Biconomy Smart Account

Smart Contract Security Assessment

June 20, 2023

Prepared for:

Phil Makarov

Biconomy Labs

Prepared by:

Katerina Belotskaia and Ulrich Myhre

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Biconomy Labs from June 12th to June 16th, 2023. During this engagement, Zellic reviewed Biconomy Smart Account'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:

- Could an invalid signature pass verification?
- Could a malicious user enable a malicious module?

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 Smart Account contracts, we discovered one finding, which was informational in nature.

Breakdown of Finding Impacts

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

2 Introduction

2.1 About Biconomy Smart Account

Biconomy Smart Account is a modular ERC-4337–compatible smart account. In this implementation, it has been made ownerless by nature, so the txns/`userOps` are validated via validation modules.

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 the contracts' 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.

2.3 Scope

The engagement involved a review of the following targets:

Biconomy Smart Account Contracts

Repository	https://github.com/bcnmy/scw-contracts
Version	scw-contracts: `96bb665530eacb8f98e29b3ce404ff4596467e2e`
Programs	• BaseSmartAccount • SmartAccount • SmartAccountFactory • ModuleManager • EcdsaOwnershipRegistryModule • SmartContractOwnershipRegistryModule • ForwardFlowModule
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

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.

June 12, 2023	Kick-off call
June 12, 2023	Start of primary review period
June 16, 2023	End of primary review period

3 Detailed Findings

3.1 Improve test suite

- **Target**: Multiple (e.g., SmartContractOwnershipRegistryModule)
- **Category**: Code Maturity
- **Severity**: Low
- **Likelihood**: N/A
- **Impact**: Informational

Description

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, not just surface-level functions. It is important to test the invariants required for ensuring security and also verify mathematical properties as specified in the white paper. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Impact

For contracts like SmartContractOwnershipRegistryModule, there are not even basic test cases to verify that it works at all. This introduces some risk of the functionality being broken, and any finding stemming from a later fix might not be evaluated as a part of the audit.

Recommendations

Implement test cases for all the modules and aim to keep the coverage report possible to run, without too much friction (right now this does not support the `--grep` param and runs failing tests). As a minimum, check that all reverts can be hit and that basic functionality works. If the input space is big, consider writing fuzzing test cases for these.

Remediation

This issue has been acknowledged by Biconomy Labs.

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart 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.

4.1 Module: ModuleManager.sol

Function: `execTransactionFromModule(address to, uint256 value, byte[] data, Enum.Operation operation, uint256 txGas)`

Available only for enabled modules. Allows a trusted module to perform execute transaction directly.

Branches and code coverage (including function calls)

Negative behavior

- The caller is not a trusted module.
  - ☐ Negative test
- The caller is the disabled module.
  - ☐ Negative test

Function call analysis

- `execute(to, value, data,operation,txGas == 0 ? gasleft() : txGas);` → `delegatecall(txGas,to,add(data, 0x20),mload(data),0,0)`
  - **External/Internal?** External.
  - **Argument control?** `txGas`, `to`, and `data`.
  - **Impact**: Perform `delegatecall` of `to` address.
- `execute(to, value, data,operation,txGas == 0 ? gasleft() : txGas);` → `call(txGas,to,value,add(data, 0x20),mload(data),0,0)`
  - **External/Internal?** External.
  - **Argument control?** `txGas`, `to`, `data`, and `value`.
  - **Impact**: Perform call of `to` address.

Function: `execTransactionFromModule(address to, uint256 value, byte[] data, Enum.Operation operation)`

The same as `execTransactionFromModule(address to, uint256 value, bytes memory data, Enum.Operation operation, uint256 txGas)`, but additional `txGas` is zero.

4.2 Module: SmartAccountFactory.sol

Function: `deployCounterFactualAccount(address moduleSetupContract, byte[] moduleSetupData, uint256 index)`

Allows any users to deploy smart account contracts.

Inputs

- `moduleSetupContract`
  - **Control**: Full control.
  - **Constraints**: No restrictions.
  - **Impact**: The address of the module that will be enabled and set up during the `init()` function call.
- `moduleSetupData`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: Contain function signature and data for module call.
- `index`
  - **Control**: Full control.
  - **Constraints**: If the contract with the index was already deployed, the transaction will be reverted.
  - **Impact**: Extra salt.

Branches and code coverage (including function calls)

Intended branches

- New smart account contract was initialized properly.
  - ☐ Test coverage

Negative behavior

- Revert if `index` was already used by the same EOA.
  - ☐ Negative test

Function call analysis

- `proxy.init(address(minimalHandler), moduleSetupContract, moduleSetupData)`
  - **External/Internal?** External.
  - **Argument control?** `moduleSetupContract` and `moduleSetupData`.
  - **Impact**: Initialize new smart account contract with required state.
- `moduleSetupContract.call( ... ,moduleSetupData, ... )`
  - **External/Internal?** External.
  - **Argument control?** `moduleSetupContract` and `moduleSetupData`.
  - **Impact**: Call `moduleSetupContract` over the low-level `call` for the initialization of the module, for example, the installation of the owner address of this smart account.

4.3 Module: SmartAccount.sol

Function: `addDeposit()`

Available for anyone. Transfer native tokens provided by caller to the EntryPoint contracts.

Function: `disableModule(address prevModule, address module)`

Function available only for EntryPoint or self-call. Allows to disable the module address, and it cannot be used any more for validation.

Function: `enableModule(address module)`

Function available only for EntryPoint or self-call. Allows caller to enable the new address of the module.

Inputs

- `module`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The address of the module used to verify user operation.

Branches and code coverage (including function calls)

Intended branches

- New module was enabled.

