Zellic

Prepared for
Ankur Dubey
Sachin Tomar
Biconomy

Prepared by
Ayaz Mammadov
Sina Pilehchiha
Zellic

November 29, 2023

Security Policy

Smart Contract Security Assessment

Contents

About Zellic **3**

1. Executive Summary **3**

1.1. Goals of the Assessment 4

1.2. Non-goals and Limitations 4

1.3. Results 4

2. Introduction **5**

2.1. About Security Policy 6

2.2. Methodology 6

2.3. Scope 8

2.4. Project Overview 8

2.5. Project Timeline 9

3. Discussion **9**

3.1. Return datasize unchecked 10

3.2. Bypassing Security Policies 11

4. Threat Model **11**

4.1. Module: ERC7484SecurityPolicy.sol 12

4.2. Module: SecurityPolicyManagerPlugin.sol 12

5. Assessment Results **17**

5.1. Disclaimer 18

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 from November 21st to November 24th, 2023. During this engagement, Zellic reviewed Security Policy'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:

- Are the enabled security policies bypassable in any way?
- Could any entity apart from the relevant SA enable or disable security policies?
- Are there any errors in the assembly blocks used in `check*` functions?

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 Security Policy contracts, there were no security vulnerabilities discovered.

Zellic recorded its notes and observations from the assessment for Biconomy's benefit in the Discussion section (3. ↗).

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Security Policy

Security Policies allow SA users to enforce arbitrary restrictions on which modules can be installed on their SAs. The checks are implemented as on-chain contracts called “Security Policies”, which can be individually enabled by SAs.

Once enabled, any module installed on the SA must satisfy all enabled Security Policies. Security Policies can also leverage a Module Registry and use the data stored in attestations to verify conditions on the modules to be installed.

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 (3. ↗) 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:

Security Policy Contracts

Repository	https://github.com/bcnmy/scw-contracts ↗
Version	scw-contracts: `dfd6fad093f5c6904f8c0851b4f873d64391cdbb`
Programs	• ERC7484SecurityPolicy • SecurityPolicyManagerPlugin
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 person-week. The assessment was conducted over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov
Engineer
ayaz@zellic.io ↗

Sina Pilehchiha
Engineer
sina@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 1, 2023	Kick-off call
September 1, 2023	Start of primary review period
September 7, 2023	End of primary review period

3. 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.

3.1. Return datasize unchecked

In several snippets of YUL, external functions are called and data is loaded from the return data, but there are no checks on the length of the return data. As a consequence, irrelevant data is loaded into variables / returned. This mostly occurs when returning the `module` address.

```
function checkSetupAndEnableModule(
    address, //setupContract
    bytes calldata //setupData
) external override returns (address) {

    ...
    // copy the returndata to ptr
    let size := returndatasize()
    returndatacopy(ptr, 0, size)

    switch success
    case 0x1 {
        moduleInstallationSuccess := mload(ptr)
        module := mload(add(ptr, 0x60))
    }
    case 0x0 {
        revert(ptr, size)
    }
    ...
}
```

In the example above, the call to `execTransactionFromModuleReturnData` is guaranteed to return a `module` address; however, the source of that `module` address has the same return issue when setting up the module. However, we do not consider this to be a security issue because all modules are attested to be created by the Biconomy team and therefore should return the correct values.

This issue has been acknowledged by Biconomy, and a fix was implemented in commit a2576d6b ↗.

3.2. Bypassing Security Policies

Biconomy expressed their concern that Security Policies should not be bypassable. After a thorough review of the specified code and relevant tests regarding the `SecurityPolicyManagerPlugin` contract, we did not identify any instances in which a Security Policy is enabled by a user, but is not called by the Security Policy Manager plugin.

In multiple tests provided by Biconomy, including `testEnable*` and `testDisable*` functions, along with their negative scenario counterparts, we found that the ideal end state of enabling or disabling a Security Policy is consistently achieved and validated, without any unexpected side effects.

4. 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.

4.1. Module: ERC7484SecurityPolicy.sol

Function: `setConfiguration(Configuration _config)`

This sets the configuration for the plug-in for a given account.

Inputs

- `_config`
    - **Control**: Full.
    - **Constraints**: N/A.
    - **Impact**: The configuration to set.

Branches and code coverage (including function calls)

Intended branches

- Properly sets the configuration.
    - ☑ Test coverage

Negative behavior

- Should revert if not configured.
    - ☑ Negative test

4.2. Module: SecurityPolicyManagerPlugin.sol

Function: `checkAndEnableModule(address _module)`

This checks and enables a module without setting up.

Inputs

- `_module`
    - **Control**: Full.
    - **Constraints**: Must be a contract && must be attested.

- **Impact**: The module to be checked and enabled.

