Security Review Report
NM-0453 Pimlico

NETHERMIND SECURITY

(Mar 10, 2025)

Contents

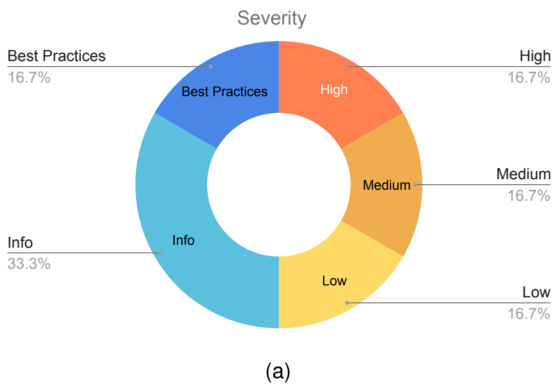
1 Executive Summary

This document presents the security review performed by Nethermind Security of the Pimlico's Singleton Paymaster contracts. The review focused on the protocol's implementation of an ERC-4337-compatible paymaster designed to facilitate gas sponsorship for account abstraction transactions. The Singleton Paymaster provides two primary modes for sponsoring gas costs: one based on a user's Pimlico balance (off-chain) and another enabling gas payments through ERC-20 tokens. The security review evaluated the integrity of these mechanisms, ensuring that the protocol securely handles transaction validation and gas sponsorship in both modes.

The audited code comprises 756 lines of code written in the Solidity language. The audit focused on the Paymaster V6 and V7 contracts.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report** six points of attention, where one is classified as `High`, one is classified as `Medium`, one is classified as `Low`, and three are classified as `Informational` or `Best Practice` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation. Section 9 concludes the document.



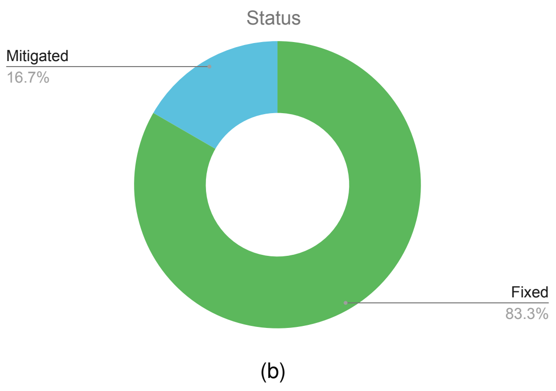


Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (2), **Best Practices** (1).
Distribution of status: Fixed (5), **Acknowledged** (0), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 7, 2025
Final Report	March 10, 2025
Repository	singleton-paymaster
Start Commit	755a9d5e71ffc650a92b714a043acb7c1dd4b89a
Final Commit	bc902248084fc966269075d5b8d984cef2b569b2
Documentation	Docs
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/SingletonPaymasterV7.sol	191	94	49.2%	45	330
2	src/SingletonPaymasterV6.sol	171	103	60.2%	45	319
3	src/base/MultiSigner.sol	23	20	87.0%	12	55
4	src/base/BaseSingletonPaymaster.sol	277	152	54.9%	65	494
5	src/base/ManagerAccessControl.sol	15	5	33.3%	6	26
6	src/base/BasePaymaster.sol	35	35	100.0%	11	81
7	src/interfaces/IPaymasterV6.sol	16	35	218.8%	3	54
8	src/interfaces/IPaymasterV7.sol	22	37	168.2%	3	62
9	src/interfaces/PostOpMode.sol	6	5	83.3%	1	12
	Total	**756**	**486**	**64.3%**	**191**	**1433**

3 Summary of Issues

	Finding	Severity	Update
1	Users undercharged due to erroneous cost calculation	High	Fixed
2	User is not charged when first `postOp` call reverts	Medium	Fixed
3	Paymaster and bundler financial loss risks in ERC20 mode	Low	Mitigated
4	Redundant signing of paymaster address	Info	Fixed
5	Unnecessary calculation of `preFund` amount	Info	Fixed
6	Unused imports	Best Practices	Fixed

4 System Overview

The Singleton Paymaster is an ERC-4337 compatible paymaster that facilitates gas sponsorship for account abstraction transactions. It provides two primary modes of operation: one that allows gas sponsorship based on a user's Pimlico balance (off-chain) and another that enables payments in ERC-20 tokens with a configurable exchange rate for gas tokens. The protocol consists of two main contracts: `SingletonPaymasterV6`, which supports the `EntryPoint` v6 contract, and `SingletonPaymasterV7`, which supports the `EntryPoint` v7 contract.

