Zellic

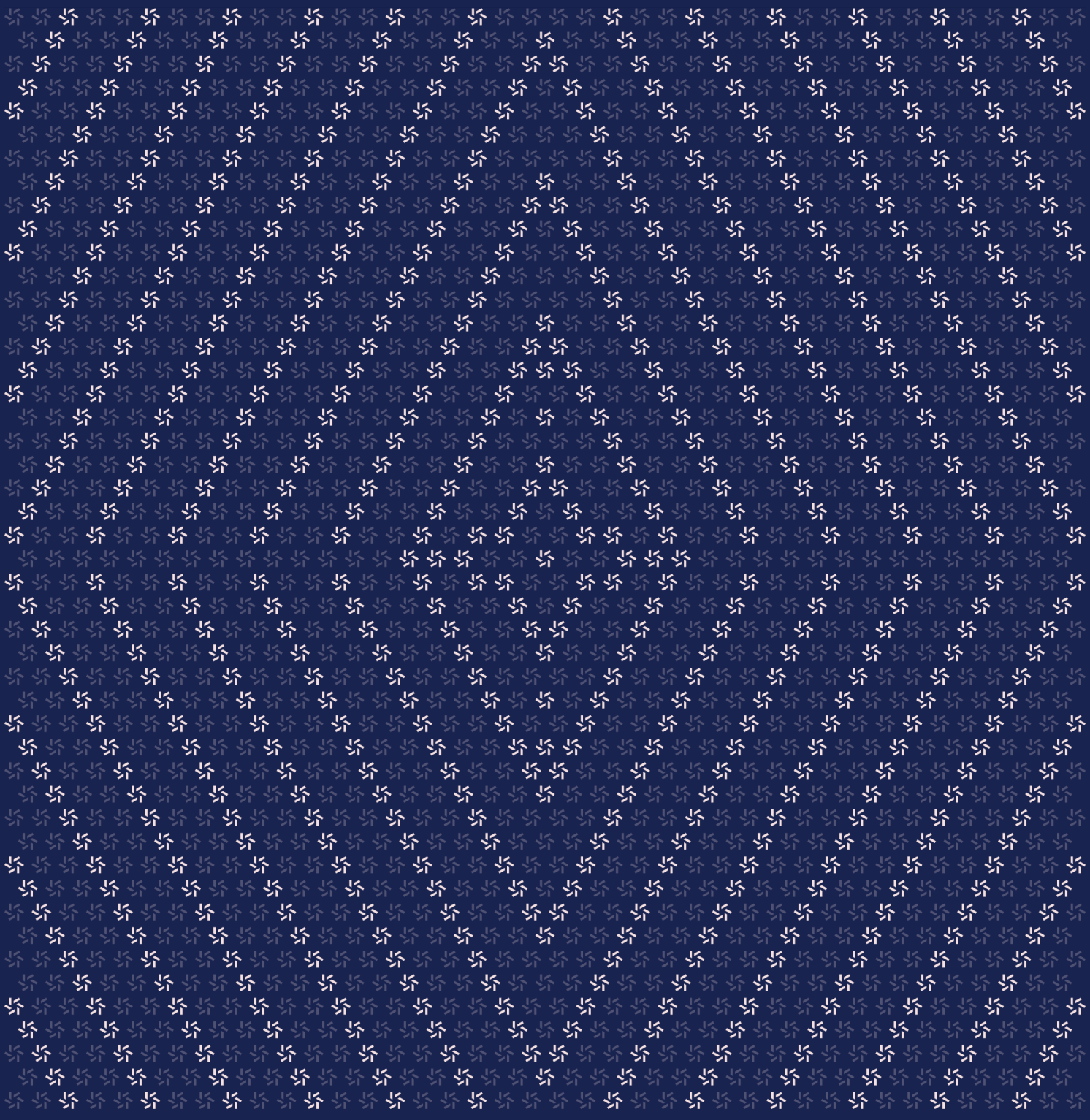
Prepared for
Filipp Makarov
Biconomy Labs

Prepared by
Filippo Cremonese
Zellic

January 3, 2024

AccountRecoveryModule

Smart Contract Security Assessment



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 January 2nd to January 3rd, 2024. During this engagement, Zellic reviewed AccountRecoveryModule's code for security vulnerabilities, design issues, and general weaknesses in security posture. The review targeted changes made to the account-recovery module since our previous review.

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 the account-recovery flow working and secure?
- Is the account-recovery module's `validateUserOp` function only authorizing the intended transactions?
- Is the security delay effective?

