoding, so it is safe to skip tests tagged "BER". Additionally, test cases where they use numbers larger than 256 bits can be ignored, as they are invalid in Solidity when using `uint256` types.

These test vectors can be somewhat easily converted to Solidity library tests, giving hundreds of tests for free.

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit 5c5a6bfe.

3.4 Modexp has arbitrary gas limit

- **Target**: Secp256r1.sol
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: Medium

Description

The Secp256r1 library makes use of the EIP-198 precompile in order to do modular exponentiation. This function is located at address `0x5` and is called near the end of the function.

```
function modexp(
    uint _base,
    uint _exp,
    uint _mod
) internal view returns (uint ret) {
    // bigModExp(_base, _exp, _mod);
    assembly {
        if gt(_base, _mod) {
            _base := mod(_base, _mod)
        }
        // Free memory pointer is always stored at 0x40
        let freemem := mload(0x40)

        mstore(freemem, 0x20)
        mstore(add(freemem, 0x20), 0x20)
        mstore(add(freemem, 0x40), 0x20)

        mstore(add(freemem, 0x60), _base)
        mstore(add(freemem, 0x80), _exp)
        mstore(add(freemem, 0xa0), _mod)

        let success := staticcall(1500, 0x5, freemem, 0xc0, freemem, 0x20)
        switch success
        case 0 {
            revert(0x0, 0x0)
        }
        default {
            ret := mload(freemem)
        }
```

```
    }
}
```

A gas limit of 1,500 is set for this operation. After EIP-2565, the precompile was updated to become more optimized and cost less gas. Before this optimization, the gas cost was sometimes very high and often overestimated. The EIP provides a function to calculate the approximate gas cost, and using the parameters from the library, we calculated it to be around 1,360 gas, which is barely within the limit. With EIP-198 pricing, the cost was significantly higher.

Some chains have not yet implemented this optimization — one example being the BNB chain, which plans to implement their equivalent BEP-225 around August 30th, 2023.

The standard for signature validation methods, EIP-1271, also states the following:

> Since there [is] no gas-limit expected for calling the `isValidSignature()` function, it is possible that some implementation will consume a large amount of gas. It is therefore important to not hardcode an amount of gas sent when calling this method on an external contract as it could prevent the validation of certain signatures.

Impact

On chains without the gas optimization change for the precompile, the contract will either not work or randomly work for certain keys and signatures but not others. In the worst-case scenario, someone could be extremely lucky and manage to transfer money in but not be able to get them out again. The main risk is just the functionality of the module being broken.

Recommendations

Provide more or the maximum amount of gas to this function call:

```
let success := staticcall(not(0), 0x5, freemem, 0xc0, freemem, 0x20)
```

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit 5c5a6bfe.

3.5 Secp256r1 curve test failures

- **Target**: Secp256r1.sol
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: Low
- **Impact**: Low

Description

During the audit, a number of test vectors were ported to verify the ECDSA signature verification module Secp256r1.sol. One of these test vectors was `test case ID 345: extreme value for k and s^-1` from Project Wycheproof. The parameters in this test picks values such that the inverse of **s** is very high and verifies that the signature still passes the check. This test case fails for this module.

To verify that the signature is indeed correct, we tested all the parameters in SageMath with the following script,

```
# Curve parameters
p256 = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
a256 = p256 - 3
b256 = 0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
gx = 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
gy = 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5
qq = 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
FF = GF(p256)
E = EllipticCurve([FF(a256), FF(b256)])
E.set_order(qq)
G = E(FF(gx), FF(gy))
# Test parameters
pubx = 0xc6a771527024227792170a6f8eee735bf32b7f98af669ead299802e32d7c3107
puby = 0xbc3b4b5e65ab887bbd343572b3e5619261fe3a073e2ffd78412f726867db589e
r = 0x7cf27b188d034f7e8a52380304b51ac3c08969e277f21b35a60b48fc47669978
s = 0xb6db6db624924925492492492492492492492492492492492492492492492492
h = 0xbb5a52f42f9c9261ed4361f59422a1e30036e7c32b270c8807a419feca605023
s1 = inverse_mod(s, E.order())
assert ((h * s1) * G + (r*s1)*E(pubx, puby)).xy()[0] == r
```

which passes with no issues. A similar test failure raised concerns in the fastecdsa project and got assigned CVE-2020-12607 at a very high severity level (though this CVE score is disputed).