4.1 Actors

- **Users**: Individuals or institutions looking to have their transactions sponsored by Pimlico's paymaster contracts.
- **Signers**: Accounts operated by Pimlico's backend that sign user operations. Only signed operations can be sponsored.
- **Bundlers**: Block builder nodes that take signed user operations and submit them on-chain to the `EntryPoint` contract.
- **Admin**: Accounts with admin or manager role. Admins can update the whitelist of approved Bundlers and manage the set of Signers authorized to sign user operations.

4.2 Architecture

The system consists of two main contracts that facilitate account abstraction transactions: the `Paymaster` contract and the `EntryPoint` contract. The `EntryPoint` is a singleton contract, meaning there is a single deployment per network, similar to the wrapped Ether contract. Bundlers interact with the `EntryPoint` contract and submit user operations to it. While the `EntryPoint` drives the execution, the `Paymaster` contract covers the gas costs for these operations.

User operation execution consists of two phases:

- **Validation Phase**: When the bundler submits a user operation on-chain by calling `handleOps(...)` on the `EntryPoint` contract, the `EntryPoint` checks whether the `Paymaster` will cover the gas costs.
- **Execution Phase**: After validation, the operation is executed. The `EntryPoint` then calls the `Paymaster` again to finalize gas calculations and handle potential refunds.

4.3 Paymaster Operations

ERC-4337 paymaster implementations must include two key functions that are called by the `EntryPoint` contract: `validatePaymasterUserOp(...)` and `postOp(...)`.

- The **validatePaymasterUserOp(...)** function determines whether the `Paymaster` is willing to sponsor a given user operation. In Pimlico's `PaymasterV6` and `PaymasterV7` implementations, this function verifies that the user operation contains a valid signature from an authorized signer. It also checks which mode the user intends to use: verifying mode (off-chain Pimlico balance) or ERC-20 mode. If ERC-20 mode is selected, this function returns extra context, including gas approximations for use in the `postOp(...)` function.
- The **postOp(...)** function finalizes the sponsorship process and accounts for the total gas used in the transaction. In Pimlico's implementation, this function handles the ERC-20 payment logic, ensuring that the appropriate token amount is deducted and transferred. It also accounts for potential refunds and enforces penalties if necessary.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Users undercharged due to erroneous cost calculation

File(s): src/SingletonPaymasterV6.sol

Description: During a user operation execution, the paymaster's postOp flow is triggered when a non-empty context is provided by the validation step. The _postOp function is responsible for executing the postOp logic. It calculates the total cost of the userOp, including preOp, userOp execution, and an estimated postOp execution cost. This total cost is then converted into a token amount using the exchange rate provided in the UserOp struct, and the corresponding amount is transferred from the user to the treasury address.

```
function _postOp(PostOpMode _postOpMode, bytes calldata _context, uint256 _actualGasCost) internal {
    // ...
    ERC20PostOpContext memory ctx = _parsePostOpContext(_context);

    uint256 actualUserOpFeePerGas = _calculateActualUserOpFeePerGas(ctx.maxFeePerGas, ctx.maxPriorityFeePerGas);
    //@audit _actualGasCost should not be divided by actualUserOpFeePerGas
    uint256 costInToken = getCostInToken(
        _actualGasCost / actualUserOpFeePerGas, ctx.postOpGas, actualUserOpFeePerGas, ctx.exchangeRate
    ) + ctx.constantFee;

    bool success = SafeTransferLib.trySafeTransferFrom(ctx.token, ctx.sender, ctx.treasury, costInToken);
    // ...
}
```

The getCostInToken function computes the total cost in tokens, with the first parameter representing the actual gas cost (in wei) consumed by the userOp, excluding the postOp.

```
function getCostInToken(...) public pure returns (uint256) {
    return ((_actualGasCost + (_postOpGas * _actualUserOpFeePerGas)) * _exchangeRate) / 1e18;
}
```

However, the current implementation divides the _actualGasCost by the actualUserOpFeePerGas before passing it to getCostInToken. This results in the value representing gas units instead of the correct wei amount, leading to an incorrect estimation of the total gas cost. As a result, the gas cost is under-calculated, and users are charged less than what the paymaster actually pays.

Recommendation(s): Remove the erroneous division by actualUserOpFeePerGas in the first parameter passed to getCostInToken.

Status: Fixed

Update from the client: Fixed in commit: 56ee7d6.

