Klaster Security Review

Pashov Audit Group

Conducted by: defsec, Koolex, SpicyMeatball

June 13th 2024 - June 15th 2024

Contents

1. About Pashov Audit Group 2
2. Disclaimer 2
3. Introduction 2
4. About Klaster 2
5. Risk Classification 3
   5.1. Impact 3
   5.2. Likelihood 3
   5.3. Action required for severity levels 4
6. Security Assessment Summary 4
7. Executive Summary 5
8. Findings 7
   8.1. Medium Findings 7
      [M-01] Possible financial loss if the duration of validity is low 7
      [M-02] Multiple ops for the same address case 8
      [M-03] Lack of isSmartContract check 9
   8.2. Low Findings 11
      [L-01] Unfulfilled transaction repayment 11
      [L-02] Dealing with PreVerificationGas 11
      [L-03] Using EntryPoint v6 12
      [L-04] Centralized node batching and potential MEV exploitation 13
      [L-05] Missing payable definition 14
      [L-06] No checks on the size & gas consumption 15
      [L-07] Not compatible with Shanghai hardfork 15
      [L-08] Missing check for non-zero msg.value 16
      [L-09] Lack of validation for upperBoundTimestamp 16

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **klaster-smart-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Klaster

Klaster Protocol contracts support a chain-abstraction protocol, enabling inter-chain transactions (iTx). The primary contracts, KlasterPaymaster and KlasterEcdsaOwnershipModule, handle user operation validation, gas cost management, and the execution of transaction bundles across various blockchain networks.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - a819679b17716bb0b01e787fbb7fc54d7c70127e

fixes review commit hash - ff6a0931d88f30e747345c043e1b1dce53bb0c61

Scope

The following smart contracts were in scope of the audit:

- `KlasterEcdsaOwnershipModule`
- `KlasterPaymaster`
- `DeterministicDeployFactory`
- `BaseAuthorizationModule`
- `Dependencies`

7. Executive Summary

Over the course of the security review, defsec, Koolex, SpicyMeatball engaged with Klaster to review Klaster. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Klaster
Repository	https://github.com/0xPolycode/klaster-smart-contracts
Date	June 13th 2024 - June 15th 2024
Protocol Type	Chain abstraction protocol

Findings Count

Severity	**Amount**
Medium	3
Low	9
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Possible financial loss if the duration of validity is low	Medium	Acknowledged
[M-02]	Multiple ops for the same address case	Medium	Acknowledged
[M-03]	Lack of isSmartContract check	Medium	Resolved
[L-01]	Unfulfilled transaction repayment	Low	Resolved
[L-02]	Dealing with PreVerificationGas	Low	Acknowledged
[L-03]	Using EntryPoint v6	Low	Acknowledged
[L-04]	Centralized node batching and potential MEV exploitation	Low	Acknowledged
[L-05]	Missing payable definition	Low	Resolved
[L-06]	No checks on the size & gas consumption	Low	Acknowledged
[L-07]	Not compatible with Shanghai hardfork	Low	Resolved
[L-08]	Missing check for non-zero msg.value	Low	Resolved
[L-09]	Lack of validation for upperBoundTimestamp	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Possible financial loss if the duration of validity is low

Severity

Impact: Medium

Likelihood: Medium

Description

Simulation might succeed but may fail in the actual on-chain execution. This results in financial losses for the Bundler (the node). Therefore, maintaining consistency between simulation and actual on-chain submission outcomes is very important.

For example, consider a UserOp in the verification stage that only passes if the timestamp value of the block is below 2000. During simulation, conditions like this might be met, but if the timestamp of the transaction when included in a block is higher than 2000, the transaction could fail on-chain. As a result, a loss of funds for the node.

Recommendations

Consider a check for time between simulation and actual execution. Accept UserOps with a minimum duration of validity

Klaster comments

It's handled by the node on the backend side. The node will not accept to execute the iTx bundle if one of the userOps in a bundle has got a pretty tight deadline: deadline too short, or the execution starts in a far future (gas spike risks).

Again, we want to keep the contracts as light as possible, and let the nodes decide on their side what's the level of risk they want to accept with executing iTx bundles - including the risk of financial loss.