Impact

The full impact is hard to classify without more knowledge about the root cause of the bug, but the most likely effect will be that some valid signatures will be rejected — but rarely. The mentioned CVE-2020-12607 claims that this is not merely a usability problem, so depending on the root cause, it could be dangerous.

Recommendations

The likely culprit is lack of consideration for the point at infinity when adding or doubling with modified Jacobians. The code comments link to the website Point-at-Infinity where the optimized algorithm used in the code is broken down. Some of the conditionals, where an early `return POINT_AT_INFINITY` is reached, are missing from the contract code. These checks should be implemented properly to check if the point at infinity occurs during calculations.

We also **strongly** recommend to implement comprehensive test cases that go beyond test vectors but also target edge cases in the calculations.

Remediation

This issue has been acknowledged by Biconomy Labs.

Following the completion of this audit, Zellic conducted a separate review of Secp256r1, the details of which can be accessed here.

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 Vulnerable MerkleProof library version

During the security audit, it has been identified that the SessionKeyManager is utilizing an outdated version of the MerkleProof library from OpenZeppelin. The specific version in use, ranging from V4.7 to V4.9, is vulnerable to the CVE-2023-34459 exploit, which may allow malicious actors to prove arbitrary leaves for specific trees when utilizing multiproofs. Although the SessionKeyManager is not directly utilizing the vulnerable functions of the MerkleProof, it is strongly recommended to upgrade to the latest version of the library as a best practice for maintaining the security and integrity of the project. By using the latest version of the library, the project ensures its protection against potential future exploits and vulnerabilities that may arise due to using outdated code.

4.2 The `validateSessionUserOp` function repeatedly decodes `_sessionKeyData`

The `validateSessionUserOp` function repeatedly decodes `_sessionKeyData` using hard-coded offsets, introducing a risk of error-prone code. Such redundancy in decoding not only complicates the code structure but also increases the probability of mistakes when making changes or updates.

```
    function validateSessionUserOp(
        UserOperation calldata _op,
        bytes32 _userOpHash,
        bytes calldata _sessionKeyData,
        bytes calldata _sessionKeySignature
    ) external view returns (bool) {
        address sessionKey = address(bytes20(_sessionKeyData[0:20]));
        // 20:40 is token address
        address recipient = address(bytes20(_sessionKeyData[40:60]));
        uint256 maxAmount = abi.decode(_sessionKeyData[60:92],
    (uint256));
```

```
        {
            address token = address(bytes20(_sessionKeyData[20:40]));

            // we expect _op.callData to be `SmartAccount.executeCall(to,
    value, calldata)` calldata
            (address tokenAddr, uint256 callValue, ) = abi.decode(
                _op.callData[4:], // skip selector
                (address, uint256, bytes)
            );
             ...
        }
...
```

We recommend using the `api.decode` function to avoid these problems.

```
(address sessionKey, address token, address recipient, uint256 maxAmount)
    = abi.decode(_sessionKeyData, (address, address, address, uint256));
```

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: ERC20SessionValidationModule.sol

Function: `validateSessionUserOp(UserOperation _op, bytes32 _userOpHash, bytes _sessionKeyData, bytes _sessionKeySignature)`

The purpose of this function is to ensure that the provided user operation is valid, matches the session key permissions, and is approved by the session key signature.

- `_op`: This is a `UserOperation` struct representing the user's operation details.
- `_userOpHash`: This is the hash of the user's operation.
- `_sessionKeyData`: This is encoded session-key data containing the session key, token address, recipient address, and maximum-allowed amount.
- `_sessionKeySignature`: This is the signature of the session key.

The ECDSA signature of the `_userOpHash` is recovered using the provided `_sessionKeySignature`. The recovered address is then compared with the session key. If they match, the function returns true, indicating that the user operation is valid.

5.2 Module: PasskeyRegistryModule.sol

Function: `initForSmartAccount(uint256 _pubKeyX, uint256 _pubKeyY, string _keyId)`

The function takes the public keys (`_pubKeyX` and `_pubKeyY`) generated by decoding the data from the passkey. The `msg.sender` has the capability to establish their public keys just once and is unable to make any modifications subsequently.

