Kinto Access Security Analysis

by Pessimistic

This report is public

February 21, 2024

Abstract

In this report, we consider the security of smart contracts of Kinto Access project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

Summary

In this report, we considered the security of Kinto Access smart contracts. We described the audit process in the section below.

The initial audit showed one issue of medium severity: Overpowered owner. Also, several low-severity issues were found.

After the initial audit, the codebase was updated. The developers provided the comment on the Overpowered owner issue of medium-severity. They fixed most of the low-severity issues and increased the number of tests and the code coverage.

The overall code quality of the project is good.

General recommendations

We recommend fixing the mentioned issues before the deployment.

Project overview

Project description

For the audit, we were provided with Kinto Access project on a public GitHub repository, commit 1dc662324aa1ada7554be68bdae43b39a15c2b38.

The scope of the audit included:

- **src/access/AccessPoint.sol**;
- **src/access/AccessRegistry.sol**;
- **src/access/workflows/WithdrawWorkflow.sol**;
- **src/interfaces/IAccessPoint.sol**;
- **src/interfaces/IAccessRegistry.sol**;
- **src/paymasters/SignaturePaymaster.sol**.

The documentation for the project included this link.

All 207 tests pass successfully. The code coverage was 72.30%. The tests were executed for the entire project, while the code coverage was determined based on the audit scope.

The total LOC of audited sources is 321.

Codebase update #1

After the initial audit, the codebase was updated, and we were provided with commit 4ccf134dc10225fc69a8c7baa20fd6bd36d28228.

The developers commented on the medium-severity issue. Also, they fixed most of the low-severity issues. The number of tests increased to 335. All of them passed. The code coverage was 82.09%.

Audit process

We started the audit on February 2, 2024 and finished on February 7, 2024.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project. After a discussion, we performed preliminary research and started the review.

During the work, we stayed in touch with the developers and discussed confusing or suspicious parts of the code.

We manually analyzed all the contracts within the scope of the audit and checked their logic. Among others, we verified the following properties of the contracts:

- Only privileged account can enable and disable workflows;
- The user's signature and the paymaster's signature are verified correctly;
- The codebase does not contain unfinished functions or contracts;
- Functions revert with informative messages when incorrect arguments were supplied;
- There is no possibility to withdraw funds without the desire of the wallet's owner.

We scanned the project with the following tools:

- Static analyzer Slither;
- Our plugin Slitherin with an extended set of rules;
- Semgrep rules for smart contracts. We also sent the results to the developers in the text file.

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tools.

We made the recheck #1 from February 15 to February 19, 2024. We checked whether the developers fixed previous issues. Also, we ran the tests and calculated the code coverage. Finally, we updated the report.

Manual analysis

The contracts were completely manually analyzed, and their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

The audit showed no critical issues.

Medium severity issues

Medium severity issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Overpowered owner (commented)

The owner of **AccessRegistry** can enable or disable any workflow, which results in a reduced number of actions that can be performed by **AccessPoint** contracts. Thus, some scenarios can lead to undesirable consequences for the project and its users, e.g., if the owner's private keys become compromised. We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig. The issue is presented in lines 121 and 127 in **AccessRegistry** contract.

> *Comment from the developers: The owner of the access registry contract will initially be a multi-signature wallet, and eventually, it will transition to on-chain governance. This approach reduces the security risk associated with the compromise of an Externally Owned Account (EOA).*

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. AccessRegistry does not maintain the beacon address consistency (fixed)

In the factory context, the beacon plays a crucial role in fetching the wallet implementation and is thus an important part of the wallet creation process. To deploy a wallet, the `CREATE2` opcode is used. It may result in different addresses for the same owner if the beacon address changes. Consequently, if the beacon is altered, the `getAddress()` will return incorrect addresses for the previously deployed contracts. In order to prevent that, **AccessRegistry** should guarantee the beacon's immutability. Consider marking the beacon address as `immutable` and initialize it within the constructor.

The issue has been fixed and is not present in the latest version of the code.

L02. Optimization is not explicitly turned on

Currently, **foundry.toml** does not contain any information about the optimization parameters like `optimizer` and `optimizer_runs`. Consider specifying optimization parameters explicitly.

L03. NatSpec issue (fixed)

The comment for the `getHash` function at line 82 in **SignaturePaymaster** contract says that `this signature covers all fields of the UserOperation, except the "paymasterAndData"`. However, the `signature` field of the `UserOperation` struct is not used either. Consider mentioning this in the comment.

The issue has been fixed and is not present in the latest version of the code.

L04. No encoding of validation data (commented)

The function `validateUserOp` returns the encoded data that is expected to contain specific information. However, the values for `validUntil` and `validAfter` are not encoded properly, and the `0` value is returned on success in `_validateSignature` function of **AccessPoint** contract.

> *Comment from the developers: Currently, we are not prioritizing the provision of this functionality, as we expect all transactions to be processed through **SignaturePaymaster**. Furthermore, given that **AccessPoint** is an upgradeable contract, we retain the flexibility to incorporate this functionality at a later stage.*

L05. The SignaturePaymaster does not handle compact signatures (commented)

To verify the signature provided by the paymaster, **ECDSA** library is utilized, supporting signatures of both 64 and 65 bytes in length. However, if the shortened version is supplied, the **SignaturePaymaster** wrongly invokes `ECDSA.recover(bytes32, bytes)`, causing `ECDSA.tryRecover(bytes32, bytes)` to revert. Consider using `ECDSA.recover(bytes32, bytes32, bytes32)` when a compact signature is provided at line 126 in `_validatePaymasterUserOp` function of **SignaturePaymaster** contract.

> *Comment from the developers: Unfortunately, the OpenZeppelin library does not provide a function to support both normal and compact signatures due to concerns about malleability. Considering this, we have decided to support only normal signatures for now.*

L06. Unfinished part of the code (fixed)

A comment at line 114 for `getWalletTimestamp` function of **AccessRegistry** contract says that this part of the code should be removed. Consider removing the function that is not needed.

The `getWalletTimestamp` function has been removed from the code.

L07. Unused imports

Throughout the code, there are imports that are not used. Consider removing them to improve code readability.

L08. Unused modifier (fixed)

The modifier `onlyCallerWithAccessPoint` at line 37 in **AccessRegistry** contract is not used. Consider removing it to reduce the size of the final bytecode.

The issue has been fixed and is not present in the latest version of the code.

Notes

N01. No support for ERC721/ERC1155 tokens withdrawals

In the current implementation, the **AccessPoint** contract allows to receive ERC721 and ERC1155 tokens as it inherits from the **TokenCallbackHandler** contract. However, the **WithdrawWorkflow**, which is designed to allow withdrawals from **AccessPoint**, does not allow taking out of these types of tokens.

This analysis was performed by Pessimistic:

Daria Korepanova, Senior Security Engineer
Oleg Bobrov, Security Engineer
Rasul Yunusov, Security Engineer
Konstantin Zherebtsov, Business Development Lead
Irina Vikhareva, Project Manager
Alexander Seleznev, Founder

February 21, 2024