Zellic

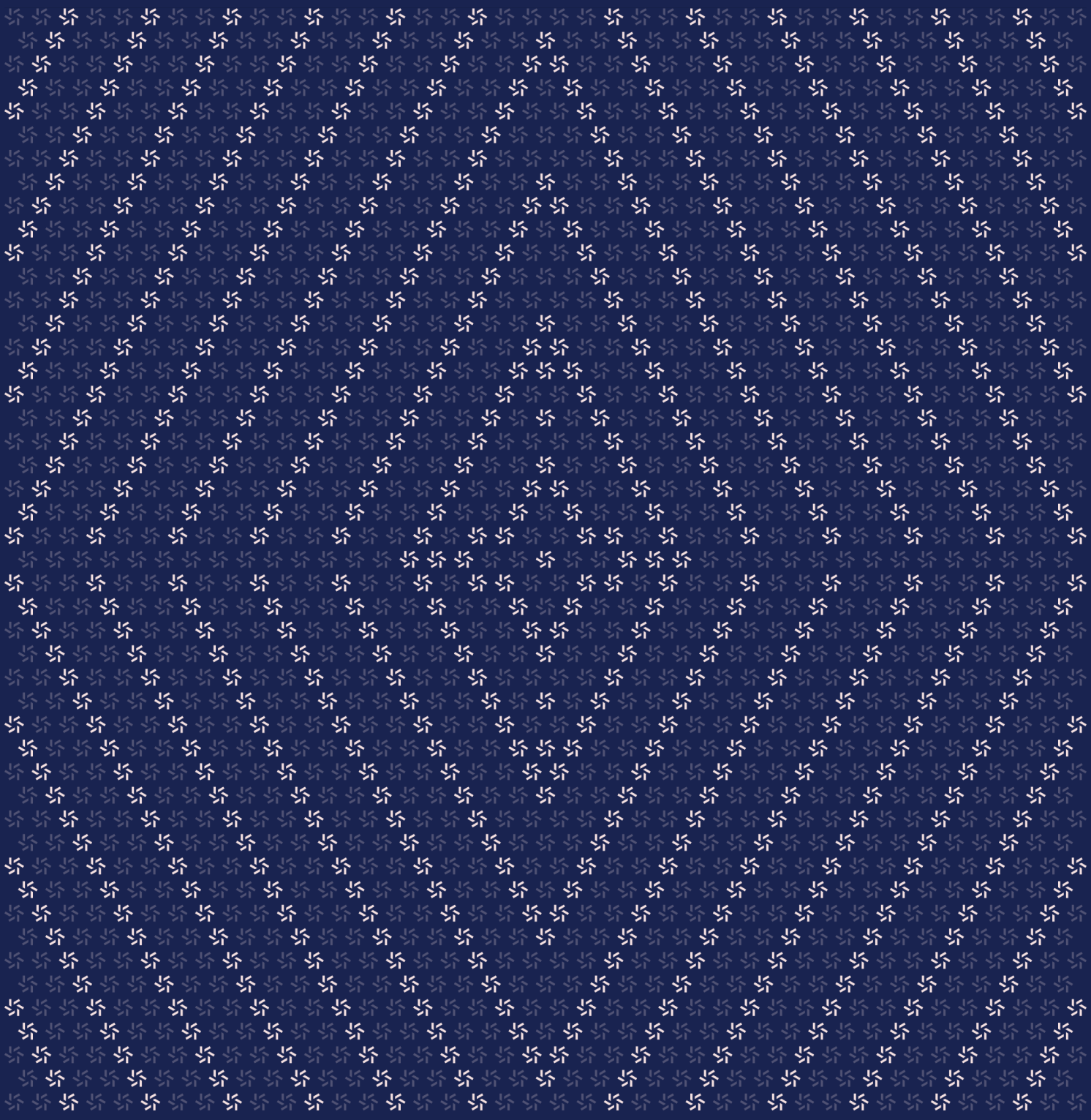
Prepared for
Chirag Titiya
Biconomy Labs

Prepared by
Syed Faraz Abrar
Ayaz Mammadov
Zellic

November 14, 2023

SponsorshipPaymaster

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 9th to November 10th, 2023. During this engagement, Zellic reviewed SponsorshipPaymaster'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 price markup logic sound?
- Does the validation logic miss any edge cases?
- Is there appropriate test coverage for all functions in the new contract?

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.

During this assessment, the short two-day time frame prevented us from fully reviewing the test coverage of the newly added code.

1.3. Results

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

After the conclusion of this audit, in commit `1f940bdd` ↗ Biconomy Labs increased the default value of `unaccountedEPGasOverhead` in the constructor of SponsorshipPaymaster. No security issues have been identified in association with this particular update.