6.2 [Medium] User is not charged when first `postOp` call reverts

File(s): `src/SingletonPaymasterV6.sol`

Description: The `SingletonPaymasterV6` contract implements the Pimlico paymaster, compatible with EntryPoint v6. In this version, the EntryPoint executes the user operation (`userOp`), followed by attempting to execute the `postOp` on the paymaster. The mode of the postOp can either be `opSucceeded` or `opReverted`. If the postOp execution reverts, the error is caught by EntryPoint, reverting the state to before the `userOp` execution, and a second call is initiated with the mode `postOpReverted`.

The postOp implementation in `SingletonPaymasterV6` attempts to charge the user for the actual cost of the `userOp`. However, during the second call (with `postOpReverted` mode), the function returns before the user has been charged. As a result, in cases where the `userOp` execution leads to a decrease in the user's token approval or balance (making it insufficient to cover the `userOp` cost), the second `postOp` call does not enforce the charge after the `userOp` state changes are reverted. Consequently, the paymaster ends up covering the gas fees.

```
function _postOp(PostOpMode _postOpMode, bytes calldata _context, uint256 _actualGasCost) internal {
    if (_postOpMode == PostOpMode.postOpReverted) {
        // @audit User is not charged yet due to the first call falling
        return;
    }
    // ...
    bool success = SafeTransferLib.trySafeTransferFrom(ctx.token, ctx.sender, ctx.treasury, costInToken);
    // ...
    emit UserOperationSponsored(ctx.userOpHash, ctx.sender, ERC20_MODE, ctx.token, costInToken, ctx.exchangeRate);
}
```

Recommendation(s): The `postOp` should charge the user even when the mode is `postOpReverted`.

Status: Fixed

Update from the client: Fixed in commit: `6fa0afa`.

6.3 [Low] Paymaster and bundler financial loss risks in ERC20 mode

File(s): `src/SingletonPaymasterV6.sol`, `src/SingletonPaymasterV7.sol`

Description: In the ERC20 mode, the Pimlico team performs an off-chain verification of the user's balance to ensure it can cover the userOp fees and prevent the execution from reverting. However, without on-chain verification during the validation phase, the user's balance or approvals to the Paymaster can be manipulated so that the user does not have the funds to pay for the operation in the postOp phase. The user's token balance/allowance can be manipulated at two key time points:

– During the execution of the userOp itself;

– Between the off-chain balance check and the point when the userOp is processed by the bundler (front-running the call to `handleOps(...)` function);

```
// SingletonPaymasterV7.sol
function _postOp(
    // ...
    uint256 _actualGasCost,
    uint256 _actualUserOpFeePerGas
)
    internal
{
    // ...
    uint256 costInToken = getCostInToken(
        actualGasCost, ctx.postOpGas,
        _actualUserOpFeePerGas, ctx.exchangeRate
    ) + ctx.constantFee;

    // @audit-issue If the user does not have enough tokens,
    // or the approval is insufficient, the post op reverts.
    SafeTransferLib.safeTransferFrom(
        ctx.token, ctx.sender, ctx.treasury, costInToken
    );
    // ...
}
```

Suppose the `_postOp(...)` execution fails due to the revert in `safeTransferFrom(...)` caused by insufficient balance or allowance. In that case, the execution will continue in the `_executeUserOp(...)` on the `Entrypoint` side, where it will finish without charging the user for the operation. Even though the action specified by the user in the userOp (the calldata portion) was reverted due to insufficient payment, the paymaster pays for the wasted gas and compensates the bundler.

The first scenario, where the userOp itself modifies the user's token balance to cause the `postOp(...)` to revert, is less likely since the bundler is expected to simulate the operation before submitting it on-chain. It can still happen if the bundler performs limited validation, e.g., does not simulate the execution itself but only the validation phase.

The second scenario, where the user front-runs the bundler's call to `handleOps(...)` is more likely to happen. While there is no direct incentive for the user to do so, a griefer or market competitor of Pimlico might want to cause a financial loss to the Paymaster at no upfront cost.

In certain cases, the attacker might also be able to shift the financial loss from the Paymaster to the Bundler. By deliberately causing `postOp(...)` to revert, the additional gas overhead from post-execution failure handling logic could exceed the prefunded amount. If the `Entrypoint` contract's gas prefund is insufficient to cover these costs, the Paymaster will be unable to compensate the Bundler, resulting in a financial loss for the Bundler. If this happens repeatedly, the Paymaster could be throttled or banned.