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.

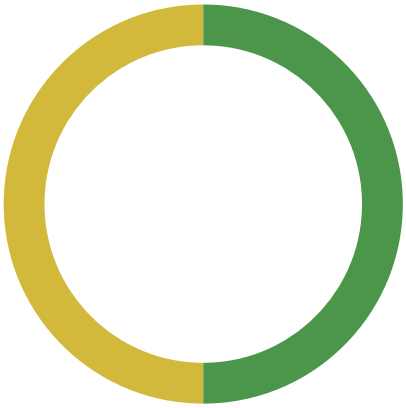
1.3. Results

During our assessment on the scoped AccountRecoveryModule contracts, we discovered two findings. No critical issues were found. One finding was 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	0
High	0
Medium	1
Low	1
Informational	0



2. Introduction

2.1. About AccountRecoveryModule

AccountRecoveryModule is the module for the Biconomy Modular Smart Account, which allows for account recovery when a user loses access to the signing private key.

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:

AccountRecoveryModule Contracts

Repository	https://github.com/bcnmy/scw-contracts ↗
Version	scw-contracts: `13893770a3c76b7fa95ed69197ea38ce7869dbfc`
Program	AccountRecoveryModule
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with one consultant for a total of two person-days. The assessment was conducted over the course of two calendar days.

The assessment was limited to the changes made to the in-scope contracts since our previous review, which targeted commit `e66df039` ↗.

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:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 2, 2024	Start of primary review period
January 3, 2024	End of primary review period

3. Detailed Findings

3.1. Incorrect calldata deserialization

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

Description

The `_validateRequestToBeAdded` was introduced to validate calldata associated with recovery requests. The function tries to ensure that that the calldata invokes the `execute` or `execute_ncC` functions on the smart account. This will cause an internal call to be performed. The parameters supplied to `execute` are further validated to ensure that the target of the call is the account-recovery module and that the invoked function is `executeRecovery`.

The checks are implemented in YUL, manually parsing the calldata. The code makes an incorrect assumption about the layout of the calldata, which could lead to an incorrect deserialization. The error lies in this snippet of code:

```
assembly {
    //32(memory bytes array length) + 4(submitRecoveryRequest selector)
    + 32(offset) + 32(length)
    expectedExecuteSelector := mload(add(innerCallData, 0x64))
    expectedThisAddress := mload(add(innerCallData, 0x68))
}
```

The code is assuming that the calldata (supplied in the byte array `innerCallData` argument) layout is as follows:

```
0x0     innerCallData byte array length
0x20    innerCallData data
0x20            submitRecoveryRequest selector
0x24    0x00    recoveryCallData offset -> assumed to be 0x20
0x44    0x20    recoveryCallData length
0x64    0x40    recoveryCallData data
0x64            0x0     expectedExecuteSelector
0x68            0x4     0x00    expectedThisAddress
0x88            0x24    0x20    [execute `value`]
0xa8            0x44    0x40    executeRecoveryCallDataOffset
0xc8            0x64    0x60    [execute `func` length]
0xe8            0x84    0x80    [execute `func` data]
...
```

The code is incorrectly assuming that the calldata is laid out so that the data of `recoveryCallData` starts immediately after the bytes specifying where such data is located. While this is normally true, this is not necessarily the case, as an unusual or future compiler or an attacker could construct valid calldata with different layouts.

Impact

This issue could allow to bypass the checks performed by `_validateRequestToBeAdded`; manually crafted calldata could contain data that would pass the checks at the hardcoded incorrect offset, while placing the real `recoveryCallData` further in the calldata byte array, bypassing `_validateRequestToBeAdded` entirely.

This would allow to bypass the intended `execute -> executeRecovery` flow and to invoke any other function directly, allowing to bypass the limit on the number of allowed recoveries.

Recommendations

Do not assume the offset of `recoveryCallData` to be fixed to `0x20`; instead, strictly follow the Solidity ABI.

Since this code is not intended to be used frequently, we advise to reevaluate the tradeoff made for choosing to use YUL assembly over plain `abi.decode`. While handcrafted YUL may bring some minor gas savings, this introduces some additional complexity, which increases the risk of security issues.

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit `a22e3d4d` ↗.