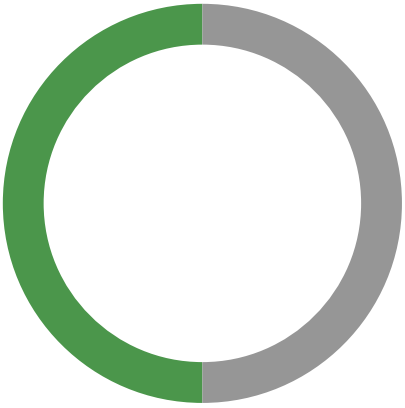
Please be aware that in the initial audit commit `013348cd` ↗, the file SponsorshipPaymaster was originally named VerifyingSingletonPaymasterV2. Subsequently, it underwent a renaming

process in commit 45be83fe ↗. To maintain consistency, we have revised this report to align with the updated filename.

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	0
Low	1
Informational	1



2. Introduction

2.1. About SponsorshipPaymaster

SponsorshipPaymaster is a token-based Paymaster that allows user to pay gas fees in ERC-20 tokens. Paymasters can sponsor transaction fees for contract accounts. The ERC-4337 Entry-point verifies whether the Paymaster has a sufficient deposit or if the contract account holds enough funds to cover gas fees. During execution, if a Paymaster is involved, it can implement custom fee logic.

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:

SponsorshipPaymaster Contracts

Repository	https://github.com/bcnmy/biconomy-paymasters ↗
Version	biconomy-paymasters: `013348cd54ac77424319f7f4ae7ea9f88d160534`
Program	SponsorshipPaymaster
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four 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:

Syed Faraz Abrar
Engineer
faith@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

November 9, 2023	Start of primary review period
November 10, 2023	End of primary review period

3. Detailed Findings

3.1. No enforced minimum value on fixedPriceMarkup

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

Description

The `setFixedPriceMarkup()` function is used to set the `fixedPriceMarkup` storage variable. This variable is a multiplier that is used to calculate the marked-up `requiredPrefund` amount in `_validatePaymasterUserOp()` that a user must prefund the Paymaster with prior to submitting any user operations.

In this case, the `setFixedPriceMarkup()` enforces a maximum value of 2e6 (i.e., a 2x multiplier) but does not enforce a minimum value.

Impact

If the owner accidentally sets the `fixedPriceMarkup` to a value less than 1e6, `_validatePaymasterUserOp()` will fail anytime a `priceMarkup` between `[0, 1e6)` is used. This is due to the following code within `_validatePaymasterUserOp()`:

```
require(priceMarkup <= 2e6, "Verifying PM:high markup %");

uint32 dynamicMarkup = MathLib.maxuint32(priceMarkup, fixedPriceMarkup);
require(dynamicMarkup >= 1e6, "Verifying PM:low markup %");
```

Recommendations

Enforce a minimum value of 1e6 for `fixedPriceMarkup` in `setFixedPriceMarkup()`.

Remediation

Biconomy Labs implemented a fix for this issue in commit 6074b93 ↗.

3.2. The `verificationGasLimit` is not checked during validation

Target	SponsorshipPaymaster		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The amount of gas passed into the `_validatePaymasterUserOp()` function is equal to the `userOp.verificationGasLimit`. This same gas limit is later used when the `_postOp()` function is called. However, `_validatePaymasterUserOp()` does not check to ensure that `userOp.verificationGasLimit` is high enough to handle the call to `_postOp()`.

Impact

If `_validatePaymasterUserOp()` succeeds, and execution then fails in the call to `_postOp()`, the Entrypoint contract will revert the entire transaction, so the whole bundle of user operations being executed would revert.

Recommendations

Enforce a minimum value for `userOp.verificationGasLimit` in `_validatePaymasterUserOp()`. This will ensure that the bundle containing this user operation is invalidated sooner, saving on gas fees.

Remediation

The client stated that the `verificationGasLimit` is checked in the out of scope Entrypoint contract. They have also added an integration test for this case in `test/bundler-integration/sponsorship-paymaster/biconomy-verifying-paymaster-v2-specs.ts` at commit c7dc1dff ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1. No domain separator for signed message

In SponsorshipPaymaster, the hash of the data is calculated as such:

```
function getHash(/* ... */) public view returns (bytes32) {
    //can't use userOp.hash(), since it contains also the paymasterAndData
    itself.
    return
        keccak256(
            abi.encode(
                userOp.getSender(),
                userOp.nonce,
                userOp.initCode,
                userOp.callData,
                userOp.callGasLimit,
                userOp.verificationGasLimit,
                userOp.preVerificationGas,
                userOp.maxFeePerGas,
                userOp.maxPriorityFeePerGas,
                block.chainid,
                address(this),
                paymasterId,
                validUntil,
                validAfter,
                priceMarkup
            )
        );
}
```

There is no domain separator in the signed message structure. Therefore, if another protocol had a signed message structure similar to the one in this message, then there could possibly be a replay attack.

However, due to the inclusion of `chainid` in the hash, cross-chain replay attacks are mitigated. Additionally, the data includes many unique parameters, such as `initCode`, which holds `bytes` for the construction of the `account contract`; therefore, it is highly improbable that a collision may occur.