[M-02] Multiple ops for the same address case

Severity

Impact: High

Likelihood: Low

Description

A possible failure of transaction done in handleOps method for all UserOperation[], since there is no check for ops array if ops have multiple UserOperation for the same address:

```
function handleOps(UserOperation[] calldata ops) public payable {
        entryPoint.depositTo{value: msg.value}(address(this));
        entryPoint.handleOps(ops, payable(msg.sender));
        entryPoint.withdrawTo(payable(msg.sender), entryPoint.getDepositInfo
          (address(this)).deposit);
    }
```

According to https://www.alchemy.com/blog/account-abstraction, it's very important that:

> As long as the bundle doesn’t include multiple ops for the same wallet, we actually get this for free because of the storage restrictions discussed above: if the validations of two ops don’t touch the same storage, they can’t interfere with each other. To take advantage of this, executors will make sure that a bundle contains at most one op from any given wallet.

two OPs for the same wallet address can cause an issue, A simple example is explained in the next scenario:

- Users send their OPs to the bundler
- ops array has 2 ops for the same wallet address of Bob
- A bundler will start and choose by arranging those OPs according to simulateValidation test
- Assume Bob's account has 1000USDC
- on simulateValidation of Bob's first op, it will check if it can transfer 1000 USDC to another wallet, and it passes successfully through `simulateValidation`.
- the second op of Bob transferring 500 USDC (same wallet's address) goes through simulateValidation and also succeeded as the account holds 1000 USDC.
- the Bundler ready to execute ops array by calling handleOps.
- first op of Bob is executed successfully.
- second op of Bob execution failed since there was not enough USDC.

Recommendations

Add a check in the array to make sure there are no multiple ops for the same wallet address.

Klaster comments

The check is very much needed, but it's handled on the node/backend side of the system - the same way how in 4337 infra it's controlled by the bundler (not the smart contract).

We want to keep smart contracts as light and generic as possible, and do everything that we can off-chain.

[M-03] Lack of `isSmartContract` check

Severity

Impact: Medium

Likelihood: Medium

Description

The `initForSmartAccount` function in the `KlasterEcdsaModule` contract is responsible for initializing the module for a Smart Account and setting the EOA owner. However, it doesn't include a check to ensure that the provided `eoaOwner` address is not a smart contract.

```
function initForSmartAccount(address eoaOwner) external returns (address) {
    if (_smartAccountOwners[msg.sender] != address(0)) {
        revert AlreadyInitedForSmartAccount(msg.sender);
    }
    if (eoaOwner == address(0)) revert ZeroAddressNotAllowedAsOwner();
    _smartAccountOwners[msg.sender] = eoaOwner;
    return address(this);
}
```

While the contract includes a `_isSmartContract` function to check if an address is a smart contract, it is not utilized in the `initForSmartAccount` function. This means that it is possible to set a smart contract as the owner of a Smart Account during initialization.

Although the provided information suggests that this doesn't compromise security due to the requirement of valid EOA signatures for operations, it is still worth considering adding the `isSmartContract` check for consistency and enforcing the intended behavior of only allowing EOA owners.

Recommendations

To improve the clarity and enforceability of the intended behavior, it is recommended to add the `isSmartContract` check in the `initForSmartAccount` function. This ensures that only EOA addresses can be set as owners during the initialization of a Smart Account.

Here's the updated code with the `isSmartContract` check:

```
function initForSmartAccount(address eoaOwner) external returns (address) {
        if (_smartAccountOwners[msg.sender] != address(0)) {
            revert AlreadyInitedForSmartAccount(msg.sender);
        }
        if (eoaOwner == address(0)) revert ZeroAddressNotAllowedAsOwner();
+       if (_isSmartContract(eoaOwner)) revert NotEOA(owner);
        _smartAccountOwners[msg.sender] = eoaOwner;
        return address(this);
    }
```

By adding the `if (_isSmartContract(eoaOwner)) revert NotEOA(eoaOwner);` line, the function will revert if the provided `eoaOwner` address is a smart contract, ensuring that only EOA addresses can be set as owners.

8.2. Low Findings

[L-01] Unfulfilled transaction repayment

If other than the node account tries to use the `KlasterPaymaster.sol`, it will revert because there is no deposited ETH. The paymaster is only prepaid when a Klaster node sends userOps to the entry point:

```
function handleOps(UserOperation[] calldata ops) public payable {
>>      entryPoint.depositTo{value: msg.value}(address(this));
        entryPoint.handleOps(ops, payable(msg.sender));
>>      entryPoint.withdrawTo(payable(msg.sender), entryPoint.getDepositInfo
  (address(this)).deposit);
    }
```

To address this issue paymaster's funds should be verified:

```
function _validatePaymasterUserOp
      (UserOperation calldata userOp, bytes32 userOpHash, uint256 maxCost)
        internal
        virtual
        override
        returns (bytes memory context, uint256 validationData)
    {
+       if (address(this).balance < maxCost)  revert InsufficientBalance();
        (uint256 maxGasLimit, uint256 nodeOperatorPremium) =
            abi.decode(userOp.paymasterAndData[20:], (uint256, uint256));
        return (abi.encode(
          abi.encode

        ), 0
    }
```