3.2. Out-of-bounds read

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

Description

The `validateUserOp` function reads the `innerSelector` from a bytes array using inline YUL without first checking the array length.

```
(address dest, uint256 callValue, bytes memory innerCallData) = abi
    .decode(
        userOp.callData[4:], // skip selector
        (address, uint256, bytes)
    );
bytes4 innerSelector;
assembly {
    innerSelector := mload(add(innerCallData, 0x20))
}
```

Impact

The issue does not appear to be exploitable. However, this may be dependant on the specific bytecode emitted by the compiler, and this issue could lead to unpredictable results.

Recommendations

Check the length of the `innerCallData` array before reading it.

Remediation

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

- a22e3d4d ↗
- 7b853b9c ↗

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.

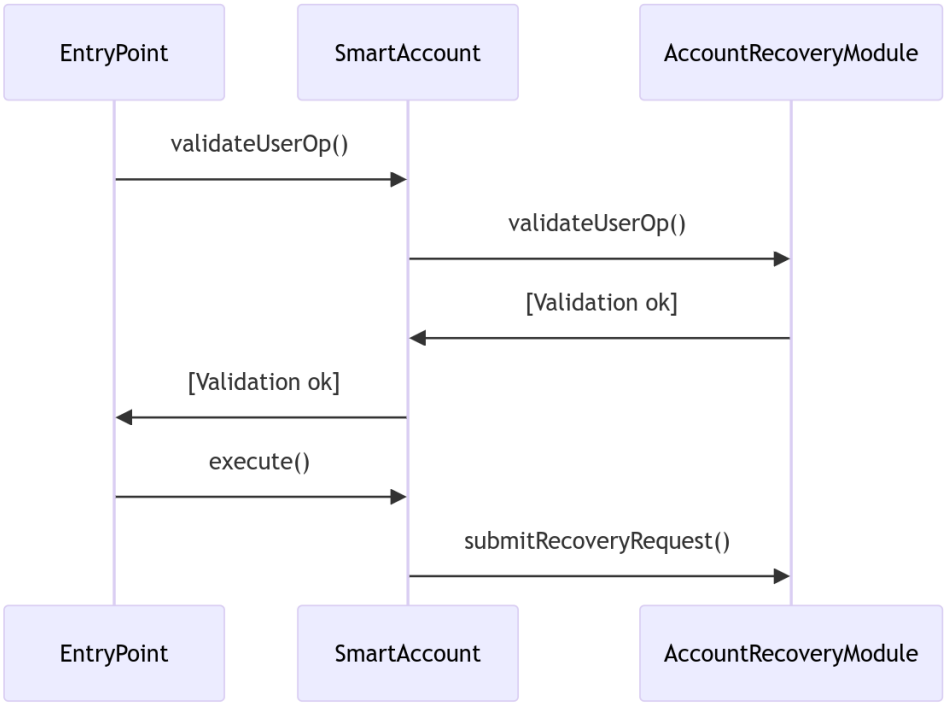
4.1. Notable changes

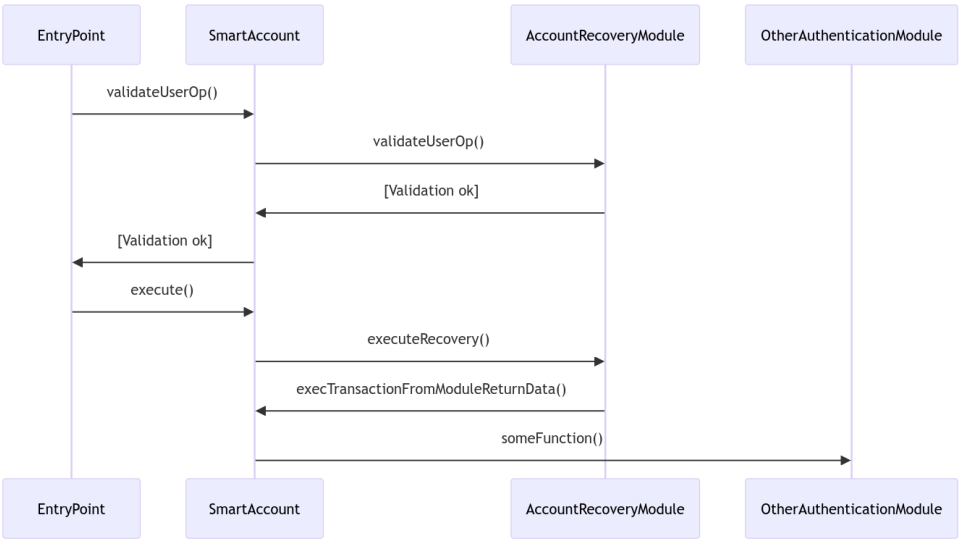
This section documents the notable changes applied to the in-scope code from commit `e66df039` ↗ to commit `13893770` ↗ (the latest commit of branch `feat/SMA-28-Account-Recovery-Kawach-Audit` at the time this engagement started).

New recovery-execution mechanism

The recovery-execution flow was changed, introducing a new `executeRecovery` function. The recovery flow must use this function, which checks and updates the number of recoveries left on the smart account being recovered. This section documents the updated recovery-execution flow.

The recovery process happens in two steps, represented in the following two diagrams. The first step is needed to submit a recovery request and the second to execute the recovery. Both steps are initiated by submitting a `UserOperation` to the regular ERC-4337 entry-point contract.


EntryPoint
SmartAccount
AccountRecoveryModule
validateUserOp()
validateUserOp()
[Validation ok]
[Validation ok]
execute()
submitRecoveryRequest()
EntryPoint
SmartAccount
AccountRecoveryModule


EntryPoint
SmartAccount
AccountRecoveryModule
OtherAuthenticationModule
validateUserOp()
validateUserOp()
[Validation ok]
[Validation ok]
execute()
executeRecovery()
execTransactionFromModuleReturnData()
someFunction()
EntryPoint
SmartAccount
AccountRecoveryModule
OtherAuthenticationModule

The first step involves submitting a transaction to the common entry-point contract, which will check for authorization by invoking the smart account (SA) contract `validateUserOp` function, which in turn will invoke the account-recovery module (ARM) `validateUserOp` function. The `validateUserOp` performs the required checks (e.g., calldata validation and guardian signature threshold validation), which allows to submit a transaction that invokes `SA:execute -> ARM:submitRecoveryRequest`. If the checks are successful, the entry point invokes the SA with the validated calldata. This call records the pending recovery request, together with the hash of the calldata that will be used to execute the recovery request. The calldata is also validated to ensure the next step happens as intended.

In the second step, the user submits a second transaction to the entry-point contract, which will again verify authorization by invoking `SA:validateUserOp -> ARM:validateUserOp`. The `ARM:validateUserOp` authorizes execution if a corresponding recovery request was previously recorded (by matching the calldata to be used for the recovery transaction) and a configurable security delay has elapsed. Finally, `SA:execute` is invoked by the entry point, which will cause the chain of calls `SA:execute -> ARM:executeRecovery -> SA:execTransactionFromModuleReturnData -> [recovery call]`.

Maximum number of recoveries

A new `recoveriesLeft` configuration parameter was introduced, limiting the number of recoveries that can be done on a smart account.

The parameter is checked to be greater than zero when submitting a recovery request, and it is decremented when a recovery is executed (reverting in case of an underflow). It is initially set when initializing the module and can be reset by invoking `setAllowedRecoveries` from the smart account contract (thus resetting the parameter that requires control of the account).

New function `resetModuleForCaller`

This function was introduced to allow resetting the module configuration for a smart account, removing all guardians and configuration. The function must be invoked directly from the smart account it acts on; therefore, it can only be invoked with control of the smart account.

If called, the function effectively disables the account-recovery module for the smart account that invokes it.

Deny new recovery requests if one already exists

The `submitRecoveryRequest` function was changed to reject submissions of new recovery requests if one was already submitted.

Other minor differences

This subsection documents minor changes that do not warrant a more extensive discussion.

In `validateUserOp`, `msg.sender` was replaced by `userOp.sender`; this change is harmless, since the two values are equal during normal operation. A user can invoke `validateUserOp` with arbitrary parameters, but no state changes are performed.

Signature validation performed in `validateUserOp` was refactored to be contained into a dedicated function `_validateGuardiansSignatures`.

Minor suggestions

This subsection documents very minor suggestions for improvement; these do not constitute security issues.

- Both `EXECUTE_SELECTOR` and `EXECUTE_OPTIMIZE_SELECTOR` could be initialized directly instead of being initialized via a value supplied to the constructor.
- The `MODULE_SIGNATURE_OFFSET` constant is not used; it looks like it was intended to be used in `validateUserOp`.
    - This item was addressed in commit `653bfaf8` ↗.
- The `guardians signatures are also validated at this point` comment in `validateUserOp` is likely misplaced.

5. Assessment Results

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

During our assessment on the scoped AccountRecoveryModule contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact. Biconomy Labs acknowledged all findings and implemented fixes.

5.1. Disclaimer

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

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

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