Inputs

- `_pubKeyX`

  - **Constraints**: No checks.
  - **Impact**: The x-coordinate of the public key, which will be used to verify the signature.
- `_pubKeyY`
  - **Constraints**: No checks.
  - **Impact**: The y-coordinate of the public key, which will be used to verify the signature.
- `_keyId`
  - **Constraints**: No checks.
  - **Impact**: The `keyId` of the smart account.

Branches and code coverage (including function calls)

Negative behavior

- `_pubKeyX` is zero.
  - ☐ Negative test
- `_pubKeyY` is zero.
  - ☐ Negative test
- Modify already initiated public keys.
  - ☐ Negative test

Function: `validateUserOp(UserOperation userOp, byte[32] userOpHash)`

The function is invoked to verify the signature each time a new transaction arrives for the smart account. In the case of a valid signature, the function yields a return value of 0. If the signature is invalid, it produces a return value of `SIG_VALIDATION_FAILED`.

Inputs

- `userOp`
  - **Constraints**: The signature will be validated by the `_verifySignature` function.
  - **Impact**: The signature field contains the signature data (`keyHash`, `sigx`, `sigy`, `authenticatorData`, `clientDataJSONPre`, `clientDataJSONPost`) that will be validated.
- `userOpHash`
  - **Constraints**: N/A.
  - **Impact**: The hash of the user operation to be validated.

Branches and code coverage (including function calls)

Negative behavior

- The `sigx` is zero.
  - ☐ Negative test
- The `sigy` is zero.
  - ☐ Negative test
- The `passKey` is not set.
  - ☐ Negative test

Function call analysis

- `_validateSignature(userOp, userOpHash)` → `_verifySignature(userOpHash, userOp.signature)` → `Secp256r1.Verify(passKey, sigx, sigy, uint256(sigHash))`;
  - **What is controllable?** Both `userOp` and `userOpHash` are controllable by the caller of this view function, but in the main smart account use case this data comes from the `EntryPoint.sol:handleOps()` function, which calculates the `userOpHash` hash using the user operation data provided by the caller.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is used by the `EntryPoint.sol:handleOps()` function to determine whether this operation is allowed to be executed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The function can be reverted in case `passKey` is not set for this smart account or in case of a calculation error.

5.3 Module: SessionKeyManagerModule.sol

Function: `setMerkleRoot(byte[32] _merkleRoot)`

Allow any caller to set the Merkle root. The Merkle root can be modified by the `msg.sender` — but exclusively for the address of the `msg.sender`. While the expectation is for the `msg.sender` to be the SmartAccount contract, in reality, it could be initiated by any caller.

The `_merkleRoot` is the Merkle root data containing the session keys with their permissions and parameters.

Function: `validateUserOp(UserOperation userOp, byte[32] userOpHash)`

The function is invoked to verify the signature each time a new transaction arrives for the smart account, triggering the associated module. This function ensures that the leaf, encompassing data such as `validUntil`, `validAfter`, `sessionValidationModule`, and `sessionKeyData`, is integrated within the Merkle root established by the `msg.sender` (the smart account). Should this condition hold true, the `validateSessionUserOp` function will be invoked for the respective `sessionValidationModule` that provides support for the `sessionKeyData`.

Inputs

- `userOp`
  - **Constraints**: This contains the data of user operation that will be validated.
  - **Impact**: The user operation data that will be validated.
- `userOpHash`
  - **Constraints**: This will be checked by the `validateSessionUserOp` function.
  - **Impact**: The hash of user operation data that will be validated.

Branches and code coverage (including function calls)

Negative behavior

- `userOp.signature` does not contain `moduleSignature`.
  - ☐ Negative test
- The untrusted `sessionValidationModule`.
  - ☐ Negative test
- The invalid user operation.
  - ☐ Negative test

Function call analysis

- `ISessionValidationModule(sessionValidationModule).validateSessionUserOp(userOp,userOpHash,sessionKeyData,sessionKeySignature)`,
  - **What is controllable?** `userOp`, `sessionKeyData`, and `sessionKeySignature`.
  - **If return value controllable, how is it used and how can it go wrong?** The returned value means that the checked user operation is valid and can be executed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This will revert if the user operation data does not match the `sessionKeyData`.

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 Biconomy PasskeyRegistry and SessionKeyManager contracts, we discovered five findings. One critical issue was found. One was of high impact, two were of medium impact, and one was of low impact. Biconomy Labs acknowledged all findings and implemented fixes.

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.