[L-02] Dealing with PreVerificationGas

Users set three gas limits when submitting a UserOp: PreVerificationGas, VerificationGasLimit, and CallGasLimit. PreVerificationGas refers to the gas fees consumed during the process of bundling multiple UserOps and sending them to the Entrypoint. Since PreVerificationGas is not pre-calculated, careful consideration is necessary when setting it. Setting it too high might lead to users paying excessive gas fees to the Bundler while setting it too low could cause the Bundler to be unable to cover the gas fees, resulting in transaction reverts and wasted fees.

Check out this analysis link

> In simple english, the PreVerificationGas paid by each userOp is equal to the sum of:
>
> 1. (batch_fixed / batch_size): The share of the fixed portion of the batch overhead.
> 2. batch_variable: The delta in the batch overhead required to add a userOp of its size to the batch.
> 3. per_userop_overhead: The per userOp overhead of running a userOp of its size through the verification and execution loops.

Klaster comments

Noted! And you're right. That's why the initial version is going to have the Node estimating these three gas parameters rather than the user providing those from the outside. That way, the Node can make sure that the UserOp is actually built to make it executable without any errors.

The user still has to agree with the estimate by signing the iTx hash, therefore the frontend part of where the user interacts with the protocol can run the simulations and maker sure the numbers are ok!

[L-03] Using EntryPoint v6

Klaster protocol is using EntryPoint v6, this could have a negative impact on the protocol in the future according to the updated in EntryPoint v7.

- Using EntryPoint v6, EntryPoint can cause redundant calls to occur, complicating the process.

  > Update in v0.7.0: Redundant postOp calls have been eliminated. Now, post-operation logic is more straightforward, with a single call handling necessary after-transaction tasks, simplifying the execution flow and reducing potential errors.

- Current version v6 has proper gas estimation is crucial for transaction execution, ensuring that operations are complete without running out of gas and enhancing security by preventing certain types of attacks.

Update in v0.7.0: New specifications for gas limits, including validatePaymasterUserOp and postOp, provide clearer guidelines for developers. This helps in better gas estimation and strengthens the security framework around transactions.

Consider integrating with EntryPoint v0.7.0.

Klaster comments

Considered switching to v7 before we even started with the development, but as we work on Biconomy smart accounts infrastructure we're limited to what they support on their smart account factory side - and at the moment that's an EntryPoint v6.

We will probably upgrade to v7 once the Biconomy rolls out their new smart account stack.

[L-04] Centralized node batching and potential MEV exploitation

In the current centralized phase of the system, a single node is responsible for batching and processing UserOps from multiple users. While this centralized approach is acceptable for the initial demo phase, it introduces potential risks of Miner Extractable Value (MEV) exploitation, similar to the concerns raised in the previous issue regarding Bundler MEV exploitation. The centralized node, acting as the sole entity responsible for batching and processing UserOps, could potentially reorder or manipulate the transactions within a batch to prioritize its own interests or extract additional value through MEV strategies. This centralized control over transaction ordering and execution could lead to unfair treatment of users' transactions, potential censorship, and erosion of trust in the system.

The centralized node could reorder transactions within a batch to prioritize its own interests, potentially causing users' transactions to be executed at unfavorable rates or even fail.

As the system progresses, introduce a decentralized network of nodes responsible for batching and processing UserOps. This decentralization will distribute the control over transaction ordering and execution, mitigating the risks associated with a single centralized node.

Klaster comments

It's a fair assessment. Klaster Protocol will host its own public node, with the implementation publicly available for everyone to take a look and verify the inner workings of the Node itself.

While the initial version of the protocol is not decentralized in a sense of having a p2p networking implemented between public Klaster nodes, anyone can still choose to run their own private Klaster Node for their own purposes and in that sense have their own private mempool which removes any concerns regarding the MEV extraction.

Launch partners and other integrators will be encouraged to run their own nodes to make the protocol even more resilient in the beginning, until the full decentralization phase is rolled out.