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.

> Please note that our threat model was based on commit 013348cd ↗., which represents a specific snapshot of the codebase. Therefore, it's important to understand that the absence of certain tests in our report may not reflect the current state of the test suite.
>
> During the remediation phase, Biconomy Labs took proactive steps to address the findings by adding test cases where applicable in commit c7dc1dff ↗. This demonstrates their dedication to enhancing the code quality and overall reliability of the system, which is commendable.

5.1. Module: SponsorshipPaymaster.sol

Function: `depositFor(address paymasterId)`

This function deposits into the entry point and add to `paymasterID`'s balance.

Inputs

- `paymasterId`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Adds to user's balance and deposits into the entry point.

Branches and code coverage (including function calls)

Intended branches

- Deposits into the specified address
  - ☐ Test coverage

Negative behavior

- Does not allow 0 value deposits
  - ☐ Negative Test

Function call analysis

- `depositFor -> entrypoint.depositTo`
  - **What is controllable?** Everything.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

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

The function that the Paymaster calls to send Ether from the entry point to the target address.

Inputs

- `withdrawAddress`
  - **Control**: Full.
  - **Constraints**: `!= 0`.
  - **Impact**: The target to receive the Ether (probably the submitter).
- `amount`
  - **Control**: Full.
  - **Constraints**: `paymasterIdBalances[msg.sender] >= amount`.
  - **Impact**: The amount to send.

Branches and code coverage (including function calls)

Intended branches

- Withdraws Eth to the specified address
  - ☑ Test coverage

Negative behavior

- Reverts when paymasterIdBalance is not enough
  - ☐ Negative test

Function call analysis

- `withdrawTo -> entrypoint.withdraw`
  - **What is controllable?** Everything.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `parsePaymasterAndData(byte[] paymasterAndData)`

This function is used to parse the `paymasterAndData` field of the `UserOperation` struct.

Inputs

- `paymasterAndData`
  - **Control**: Fully controlled by the user.
  - **Constraints**: It must be data in a valid format, where the data from the `VALID_PND_OFFSET`-1 to the `VALID_PND_OFFSET` should represent the `priceSource`. The data from the `VALID_PND_OFFSET` to the `SIGNATURE_OFFSET` is the ABI-encoded `validUntil`, `validAfter`, `feeToken`, `oracleAggregator`, `exchangeRate`, and `fee`. Data following the `SIGNATURE_OFFSET` position is a valid signature.
  - **Impact**: This is the structural data to be parsed.

Branches and code coverage (including function calls)

Intended branches

- Succeeds with parsing data properly.
  - ☑ Test coverage

Negative behavior

- Invalid `paymasterAndData` causes revert.
  - ☐ Negative test

Function: `_postOp(PostOpMode mode, bytes calldata context, uint256 actualGasCost)`

This function executes the Paymaster's payment conditions.

Inputs

- `mode`
  - **Control**: Not controlled by the user.
  - **Constraints**: Must be one of these: `opSucceeded`, `opReverted`, or `postOpReverted`.

    - **Impact**: Used to determine the state of the operation. Unused in this instance.
- `context`
    - **Control**: Partially controlled by the user.
    - **Constraints**: N/A.
    - **Impact**: This contains the payment conditions signed by the Paymaster.
- `actualGasCost`
    - **Control**: Not controlled by the user.
    - **Constraints**: N/A.
    - **Impact**: This is the gas amount that is paid to the Entrypoint.

Branches and code coverage (including function calls)

Intended branches

- Succeeds with valid arguments.
    - ☑ Test coverage

Negative behavior

- Should revert if enough gas it not provided through `verificationGasLimit`.
    - ☐ Negative test

Function: `_validatePaymasterUserOp(UserOperation calldata userOp, bytes32 userOpHash, uint256 requiredPreFund)`

This function is used to verify that the `UserOperation`'s Paymaster data were signed by the external signer.

Inputs

- `userOp`
    - **Control**: Fully controlled by user.
    - **Constraints**: All fields are used in signature validation and thus must be valid.
    - **Impact**: This is the `UserOperation` being validated.
- `requiredPreFund`
    - **Control**: Partially controlled by user.
    - **Constraints**: Must be sufficient to pay for the gas fees for this user operation.
    - **Impact**: This is the required amount of prefunding for the Paymaster, calculated using the `userOp` argument.

Branches and code coverage (including function calls)

Intended branches

- Succeeds with valid gas limit, `userOp`, and `requiredPrefund`.
  - ☑ Test coverage

Negative behavior

- Invalid signature causes error to be returned.
  - ☑ Negative test
- Insufficient `requiredPrefund` causes revert.
  - ☐ Negative test
- Insufficient `userOp.verificationGasLimit` causes revert.
  - ☐ Negative test
- Parsing invalid Paymaster data causes revert.
  - ☐ Negative test

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 SponsorshipPaymaster contracts, we discovered two findings. No critical issues were found. One finding was of low 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.