Branches and code coverage (including function calls)

Intended branches

- Validates the module against the registry using the `check*` functions.
  - ☑ Test coverage
- Calls the ModuleManager, calling `execTransactionFromModule` -> `enableModule`.
  - ☑ Test coverage

Function call analysis

- `_module.isContract()`
  - **What is controllable?** `_module`.
  - **If return value controllable, how is it used and how can it go wrong?** Is a contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `call(gas(), caller()...)`
  - **What is controllable?** Everything except the calldata used for the call to `execTranscationFromModule`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `checkSetupAndEnableModule(address None, byte[] None)`

This instructs the SA to install the module and return the address.

Inputs

- `setupContract`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.
  - **Impact**: Address of the `setupContract`.
- `setupData`
  - **Control**: Full.
  - **Constraints**: Must be valid calldata.
  - **Impact**: Calldata for the `sa.setupAndEnableModule` call.

Branches and code coverage (including function calls)

Intended branches

- Validates the module against the registry using the `check*` functions.
  - ☑ Test coverage
- Calls the ModuleManager, calling `execTransactionFromModule` -> `enableModule`.
  - ☑ Test coverage

Negative behavior

- Should revert if the security policy is not satisfied.
  - ☑ Negative test
- Should revert if the module is not a contract.
  - ☑ Negative test
- Should revert if `ExecTxFromModule` fails.
  - ☑ Negative test
- Should revert if setup fails.
  - ☑ Negative test

Function call analysis

- `_module.isContract()`
  - **What is controllable?** `_module`.
  - **If return value controllable, how is it used and how can it go wrong?** Is a contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `call(gas(), caller()...)`
  - **What is controllable?** Everything except the calldata used for the call to `execTranscationFromModule`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `disableSecurityPoliciesRange(ISecurityPolicyPlugin _start, ISecurityPolicyPlugin _end, ISecurityPolicyPlugin _pointer)`

This disables security policies, removing them from the linked list.

Inputs

- `_start`
  - **Control**: Full.

    - **Constraints**: `!= SENTINEL && != 0x0`.
    - **Impact**: Marks the beginning of the range of security policies to be disabled.
- `_end`
    - **Control**: Full.
    - **Constraints**: `!= SENTINEL && != 0x0`.
    - **Impact**: Marks the end of the range of security policies to be disabled.
- `_pointer`
    - **Control**: Full.
    - **Constraints**: Must point to a valid policy.
    - **Impact**: Reference point in the linked list.

Branches and code coverage (including function calls)

Intended branches

- Properly disable enable p1–p4 policies.
    - ✓ Test coverage

Negative behavior

- Should revert with invalid range.
    - ✓ Negative test
- Should revert with invalid pointer.
    - ✓ Negative test

Function: `disableSecurityPolicy(ISecurityPolicyPlugin _policy, ISecurityPolicyPlugin _pointer)`

This disables a security policy, removing it from the linked list.

Inputs

- `_policy`
    - **Control**: Full.
    - **Constraints**: `!= SENTINEL && != 0x0`.
    - **Impact**: The policy.
- `_pointer`
    - **Control**: Full.
    - **Constraints**: Must point to a valid policy.
    - **Impact**: Reference point in the linked list to remove `_policy`.

Branches and code coverage (including function calls)

Intended branches

- Properly disable/enable p1 policy.
  - ✓ Test coverage

Negative behavior

- Security Policy cannot be `0x0`.
  - ✓ Negative test
- Security Policy cannot be `SENTINEL`.
  - ✓ Negative test
- Disabled security policy cannot be disabled.
  - ✓ Negative test
- Reverts in the case of invalid pointer.
  - ✓ Negative test

Function: `enableSecurityPolicies(ISecurityPolicyPlugin[] _policies)`

This enables security policies, inserting into the linked list.

Inputs

- `_policies`
  - **Control**: Full.
  - **Constraints**: `policies[x] != SENTINEL && policies[x] != 0x0`.
  - **Impact**: The policies.

Branches and code coverage (including function calls)

Intended branches

- Policies are added into the linked list in order.
  - ✓ Test coverage
- Length should increase in accordance to the number of policies added.
  - ✓ Test coverage
- Works twice in a row.
  - ✓ Test coverage

Negative behavior

- Inserted Security Policies cannot be the sentinel address.
  - ✓ Negative test
- Inserted Security Policies cannot be `0x0`.

  - ☑ Negative test
- Can fully remove all the policies.
  - ☑ Negative test

Function: `enableSecurityPolicy(ISecurityPolicyPlugin _policy)`

This enables a security policy, inserting into the linked list.

Inputs

- `_policy`
  - **Control**: Full.
  - **Constraints**: `!= SENTINEL && != 0x0`.
  - **Impact**: The policy.

Branches and code coverage (including function calls)

Intended branches

- `enabledSecurityPolicies[policy]` is the previous head.
  - ☑ Test coverage
- `enabledSecurityPolicies[SENTINEL_MODULE_ADDRESS] == policy`.
  - ☑ Test coverage

Negative behavior

- Inserted Security Policy cannot be the sentinel address.
  - ☑ Negative test
- Inserted Security Policy cannot be `0x0`.
  - ☑ Negative test

5. Assessment Results

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

There were no findings during our audit.

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.