  - ☐ Test coverage

Negative behavior

- The caller is not entry point or this contract.
  - ☐ Negative test

Function call analysis

- `_enableModule(module)`
  - **External/Internal?** Internal.
  - **Argument control?** `module`.
  - **Impact**: Add module address to the `modules` to enable.

Function: `executeCall(address dest, uint256 value, byte[] func)`

Function just calls `executeCall_s1m`, which is available only for EntryPoint and that is it. See `executeCall_s1m` description.

Function: `executeCall_s1m(address dest, uint256 value, byte[] func)`

Function available only for EntryPoint. Allows EntryPoint to perform the transaction.

Inputs

- `dest`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The arbitrary contract that will be called.
- `value`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: Amount of native tokens will be transferred.
- `func`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The transaction data. Contains the function that will be called.

Branches and code coverage (including function calls)

Negative behavior

- `msg.sender` is not EntryPoint or this contract.

☐ Negative test

Function call analysis

- `_call(dest, value, func) → call( … , target, value, add(data, 0x20), mload(data), … )`
  - **External/Internal?** External.
  - **Argument control?** `target`, `value`, and `data`.
  - **Impact**: Arbitrary external call.

Function: `init(address handler, address moduleSetupContract, byte[] moduleSetupData)`

The function is called only once during deployment from proxy contract. Allows to set up and enable the module provided by call of `deployCounterFactualAccount` or `deployAccount` of the SmartAccountFactory contract.

Inputs

- `handler`
  - **Control**: Full control.
  - **Constraints**: Cannot be zero address.
  - **Impact**: The address of the contract handling the fallback calls.
- `moduleSetupContract`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The address of module.
- `moduleSetupData`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The calldata for moduleSetupContract.

Branches and code coverage (including function calls)

Negative behavior

- Cannot be called twice.
  ☐ Negative test

Function call analysis

- `_initialSetupModules → call( … , moduleSetupContract, … , moduleSetupDa`

`ta)`
  - **External/Internal?** External.
  - **Argument control?** moduleSetupContract, moduleSetupData
  - **Impact**: Calling the arbitrary moduleSetupContract contract, which should be trusted by the caller.

Function: `setupAndEnableModule(address setupContract, byte[] setupData)`

Function available only for EntryPoint or self-call. Allows caller to enable the new address of the module and call it for configuration when this is the first enabling.

Inputs

- `setupContract`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The address of the module used to verify user operation.
- `setupData`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: Data used to configure the module.

Branches and code coverage (including function calls)

Intended branches

- New module was enabled.
  - ☐ Test coverage

Negative behavior

- The caller is not entry point or this contract.
  - ☐ Negative test

Function call analysis

- `_setupAndEnableModule(setupContract, setupData) → call( … , setupContract, … , add(setupData, 0x20), mload(setupData), … )`
  - **External/Internal?** External.
  - **Argument control?** `setupContract` and `setupData`.
  - **Impact**: Calls `setupContract` to configure before use.

Function: `updateImplementation(address _implementation)`

Function available only for EntryPoint or self-call. Allows to update the address of the implementation that is used by proxy contract.

Inputs

- `_implementation`
  - **Control**: Full control.
  - **Constraints**: `_implementation` ≠ `address(0)`.
  - **Impact**: The address of the contract that will be used as smart account implementation over `delegatecall` by proxy contract.

Branches and code coverage (including function calls)

Intended branches

- The implementation was updated properly.
  - ☐ Test coverage

Negative behavior

- `_implementation == address(0)`.
  - ☐ Negative test
- Caller is not entry point and self-call.
  - ☐ Negative test
- `_implementation` is EOA.
  - ☐ Negative test

Function: `validateUserOp(UserOperation userOp, byte[32] userOpHash, uint256 missingAccountFunds)`

Function available only for EntryPoint. The function is called from the `EntryPoint.handleOps` function, which can be called by any caller that should have valid user operation data. The function will revert if the module is untrusted — otherwise 0 if signature is valid or 1 if invalid.

Branches and code coverage (including function calls)

Negative behavior

- Caller is not EntryPoint contract.
  - ☐ Negative test
- `validationModule` is not the enabled module.

    ☐ Negative test
- User operation is invalid.
    ☐ Negative test

Function call analysis

- `IAuthorizationModule(validationModule).validateUserOp(userOp, userOpHash)`
    - **External/Internal?** External.
    - **Argument control?** `validationModule`, `userOp`, and `userOpHash`.
    - **Impact**: If signature is valid, 0, or 1 if invalid. Can revert if signature has incorrect length or `userOp.sender` is zero address.

Function: `withdrawDepositTo(address payable withdrawAddress, uint256 amount)`

Function available only for EntryPoint or self-call. Withdraw funds from the EntryPoint contract.

Inputs

- `withdrawAddress`
    - **Control**: Full control.
    - **Constraints**: No.
    - **Impact**: The received withdrawn funds.
- `amount`
    - **Control**: Full control.
    - **Constraints**: No.
    - **Impact**: Amount to be withdrawn.

Branches and code coverage (including function calls)

Intended branches

- `withdrawAddress` received the funds.
    ☐ Test coverage

Negative behavior

- The caller is not entry point or this contract.
    ☐ Negative test

Function call analysis

- `EntryPoint.withdrawTo(withdrawAddress, amount)`
  - **External/Internal?** External.
  - **Argument control?** `withdrawAddress` and `amount`.
  - **Impact**: Transfer deposited funds from EntryPoint to `withdrawAddress`.

5 Audit Results

At the time of our audit, the audited code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Biconomy Smart Account contracts, we discovered one finding, which was informational in nature. Biconomy Labs acknowledged the finding and implemented a fix.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of 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.

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.