Since the attacker controls the gas limits and max gas fees, they also control the prefund taken by the `Entrypoint` contract. To exploit this, the attacker must set these parameters so that they pass the bundler's simulation under normal conditions but fail when additional failure-handling logic is triggered.

```
// Entrypoint.sol (V7)
function _postExecution(
    // ...
    UserOpInfo memory opInfo,
    uint256 actualGas
)
    private
    returns (uint256 actualGasCost)
{
    // ...
    actualGasCost = actualGas * gasPrice;
    // @audit The user controls the prefund amount. It is influenced by gas
    // limits and max gas fees selected.
    uint256 prefund = opInfo.prefund;
    if (prefund < actualGasCost) {
        // @audit This branch may be hit once the postOp reverts due
        // to payment failure caused by the user.
        if (mode == IPaymaster.PostOpMode.postOpReverted) {
            // @audit-issue The difference betwee actual gas cost and prefund
            // is the financial loss that the bundler will take.
            // The bundler is compensated with the actual gas cost.
            actualGasCost = prefund;
            emitPrefundTooLow(opInfo);
            emitUserOperationEvent(
                opInfo, false, actualGasCost, actualGas
            );
        }
        // ...
    }
}
```

Recommendation(s): Consider charging users in advance. This, however, may conflict with the current project requirement that allows users to pay for the operation only after execution, as they may need to acquire tokens first. An alternative approach is for the Paymaster to absorb the loss, relying on the fact that every protocol user must have a valid RPC URL, allowing for off-chain penalties. However, this would require the Paymaster to reserve additional tokens to cover cases where the Bundler incurs a loss.

Status: Mitigated

Update from the client: We have added optional prefund: PR 13. To clarify this commit: Since the suggestion is to charge users in advance, we've added an optional pre-fund option. If a user has a known malicious history, we'll require pre-funding during the validation phase to reduce the risk of losing reputation or money on-chain. If the user has no history of malicious behaviour, we'll allow sponsorship without pre-funding, but only up to a limit. This helps prevent our paymaster and us from being rigged by front-running. I also understand the audit's concern about the `prefund < actualGasCost` case, where the paymaster might be throttled due to losses incurred by the bundler. To address this, we've decided to include an additional `paymasterPostOpGasLimit` off-chain to account for the revert gas limit so that pre-fund is always enough.

6.4 [Info] Redundant signing of paymaster address

File(s): `src/SingletonPaymasterV6.sol`

Description: The paymaster backend signs the user operation along with other parameters, this signature is appended to the `userOp.paymasterAndData` field and is verified on-chain during the verification phase.

The `_getHash` function returns the hash of the structure signed by the backend. It hashes the full `userOp` structure, excluding the `paymasterAndData` field, which is hashed up to the signature bytes. This field contains the paymaster address (first 20 bytes) along with other parameters that vary based on the mode. After this, the hash is combined with the `chainId` and `address(this)`, which refers to the paymaster address. As a result, the paymaster address is signed twice: once as part of the `paymasterAndData` field and again when added separately with `address(this)`. This redundant signing unnecessarily increases gas costs.

```
function _getHash(UserOperation calldata _userOp, uint256 paymasterDataLength) internal view returns (bytes32) {
    bytes32 userOpHash = keccak256(
        abi.encode(
            // ...
            // hashing over all paymaster fields besides signature
            keccak256(_userOp.paymasterAndData[:PAYMASTER_DATA_OFFSET + paymasterDataLength])
        )
    );
    //@audit signing address(this) is redundant
    return keccak256(abi.encode(userOpHash, block.chainid, address(this)));
}
```

Recommendation(s): Remove the redundant inclusion of `address(this)` in the hashed data.

Status: Fixed

Update from the client: Fixed in commit: `3ea3940`.

6.5 [Info] Unnecessary calculation of `preFund` amount

File(s): `BaseSingletonPaymaster.sol`

Description: During the verification phase of an ERC20 mode userOp, a context is generated via the `_createPostOpContext` function. This context is then passed by the `EntryPoint` to the `postOp` function for further use in its logic.

The `_createPostOpContext` function populates various fields in the `ERC20PostOpContext` struct, eventually, it calculates the `preFund` amount that was paid by the paymaster using the `_getRequiredPrefund` function.