[L-05] Missing payable definition

The `deploy` function in the `DeterministicDeployFactory` contract is responsible for deploying new contracts using the `create2` opcode. However, the current implementation does not include the `payable` modifier, which means that the function cannot receive any native tokens. If the deployed contract requires native tokens to be sent during its construction, the current implementation will fail. The `create2` opcode requires the contract being deployed to be provided with a sufficient amount of native tokens to cover its deployment cost and any additional logic that might require native tokens in the constructor.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract DeterministicDeployFactory {
    event Deploy(address addr);

    function deploy(bytes memory bytecode, uint256 _salt) external {
        address addr;
        assembly {
            addr := create2(0, add(bytecode, 0x20), mload(bytecode), _salt)
            if iszero(extcodesize(addr)) { revert(0, 0) }
        }

        emit Deploy(addr);
    }
}
```

To address this issue, add the `payable` modifier to the `deploy` function, allowing it to receive native tokens from the caller.

[L-06] No checks on the size & gas consumption

The handleOps() function in the contract does not perform any checks on the size of the UserOperation array or estimate the gas consumption required to process all the operations. This oversight can potentially lead to situations where the loop iterating over the operations consumes an excessive amount of gas, causing the transaction to revert and wasting all the gas allocated for the transaction.

Either do an estimation before running all the loops with the length we have of ops, or find a way to split them into multiple batches and prevent the user before the loop from spending all the gas.

Klaster comments

This is handled by the Node as the Node will read maxGasLimit from every UserOp paymasterAndData field, and bound the UserOp to the given max gas limit when executing the batch.

The node will make sure that the simulation of the UserOp fits well inside the given maxGasLimit or otherwise reject the whole iTx bundle .

[L-07] Not compatible with Shanghai hardfork

The compiler for Solidity 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise, deployment of your contracts will fail.

The contracts/deployer/DeterministicDeployFactory.sol uses floating pragma.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract DeterministicDeployFactory {
    event Deploy(address addr);

    function deploy(bytes memory bytecode, uint256 _salt) external {
        address addr;
        assembly {
            addr := create2(0, add(bytecode, 0x20), mload(bytecode), _salt)
            if iszero(extcodesize(addr)) { revert(0, 0) }
        }

        emit Deploy(addr);
    }
}
```

Change the Solidity compiler version to 0.8.19 or define an evm version, which is compatible across all of the intended chains to be supported by the protocol (see https://book.getfoundry.sh/reference/config/solidity-compiler?highlight=evm_vers#evm_version).

[L-08] Missing check for non-zero `msg.value`

The `KlasterPaymaster` contract has three functions that accept Ether: `handleOps`, `simulateHandleOp`, and `simulateValidation`. However, these functions do not check whether the received `msg.value` is greater than zero before depositing the funds to the `entryPoint` contract using `entryPoint.depositTo{value: msg.value}(address(this))`.

If the `msg.value` is zero, the deposit operation will still be executed, but it will not have any effect on the contract's balance.

To ensure that the `msg.value` is greater than zero before depositing funds, add a check at the beginning of each affected function. If the `msg.value` is zero, the function should revert with an appropriate error message.

[L-09] Lack of validation for `upperBoundTimestamp`

In the `validateUserOp` function of the `KlasterEcdsaModule` contract, there is a lack of proper validation for the `upperBoundTimestamp` parameter. The function directly uses the `upperBoundTimestamp` value received from the decoded signature without performing any checks or validations.

```
function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash)
        external
        view
        virtual
        returns (uint256)
    {
        (bytes memory sigBytes,) = abi.decode(userOp.signature,
          (bytes, address));

        (
            bytes32 iTxHash,
            bytes32[] memory proof,
            uint48 lowerBoundTimestamp,
            uint48 upperBoundTimestamp,
            bytes memory userEcdsaSignature
        ) = abi.decode(sigBytes, (bytes32, bytes32[], uint48, uint48, bytes));

        bytes32 calculatedUserOpHash = getUserOpHash
          (userOp, lowerBoundTimestamp, upperBoundTimestamp);
        if (!_validateUserOpHash(calculatedUserOpHash, iTxHash, proof)) {
            return SIG_VALIDATION_FAILED;
        }

        if (!_verifySignature(iTxHash, userEcdsaSignature, userOp.sender)) {
            return SIG_VALIDATION_FAILED;
        }

        return _packValidationData
          (false, upperBoundTimestamp, lowerBoundTimestamp);
    }
```

If upperBoundTimestamp is equal to zero, it is recommended to set uint48 max.

Klaster comments

This is the way ERC-4337 EntryPoint works - validation modules simply report the lower/upper bound timestamps (sig validity period) back to whoever is asking this information. This information requester (EntryPoint) then chooses what to do with the received info.

In our case, ERC4337 entrypoint will receive back the timestamps and reject the UserOp for which the timestamp is out of bounds: link

It will also set the upper bound to UINT48_MAX if the upper bound is 0: link