However, the actual `preFund` amount is already passed by the `EntryPoint` to the `validatePaymasterUserOp` function. Therefore, the computation in the paymaster contract is redundant.

```
function _validatePaymasterUserOp(
    UserOperation calldata _userOp,
    bytes32 _userOpHash,
    uint256 /* maxCost */ // @audit required preFund amount passed by the EntryPoint
)
    internal
    returns (bytes memory, uint256)
{
    // ...
    bytes memory context;
    uint256 validationData;

    if (mode == VERIFYING_MODE) {
        (context, validationData) = _validateVerifyingMode(_userOp, paymasterConfig, _userOpHash);
    }

    if (mode == ERC20_MODE) {
        //@audit context is generated by recomputing the required preFund amount
        (context, validationData) = _validateERC20Mode(_userOp, paymasterConfig, _userOpHash);
    }

    return (context, validationData);
}
```

Recommendation(s): Use the `maxCost` parameter directly to generate the ERC20 context, eliminating the redundant calculation of the `preFund` amount in the paymaster contract.

Status: Fixed

Update from the client: Fixed in commit `1ab35f0`.

6.6 [Best Practices] Unused imports

File(s): `src/*.sol`

Description: The following interfaces are imported but never used.

```
SingletonPaymasterV6.sol
    import { IEntryPoint } from "@account-abstraction-v6/interfaces/IEntryPoint.sol"
MultiSigner.sol
    import { IManagerAccessControl } from "./ManagerAccessControl.sol";
    import { IERC165 } from "@openzeppelin-v5.0.2/contracts/utils/introspection/IERC165.sol";
BasePaymaster.sol
    import { IManagerAccessControl } from "./ManagerAccessControl.sol";
    import { IERC165 } from "@openzeppelin-v5.0.2/contracts/utils/introspection/IERC165.sol";
    import { MultiSigner } from "./MultiSigner.sol";
```

Recommendation(s): Remove unused imports.

Status: Fixed

Update from the client: Fixed in commit: `6e0ad63`.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Pimlico's Documentation

The documentation for the Pimlico contracts was well-organized and readily accessible through their official documentation website and README.md file in the paymaster repository. The team also provided a comprehensive audit requirements document that helped in understanding the scope and key areas of the system.

The Pimlico team was highly responsive and proactive in clearing up any doubts raised during the review process, providing clear and timely answers during regular calls. Their assistance ensured a deep and thorough understanding of the protocol and its components.

The NatSpec comments within the code are detailed, offering clear explanations of the code's inner workings. This made navigating the codebase more accessible and understandable for Nethermind Security auditors.

One area for improvement is that while the documentation covers the core protocol well, reviewing and understanding the code requires some familiarity with the off-chain components and assumptions in the system. It would be helpful for the documentation to include more context regarding the operational environment, such as expected value ranges, gas limits, and maximum fees, as these assumptions play a crucial role in the system's functionality and performance.

8 Test Suite Evaluation

Pimlico's test suite is robust and clearly outlines the contract specifications. The test suite catches any change to the business logic. To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the results of the analysis performed on Pimlico's paymaster smart contracts. The first column lists the contracts that were tested using mutation testing. The second column indicates the number of mutants generated and how many were "slain" (i.e., caught by the test suite). The third column provides the percentage of mutants slain, reflecting the effectiveness of the test suite in covering the particular contract. The higher the score, the better the test suite is at finding bugs.

Contract	Mutants (slain / total generated)	Score
`SingletonPaymasterV6.sol`	99 / 99	100%
`SingletonPaymasterV7.sol`	125 / 125	100%
`BasePaymaster.sol`	11 / 11	100%
`BaseSingletonPaymaster.sol`	246 / 246	100%
`MultiSigner.sol`	9 / 9	100%
Total	490 / 490	100%

Remarks about Pimlico's Test Suite

The test suite for Pimlico's Paymaster contract demonstrates strong fundamental coverage, achieving a perfect 100% mutation testing score. This ensures that any small modifications to the contract's logic will be caught by the test suite, making it highly resistant to unintended regressions and enhancing the safety of future upgrades.

Despite this strong foundation, the test suite could be further improved by incorporating invariant testing, which would maintain state across multiple test runs. This approach would better simulate real-world usage, where multiple users interact with the Paymaster in parallel. By introducing such tests, the suite would be able to validate long-term correctness and detect potential issues that only arise over extended interactions, such as improper state transitions or accumulated side effects.

Enhancing the test suite with these improvements would provide greater assurance of correctness and security, ensuring that the Paymaster remains robust under diverse usage scenarios.

8.1 Automated Tools

8.1.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer