certora

Security Assessment & Formal Verification Report

KINTO

March 2024

Prepared for

Table of content

Project Summary

Project Scope

The original project files are:

Repository	Files	Commits	Compiler	Platform
Kinto-core	1. KintoID.sol 2. KintoWallet.sol 3. KintoWalletFactory.sol 4. SponsorPaymaster.sol 5. KYCViewer.sol 6. KintoAppRegistry.sol (added later)	f53d802 (original) d801575 (fix)	Solidity 0.8.18	Arbitrum Nitro
Account Abstraction	7. EntryPoint.sol	61f6779	Solidity 0.8.18	Arbitrum Nitro
Geth	8. state_transition.go 9. error.go 10. instructions.go 11. kinto.go (added later) 12. kinto_forks.go (added later)	c407157 (original) 31b18ba (gist fix #1) 06701da (gist fix #2) fe443d4 (gist fix #3)		

Remark: Kinto's modification to the Geth code is not considered in scope for this engagement. However, it is included here since it turned out to have implications for parts that were in scope and ultimately proved paramount in fixing the bugs that were discovered in the original commit.

Protocol Overview

Kinto is an L2 optimistic rollup which provides KYC, insurance, and AML & fraud monitoring at the blockchain level. Furthermore, Kinto supports native account abstraction at the chain level, which means that applications can offer easy onboarding and account creation through username/password or mobile 2FA, without the need for complicated seed phrases or private keys.

Apart from the Nitro infrastructure, the protocol architecture is based on the following components:

1. The Kinto ID NFT: this smart contract controls the identity tokens minted to individual users and companies. Corresponds to the solidity file KintoID.sol in the code base.
2. The Kinto EntryPoint: a singleton Smart Contract which is a modification of the standard ERC-4337 EntryPoint. Corresponds to the solidity file EntryPoint.sol in the code base.
3. The Kinto Wallet Factory: a singleton which creates contracts and Kinto wallets for users. Corresponds to the Solidity file KintoWalletFactory.sol in the code base.
4. The Paymaster: a smart contract which sponsors ERC-4337 userOp transactions.
5. Kinto Wallets: Kinto disallows EOA. Instead, users interact with the system via a non-custodial, ERC-4337 & KYC compatible wallet called the Kinto Wallet which must be created via the factory. Corresponds to the Solidity file KintoWallet.sol in the code base.
6. Identity Nodes: interact with the Kinto ID smart contract on the one side and the KYC Providers on the other side in order to give KYC tokens to users which have completed the KYC process.
7. KYC Providers: external identity provider which allows Kinto to access off-chain KYC information from the blockchain. They interface with Kinto's identity nodes and do not have access to user's on-chain information (such as account address, balances etc).

Project Goals

The six key properties of Kinto are:

- **KYC and compliance** - Kinto has a modified execution layer that reverts transactions not originating from KYC'ed addresses, so transactions can only be performed by verified participants. In addition, the network runs AML on participants continuously and if a user gets added to a sanction list, his Kinto ID NFT will automatically be updated with this information.
- **Privacy** - Personal data is only stored in the KYC provider chosen by the users. Kinto itself doesn't store any user data - in particular, the Kinto NFT minted for the user only has flags identifying whether a user has KYC'd, accreditation, and whether or not there are AML violations in different jurisdictions. Conversely, the KYC Providers are not supplied with the user's account addresses so it should be impossible to link a user's PII (Personal Identifiable Information) and its Kinto address - even if the KYC Provider has been hacked.
- **Non-custodial.** Users retain ownership over their digital assets and data (note however that there is a way for users to recover their accounts via a week-long recovery process). The protocol and applications built on top must request access to personal data and, with the exception of emergency cases, only the owner of the data can grant access.
- **Native Account Abstraction** - Users must create their Kinto Wallet which is a Smart Contract Wallet compatible with a variation of ERC-4337. Transactions initiated by EOAs are disabled and must be sent via Account Abstraction and its Entry Point. EOA without a Kinto Wallet can only view the state of the blockchain (via RPC calls), not change it.
- **DeFi Access** - Kinto allows users to interact with Smart Contracts on mainnet, giving access to financial institutions and users to the DeFi world.
- **Blockchain Security** - Kinto is based on Arbitrum Nitro technology and settles on Ethereum Mainnet. In addition, having KYC at the chain level makes the protocol highly sybil resistant.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	3	3	3
High	4	4	4
Medium	7	5	5
Low	5	5	5
Informational	5	5	4
Total	**24**	**22**	**21**

Detailed Findings

ID	Title	Bug Category	Severity
C-01	An attacker can use the staking functionality in the EntryPoint to gain a backdoor access to Kinto, bypassing KYC and sanctions	Logic	Critical
C-02	An attacker can use the beneficiary field in the EntryPoint to gain a backdoor access to Kinto, bypassing KYC and sanctions	Logic	Critical
C-03	An attacker can use the aggregator functionality in the EntryPoint to gain a backdoor access to Kinto, bypassing KYC and sanctions	Logic	Critical
H-01	Any owner of a token can call burn()	Access Control	High
H-02	There are no restrictions on deposits	Lack Of Restrictions	High
H-03	The withdrawTo() method does not update the Paymaster balance after withdrawal	Incorrect Calculation	High
H-04	Self-destruct allows a user to	Logic, Arbitrum L2	High

	send money to a non-KYC address.		
M-01	Wrong Balance check in Paymaster	Incorrect Calculation	Medium
M-02	Direct Token Approvals are not fully blocked	Logic	Medium
M-03	approveTokens and revokeTokens can silently fail	Unchecked Return Value	Medium
M-04	The value of the constant MAX_COST_OF_USEROP is wrong	Arithmetic Error	Medium
M-05	The KintoWallet method _validateSignature accesses storage slots which are not associated with the address of the wallet	ERC-437 compatibility	Medium
M-06	SetWalletFactory has no access control and can be front-run	Access Control	Medium
M-07	Compliance can be contradictory	Architecture, Logic, Legal	Medium
L-01	Off-by-one error in threshold conditions	Arithmetic Error	Low

L-02	Wallets can be created without a valid recoverer	Input Validation	Low
L-03	monitor() execution could revert due to a bad input	Input Validation	Low
L-04	_domainSeparator() is not cached	Gas	Low
L-05	Initialization for SponsorPaymaster can be front-run	Access Control	Low
I-01	onlySignerVerified() has an _id input but is never used.	Best Practice	Informational
I-02	salt should always be bytes32	Best Practice	Informational
I-03	Any minting operation increases nextTokenID by two (and not one) overall.	Best Practice	Informational
I-04	The Entry Point queries the wallet's timestamp and not its version	Best Practice	Informational
I-05	The KintoWallet method _validateSignature returns SIG_FAULT instead of reverting when the owner is not KYC'ed	ERC-437 compatibility	Informational

C-01 An attacker can use the staking functionality in the EntryPoint to gain a backdoor access to Kinto, bypassing KYC and sanctions

Severity: **Critical**	Impact: **Critical**	Likelihood: **Very high**
Category: Logic		Files: StakeManager.sol

Description

In Kinto, only users who have Wallets and passed KYC and AML should be able to send transactions. However, using two methods (withdrawStake and withdrawTo) related to the Stake Manager functionality which is inherited by Kinto's Entry Point, an attacker can prepare a backdoor contract in advance which would allow it to continue interacting directly with the network, even under KYC and sanctions.

Exploit Scenario

- Setup:
  - The attacker deposits (or stakes) Ether in the Kinto Entry Point.
  - The attacker deploys the backdoor contract via the Kinto Wallet Factory.
- When the attacker is caught in identity fraud or placed under sanctions, the attacker regains access by activating the backdoor:
  - Withdraws a portion of its deposit or stake (via the Stake Manager functionality inherited by the Entry Point) and directs the reimbursement to go to the backdoor contract.
  - The methods in question make a raw call (no data, just value) to the backdoor contract[1] which triggers the receive() or fallback() function.
  - The amount of gas left and the msg.value are used as a way for the EOA attacker to communicate with the backdoor contract.

[1] Note that a Solidity call forwards *all* the remaining gas and not just 2300 like send/transfer.

    - Thus, the attacker is free to perform essentially arbitrary message calls on the Kinto network bypassing all the restrictions imposed by the KYC, AML, and Account Abstraction functionalities.
- A PoC of the exploit was privately delivered to the customer.

Recommendation

Block these two methods.

Customer response

Acknowledged. We will fix this by blocking these two methods at the Geth level.

Fix review

The fixes 31b18ba and the subsequent 06701da,fe443d4 solve the problem by blocking access to these methods.

C-02 An attacker can use the beneficiary field in the EntryPoint to gain a backdoor access to Kinto, bypassing KYC and sanctions		
Severity: **Critical**	Impact: **Critical**	Likelihood: **Very high**
Category: Logic		Files: EntryPoint.sol

Description

A Bundler[2] gets essentially "free action" in the end of the methods handleOps and handleAggregatedOps which allows it to call an arbitrary beneficiary (via the internal method _compensate) bypassing KYC and AML restrictions.

Exploit Scenario

- Setup:
  - The attacker deposits (or stakes) Ether in the Kinto Entry Point.
  - The attacker deploys the backdoor contract via the Kinto Wallet Factory.
- When the attacker is caught in identity fraud or placed under sanctions, the attacker regains access by activating the backdoor:
  - The attackers acts as (or collaborates with) a Bundler and submits a bundle of UserOps to handleAggregatedOps while setting the beneficiary address to the backdoor contract.

Recommendation

Require beneficiary to be a KYC'ed address.

Customer response

Acknowledged. We will fix this by requiring the beneficiary to equal msg.sender at the Geth level.

[2] Note: this is not a real restriction. Any EOA can be a bundler in Kinto, and bundlers do not have to be KYC'ed.

Fix review

The fix 31b18ba solves the problem successfully for handleOps&handleAggregatedOps. The subsequent fix 06701da,fe443d4 blocked access to handleAggregatedOps completely (see H-03) while still requiring that beneficiary == msg.sender for handleOps.

C-03 An attacker can use the aggregator functionality in the EntryPoint to gain a backdoor access to Kinto, bypassing KYC and sanctions

Severity: **Critical**	Impact: **Critical**	Likelihood: **Very high**
Category: Logic		Files: EntryPoint.sol

Description

In the method handleAggregatedOps, an attacker can set a target contract as the aggregator and submit an empty array of userOps per that aggregator, thus, by exploiting the call to validateSignatures, to gain a backdoor which would allow a non-KYC user to bypass checks.

Exploit Scenario

- Setup:
  - The attacker deposits (or stakes) Ether in the Kinto Entry Point.
  - The attacker deploys the backdoor contract via the Kinto Wallet Factory.
- When the attacker is caught in identity fraud or placed under sanctions, the attacker regains access by activating the backdoor:
  - The attackers acts as (or collaborates with) a Bundler and submits an empty bundle of UserOps to handleAggregatedOps while setting the aggregator address to the backdoor contract.

Recommendation

In the short-term, block access to the method handleAggregatedOps (since currently there is no aggregator in Kinto). In the longer term, ensure that only calls to whitelisted aggregators are allowed.

Customer response

Fixed by blocking access to handleAggregatedOps on the Geth level.

Fix review

The fix 06701da solves the problem successfully by completely blocking access to the EntryPoint's handleAggregatedOps.

H-01 Any owner of a token can call burn().

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Category: Access Control		Files: KintoID.sol

Description

Any owner of a token (including one which is not a KYC provider) can burn their identity token by calling burn(), inherited from OpenZeppelin's ERC721BurnableUpgradeable.sol, allowing them to bypass the safety precautions placed on burnKYC().
Property violated: onlyKYCCanChangeBalance, monitorCannotRevertByNonProvider

Exploit Scenario

An owner, either accidentally or as a result of being tricked by a malicious 3rd party, burns its own KYC token. Since Kinto, as policy, does not re-mint tokens to an address which have been burnt, the user loses all access to its wallet and its assets with the possibility of recovery.

Recommendation

Fix this by only enabling burn via burnKYC().

Customer response

Already fixed at the smart contract level.

H-02 There are no restrictions on deposits

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Category: Lack Of Restrictions		Files: SponsorPaymaster.sol BasePaymaster.sol

Description

Currently anyone (who is also not KYC'ed) can deposit ETH into the SponsorPaymaster, for himself or for someone else who is not KYC'ed or is currently under sanctions. Even though the target user can not use or *spend* the deposited amount, it is not clear whether or not this constitutes a breach of AML law compliance in certain jurisdictions, thereby undermining the logic of the system.

Recommendation

Research this point thoroughly and incorporate it into the system design. If true, it is likely to require some non-trivial modifications.

Customer response

We are going to fix this by blocking deposit at the Geth level and checking KYC in addDepositFor.

Fix review #1

The Geth fix 31b18ba to prohibit direct access to the BasePaymaster method deposit() and only enable access transactions which target the addDepositFor method (which updates balances and adds some KYC checks).

However this is not sufficient to solve the problem completely since there are alternative paths which activate (essentially the same) exploit:

- deposit() is simply a wrapper of the entryPoint's depositTo(...) method, which is not blocked at the Geth level.
- In addition, the EntryPoint's fallback method activates depositTo(...) as well.

Customer response

Acknowledged. The additional methods will be blocked in a subsequent fix.

Fix review #2

The fix 06701da (and fe443d4) solves the problem successfully by blocking access to the EntryPoint's fallback() and depositTo(...).

H-03 The withdrawTo() method does not update the Paymaster balance after withdrawal		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Category: Logic		Files: SponsorPaymaster.sol BasePaymaster.sol

Description

The Sponsor Paymaster contract inherits the method withdrawTo() from Base Paymaster contract:

```
Unset
  /**
   * Withdraw value from the deposit.
   * @param withdrawAddress - Target to send to.
   * @param amount - Amount to withdraw.
   */
  function withdrawTo(
    address payable withdrawAddress,
    uint256 amount
  ) public onlyOwner {
    entryPoint.withdrawTo(withdrawAddress, amount);
  }
```

which does not update the Paymaster balances, thus creating insolvency in the deposited ETH in the Entrypoint.

Furthermore, as a consequence, executing the withdrawTokensTo() method (in the SponsorPaymaster contract):

```
Unset
  /**
  * Withdraw ETH
  * can only be called after unlock() is called in a previous block.
  * @param target address to send to
  * @param amount amount to withdraw
  */
 function withdrawTokensTo(address target, uint256 amount) external override nonReentrant() {
   require(
     balances[msg.sender] >= amount &&
     unlockBlock[msg.sender] != 0 && block.number > unlockBlock[msg.sender],
     'SP: must unlockTokenDeposit'
   );
   balances[msg.sender] -= amount;
   entryPoint.withdrawTo(payable(target), amount);
 }
```

could lead to a DoS, because the deposit in the entry point would not match the actual balances of the account.

Property violated: PaymasterEthSolvency, cannotDos_withdrawTokensTo

Recommendation

Consider fixing the code to correctly track the balance.

Customer response

We are going to fix it by blocking withdrawTo at the Geth level.

Fix review

The fixes 31b18ba and the subsequent 06701da,fe443d4 solve the problem by blocking access to withdrawTo() on the Paymaster.

H-04 Self-destruct allows a user to send money to a non-KYC address.		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Category: Logic, Arbitrum L2		Files:

Description

The SELFDESTRUCT opcode allows the contract to specify a beneficiary address to which it sends its current Ether balance. Since the address can be essentially[3] arbitrary, an attacker can use this trick to send funds to a non-KYC user bypassing Kinto's sanction mechanism.

Customer Response

Acknowledged. Will fix.

Fix Review

The change to opSelfdestruct in commit fe443d4 solves the problem by specifying a new special Kinto address (the "SelfDestructWallet") onto which all Ether send, essentially ignoring the beneficiary stack variable.

[3] One small exception: in Arbitrum, if the address of the contract being destroyed is equal to the beneficiary address, then the balance is burned and no Ether is sent.

M-01 Wrong Balance check in Paymaster		
Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Category: Incorrect Calculation		Files: [SponsorPaymaster.sol]()

Description

In Line 209 of SponsorPaymaster.sol,

```
Unset
require(balances[targetAccount] >= maxCost, 'SP: deposit too low');
```

maxCost should have been ethMaxCost.

Recommendation

Rectify the requirement with the correct value.

Customer response

Already fixed.

M-02 Direct Token Approvals are not fully blocked		
Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Category: Logic		Files: KintoWallet.sol

Description

The internal function _preventDirectApproval() given in lines 371-378 of KintoWallet:

```
Unset
  function _preventDirectApproval(bytes calldata _bytes) pure internal {
    // Prevent direct deployment of KintoWallet contracts
    bytes4 approvalBytes = bytes4(keccak256(bytes("approve(address,uint256)")));
    require(
      bytes4(_bytes[:4]) != approvalBytes,
      'KW: Direct ERC20 approval not allowed'
    );
  }
```

is meant to block the possibility of direct token approvals which do not go via the intended approveTokens method:

```
Unset
  /**
   * @dev Approve tokens to a specific app
   * @param app app address
   * @param tokens tokens array
   * @param amount amount array
   */
  function approveTokens(
    address app,
    address[] calldata tokens,
    uint256[] calldata amount)
    external override onlySelf
```

```
{
  require(tokens.length == amount.length, 'KW-at: invalid array');
  require(appWhitelist[app], 'KW-at: app not whitelisted');
  for (uint i = 0; i < tokens.length; i++) {
    if (_tokenApprovals[app][tokens[i]] > 0) {
      IERC20(tokens[i]).approve(app, 0);
    }
    _tokenApprovals[app][tokens[i]] = amount[i];
    IERC20(tokens[i]).approve(app, amount[i]);
  }
}
```

Presumably to prevent a user from granting approval to a non-KYC EOA to trade on its behalf. However, this can be bypassed in multiple ways:

- First, the direct approval could be bypassed by a wrapper function with a different method signature that performs the same action exactly.
- Second, the direct approval only applies to ERC20 tokens but does not deal with other common standards like: ERC721 (non-fungible tokens), ERC777, ERC1155,... which have their own method signatures for approval.
- Third, even for ERC20 the method does not account for one of the most common implementations - OpenZeppelin's SafeERC20 which includes the method signatures of safeApprove, increaseAllowance, and decreaseAllowance.

Recommendation

We suggest reworking the mechanism of token approvals in Kinto Wallet to account for these problems.

Customer response

Decided to remove all this logic. App level checks are enough.

M-03 approveTokens and revokeTokens can silently fail

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Category: Unchecked Return Value		Files: KintoID.sol

Description

The methods approveTokens() and revokeTokens() update the internal accounting of token approvals without ever checking whether the subsequent external call to the ERC20 approve method has been successful or not.

Exploit Scenario

An ERC20 token is paused. The user calls revokeTokens to remove the approval it gave an app beforehand to access its tokens. The external call fails without any indication to the user and the internal _tokenApprovals mapping is incorrectly updated.

Recommendation

Add a check for the return value from the call to approve().

Customer response

Removed all this logic from the wallet.

M-04 The value of the constant MAX_COST_OF_USEROP is wrong

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Category: Unchecked Return Value		Files: SponsorPaymaster.sol

Description

In L.33 of SponsorPaymaster, there is a mismatch between the code and the comment:

```
Unset
uint256 constant public MAX_COST_OF_USEROP = 3e15; // 0.03 ETH
```

Assuming the comment is correct, this should have been 3e16 (==0.03 ETH), which means that the code disallows for userOps which cost merely 10% of the maximal intended amount.

Recommendation

Fix either the value or the comment so that both match each other.

Customer response

Fixed.

M-05 The KintoWallet method _validateSignature accesses storage slots which are not associated with the address of the wallet.

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Category: ERC-437 compatibility		Files: KintoWallet.sol

Description

The internal method _validateSignature access storage slots which are associated with the primary owner and not the wallet itself in L.305-307:

```
Unset
    if (!kintoID.isKYC(owners[0])) {
      return SIG_VALIDATION_FAILED;
    }
```

This is in contradiction to ERC-4337 which states that while simulating `userOp` validation, the client (i.e., the bundler) should check that the `userOp` storage access is limited as follows:

i. self storage (of factory/paymaster, respectively) is allowed, but only if self entity is staked
ii. account storage access is allowed (see below),
iii. in any case, may not use storage used by another UserOp `sender` in the same bundle (that is, paymaster and factory are not allowed as senders) “

Where we define storage slots as "associated with an address" as all the slots that are uniquely related to this address, and cannot be related with any other address. In solidity, this includes all storage of the contract itself, and any storage of other contracts that use this contract address as a mapping key.

Thus, an address `A` is associated with:

1. Slots of contract `A` address itself.
2. Slot `A` on any other address.

3. Slots of type `keccak256(A || X) + n` on any other address. (to cover `mapping(address => value)`, which is usually used for balance in ERC-20 tokens). `n` is an offset value up to 128, to allow accessing fields in the format `mapping(address => struct)`

Remark: Note that in Kinto's economic model the bundler is meant to be an *external entity* (possibly not even KYC'ed!), and that a single primary owner can possess multiple Kinto wallets. Thus, not only is this a contradiction to the ERC (which means that standard bundler implementation might be harder to adapt to this setting) but also that we are violating the e*conomical reasoning that originally underlies this restriction*, since the identity nodes and the KYC Provider have undue power over the bundler.

Customer response

Acknowledged.

M-06 SetWalletFactory has no access control and can be front-run		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Category: Access Control		Files:

Description

The custom method SetWalletFactory in Kinto's EntryPoint can be front-run which would allow an attacker to set its own wallet factory.
Note: the severity of this is higher then L-05 since the Entry Point is a non-upgradable singleton, and since deployment of a new Entry Point would necessitate changes to both the core smart contracts and the Geth code.

Recommendation

Awareness and a well-planned deployment procedure (including tests) is possibly the best mitigation to such front-running attacks depending on the method you choose to deploy your smart contracts. In general, we advise the use of a vetted deployment script, a carefully planned procedure, and/or a good off-chain monitoring system.

Customer response

Contract is already deployed so not an issue.

M-07 Compliance can be contradictory.		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: Architecture, Logic, Legal		Files:

Description

Currently, most system interaction checks whether users are KYC'ed *everywhere* or not - by design. Implicitly, embedded in the current design of Kinto's smart contracts seems to be the assumption that taking the 'maximalist' approach is the best route. However, this is not automatically true because confirming to certain sanctions in one domain could be considered as non-compliant in other domains - for example, some countries could view confirming to sanctions imposed by a foreign power as grounds for legal action (e.g., participation of american citizen in a boycott imposed by a foreign power might be considered a federal offense). Thus, the design of the KYC system might need to be more fine-grained to fit its legal purpose while maintaining its global nature.

Customer Response

I don't believe this warrants a high qualification. The theoretical collusion of countries is true in theory, but in practice, most of the Western world agrees to follow OFAC, and we are going to default to the same path. This is not a concern in the short term, and it is definitely not high on our list.

L-01 Off-by-one error in threshold conditions		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Category: Arithmetic Error		Files: [SponsorPaymaster.sol]()

Description

Threshold conditions are sometimes bound-inclusive and sometimes aren't (compare RATE_LIMIT_THRESHOLD_SINGLE vs. GAS_LIMIT_THRESHOLD_SINGLE).

Recommendation

Fix the inequalities to keep a uniform standard throughout the code base.

Customer response

Fixed, all inclusive now.

L-02 Wallets can be created without a valid recoverer

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Category: Input Validation		Files: KintoWallet.sol KintoWalletFactory.sol

Description

The Kinto Wallet Factory (as well as the Kinto Wallet initialization function) does not check that the recoverer is not the zero address. Thereby allowing the user to create Wallets that cannot be recovered.

Property violated: ZeroAddressIsNeitherWalletOwnerNorRecoverer

Recommendation

Add the missing check.

Customer response

Fixed.

L-03 monitor() execution could revert due to a bad input

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Category: Input Validation		Files: [KintoID.sol]()

Description

If a mistake is made in the input to the monitor() function, in which an invalid account is a part of the list (zero balance), then the whole monitor() execution would revert leading to a waste of gas.

Property violated: monitorCannotRevertByNonProvider

Customer response

Fixed.

L-04 _domainSeparator() is not cached

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Category: Gas		Files: KintoID.sol

Description

In Lines 416-420 of KintoID the method getEIP712message always recomputes the domain separator:

```
Unset
  function _getEIP712Message(SignatureData memory signatureData) internal view returns (bytes32)
{
    bytes32 domainSeparator = _domainSeparator();
    bytes32 structHash = _hashSignatureData(signatureData);
    return keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
  }
```

Which seldom changes (only if the chain ID changes!) and includes many gas-intensive operations (e.g., three invocations of keccak256 costing over 100 gas).

Recommendation

Consider pre-computing an initial _domainSeparator() and saving it as an immutable. Check chainID changes, and only in that case recompute and cache the result to a global variable to reduce gas consumption. Alternatively, since KintoID is upgradable, consider working only with an immutable and upgrading the contract in case that chain ID changes.

Customer response

Will fix.

Fix Review

The fix solves the problem mentioned.

Remark: one small note - it is possible in rare cases that the chain will need to perform a hard-fork, in which case the value of `chainId` and consequently `domainSeparator` ought to be recomputed. Thus it is considered best practice to account for that in code.

L-05 Initialization for SponsorPaymaster can be front-run		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Category: Input Validation		Files:

Description

The initialize method for the SponsorPaymaster can be front-run, allowing an attacker to take over the contract.

Recommendation

As in M-06, this ultimately depends on the way Kinto chooses to deploy its smart contracts. On-chain input validation is possible, but also the use of a vetted deployment script or an off-chain monitoring system is a good solution.

Customer response

Contract is already deployed so not an issue (it didn't happen).

I-01 onlySignerVerified() has an *_id* input but is never used.

Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: KintoID.sol

Description

The definition of the modifier onlySignerVerified (Lines 392-395 of KintoID.sol) begins with:

```
Unset
  /**
   * @dev Check that the signature is valid and the sender is a valid KYC provider.
   * @param _id id of the token to be signed.
   * @param _signature signature to be recovered.
   */
  modifier onlySignerVerified(
   uint256 _id,
   IKintoID.SignatureData calldata _signature
  )
```

But the uint256 _id is never used in the code itself.

Recommendation

Consider removing _id from the interface.

Customer response

Fixed.

I-02 salt should always be bytes32		
Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: [KintoWalletFactory.sol]()

Description

Some functions get salt as uint256 but then it is being converted to bytes32.

Recommendation

Consider changing the interface so salt is bytes32 everywhere.

Customer response

Ack. Fixed.

I-03 Any minting operation increases nextTokenID by 2 overall.

Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: KintoID.sol

Description

The variable _nextTokenId is accidentally advanced twice - both in the external wrapper methods mintIndividualKyc and mintCompanyKyc and in the internal function _mintTo which is invoked by them.

Recommendation

Fix this.

Customer response

Fixed.

I-04 The Entry Point queries the wallet's timestamp and not its version

Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: EntryPoint.sol

Description

Currently, the Kinto Entry Point queries the sender wallet's timestamp instead of its version to test validity.

```
Unset
    try IWalletFactory(walletFactory).getWalletTimestamp(mUserOp.sender) returns (uint256
version) {
        if (version == 0) {
          revert FailedOp(opIndex, "AA35 invalid wallet");
        }
    } catch {
        revert FailedOp(opIndex, "AA35 invalid wallet factory");
    }
```

This would be problematic in the scenario that some wallet version's would become deprecated in the future since the EntryPoint would still allow transactions for all versions.

Recommendation

Consider querying the wallet version to allow more flexibility in the future.

Customer response

Acknowledged. Won't fix. Can't upgrade entry point.

I-05 The KintoWallet method _validateSignature returns SIG_FAULT instead of reverting when the owner is not KYC'ed.		
Severity: **Informational**	Impact:	Likelihood:
Category: ERC-437 compatibility		Files: KintoWallet.sol

Description

The method _validateSignature of a Kinto Wallet returns SIG_FAULT when the primary owner is not KYC'ed or under sanctions:

```
Unset
    if (!kintoID.isKYC(owners[0])) {
      return SIG_VALIDATION_FAILED;
    }
```

This is not in accordance with ERC-4337 which states:
" If the account does not support signature aggregation, it MUST validate the signature is a valid signature of the `userOpHash`, and SHOULD return SIG_VALIDATION_FAILED (and not revert) on signature mismatch. Any other error should revert. "

Customer response

Acknowledged.

Formal Verification

Verification Overview

General assumptions

- We didn’t take the upgrade mechanism into account. All rules were verified against a static proxy contract which only calls the implementation contracts listed below.
- Any loop was unrolled at most 3 times (iterations).

The following contracts were formally verified by the properties which are listed below per library\contract:

A. src/KintoID.sol
B. src/wallet/KintoWallet.sol
C. src/wallet/KintoWalletFactory.sol
D. src/paymasters/SponsorPaymaster.sol

Verification Notations

Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Violated	A counter example exists that violates one of the assertions of the rule.

Properties

KintoID.sol

Assumptions

- The total number of sanctions per user cannot exceed 200. [4]

Rule Name	Description
Verified **lastMonitoredAtInThePast**	lastMonitoredAt() is never in the future
Verified **AdminRoleIsDefaultRole**	The role admin of any role is the DEFAULT_ADMIN_ROLE()
Verified **onlyRoleAdminRevokesRole**	Only the DEFAULT_ADMIN_ROLE() can revoke/grant a role from/to an account.
Verified **TokenBalanceIsZeroOrOne**	The ERC721 token balance of any user is either zero or one.
Verified **IsOwnedInTokensArray**	If a token has a (non-zero) owner, then the total supply is greater than zero.
Verified **TokenIndexIsUpToArrayLength**	The token index of any tokenID must be less than the number of minted tokens.
Verified **TokenAtIndexConsistency**	The token index points to the same token in the tokens array.
Verified **NoOwnerNoIndex**	If a tokenID has no owner, then its index is zero.

[4] If any account has 255 (max uint8) sanctions, any additional sanction could not be added and such operation would revert due to overflow.

Verified **ZeroAddressNotKYC**	The zero address is not KYCd.
Verified **BalanceOfZero**	The ERC721 token balance of the zero address is zero.
Verified **isKYC_cannotRevert**	isKYC() cannot revert.
Verified **isSanctionsSafeIn_cannotRevert**	isSanctionsSafeIn() cannot revert.
Verified **isSanctionsSafe_cannotRevert**	isSanctionsSafe() cannot revert.
Verified **isCompany_cannotRevert**	isCompany() cannot revert.
Verified **isIndividual_cannotRevert**	isIndividual cannot revert.
Verified **ownerCanChangeOnlyFromZeroAndBack**	The owner of a token could only be transferred to, or from the zero address.
Verified **mintOnlyNextID**	Only the _nextTokenID+1 is minted, and only by the mintCompanyKyc() or mintIndividualKyc() functions.
Verified **mintToOwnerOnly**	The new owner of the nextTokenID is the only one who is being minted a token.
Violated **onlyKYCCanChangeBalance**	Only a KYC provider can change any account ERC721 balance.

Issue: H-01 Any owner of a token can call burn().	Violation reason: Any owner of a Kinto NFT could burn his token. **Rule is verified after the latest commit fix.**
Verified **cannotBurnRightAfterMint**	It's impossible, by anyone, to burn a KYC token right after it's being minted.
Verified **hasSanctionCountIsNonZero**	If an account is sanctioned anywhere, then its sanctions count is non-zero.
Verified **onlySanctionMethodCanSanction**	Only the addSanction(), removeSanction(), and monitor() functions can change the sanction status (ignoring last monitored time).
Verified **addSanctionIntegrity**	The addSanction() function: (a) Must turn on the sanction status for the correct countryID and account. (b) Must increase the sanction count for that account by 1.
Verified **The removeSanction() function**	The removeSanction() function: (a) Must turn off the sanction status for the correct countryID and account. (b) Must decrease the sanction count for that account by 1.
Verified **addSanctionIdempotent**	The addSanction() function has no effect if the account is already sanctioned in the same country.
Verified **removeSanctionIdempotent**	The removeSanction() function has no effect if the account is not sanctioned in the same country.

Verified **addSanctionCommutativity()**	addSanction() is commutative with respect to the account and country ID.
Verified **removeSanctionCommutativity()**	removeSanction() is commutative with respect to the account and country ID.
Verified **addedSanctionCanBeRemoved**	Any sanction that was added could be removed later (by any KYC provider).
Verified **removedSanctionCanBeAdded**	Any sanction that was removed could be added later (by any KYC provider).
Verified **addingOrRemovingSanctionsAreIndependent**	addSanction() or removeSanction() are account and countryID independent.
Verified **addedTraitCanBeRemoved**	Any trait that was added could be removed later (by any KYC provider).
Verified **removedTraitCanBeAdded**	Any trait that was removed could be added later (by any KYC provider).
Verified **noncesIncreaseCorrectly**	Integrity of nonce transition: (a) Nonces cannot decrease and can increase by 1 at most. (b) A nonce could only change for one signer at a time.
Violated **monitorCannotRevertByNonProvider** **Issue: H-01 Any owner of a token can call burn().** **Issue: L-03 monitor() execution could revert due to a bad input**	An account which is not a KYC provider cannot make monitor() revert. Violation reason: Any owner of a Kinto NFT could burn his token. **Rule is verified after the latest commit fix.**

Verified **monitorSanctionsCommutative**	The monitor() function is commutative with respect to update data, if the IDs are different (for a single account).
Verified **monitorSanctionsAssociative**	The monitor() function is associative with respect to the updated data (for a single account).
Verified **monitorAccountsAssociative**	The monitor() function is associative with respect to the accounts list
Verified **monitorAccountsCommutative**	The monitor() function is commutative with respect to the accounts list (for a single type of data).
Verified **monitorSanctionsCannotFrontRun**	The monitor() function cannot front-run and cause a subsequent call to monitor() revert, for a single account.
Verified **monitorAccountsCannotFrontRun**	The monitor() function cannot front-run and cause a subsequent call to monitor() revert (for independent accounts).
Verified **monitorEmptyDataSucceeds**	A KYC provider can always call monitor() with empty data.

KintoWallet.sol

Assumptions

- We assume the wallet contract is always in the post-initialization state, which represents the proxy, deployed from the wallet factory.
- We assume that before a call to finishRecovery() the new owner (owners[0]) was already minted KYC.

Verified **AllowedSignerPolicy**	The allowed signer policies are either SINGLE_SIGNER(), MINUS_ONE_SIGNER() or ALL_SIGNERS().
Verified **ZeroAddressApp**	The appSigner() of the zero address is the zero address.
Verified **NumberOfOwnersIntegrity**	The number of the wallet owners is three at most.
Verified **OwnerisNonZero**	The zero address is never an owner
Verified **OwnerListNoDuplicates**	The owners array has no duplicates (no two identical owners).
Verified **SignerPolicyCannotExceedOwnerCount**	The signer policy can never exceed the required owners account.
Verified **FirstOwnerIsKYC**	The first wallet owner (owners(0)) is always KYC
Verified **whichFunctionRemovesOwner**	Only the resetSigners() and finishRecovery() functions can remove an owner().

Verified **firstOwnerIsChangedOnlyByRecovery**	The first owner can only be changed by the finishRecovery() function (post-initialization).
Verified **finishRecoveryIntegrity**	finishRecovery() sets the three owners to the three new signers.
Verified **validationSignerPolicyIntegrity**	For a non-app validation, if the validation succeeds, then all relevant signers (according to policy) must be owners.
Verified **entryPointPriviligedFunctions**	execute(), executeBatch() and validateUserOp() are only called by the EntryPoint.
Verified **whichFunctionsChangeWhiteList**	Only the contract can change the app white list and by calling setAppWhitelist().
Verified **whichFunctionsChangeFunderWhiteList**	Only the contract can change the funder white list and by calling setFunderWhitelist().
Verified **whichFunctionsChangeAppSigner**	Only the contract can change the funder white list and by calling setAppKey().

KintoWalletFactory.sol

Assumptions

-

Verified **onceActiveAlwaysActive**	Once a wallet is active (timestamp > 0), it never becomes inactive (timestamp = 0).
createWalletForKYCdOnly	A wallet could only become active (created) for an owner who is KYCd.
Violated **ZeroAddressIsNeitherWalletOwnerNorRecoverer** **Issue: L-02 Wallets can be created without a valid recoverer**	The zero address is never a wallet owner or a wallet recoverer. Violation reason: The factory / wallet doesn't verify that the recoverer address is non-zero **Rule is verified after the latest commit fix.**

SponsorPaymaster.sol

Assumptions

- We assume the contract is in the post-initialization phase.
- Any application has sufficient balance in the paymaster to cover the gas expenses of its users in the validation phase and postOp.

Violated **PaymasterEthSolvency** **Issue:** **H-03 The withdrawTo() method does not update the Paymaster balance after withdrawal**	The sum of user balances is covered by the EntryPoint deposit of the Paymaster. Violation reason: When the owner calls withdrawTo(), the balance won't be updated. **Rule is verified after the latest commit fix.**
Verified **postOpGasCostIsUserFree**	The gas cost post-op cannot depend on the user address. The contract spent amount can only change for the post-op context account.
balanceOnlyIncreasesByDeposit	The balance of any account can only increase by addDepositFor().
Violated **balanceDecreaseIsAtMostMaxOpCost** **Issue:** **M-01 Wrong Balance check in Paymaster**	The balance of any account can decrease at most by the MAX_COST_OF_USEROP(). Violation reason: The actual eth cost balance decrease in postOp() is given by the EntryPoint executor and is unbounded. **Rule is verified after the latest commit fix.**
validationContextIsConsistent	No operation can change the context output of validatePaymasterUserOp().

Violated **validatePayMasterCannotFrontRunEachOther**	A call validatePaymasterUserOp() and postOp() for one sender can never front-run another call for another sender and make it revert. <u>Violation reason:</u> When the two senders share the ETH balance of the same app, where the first of them can drain the balance that will make the postOp revert. **Rule is verified after the latest commit fix.**[5]
Verified **validatePayMasterCannotFrontRunEachOther (post-fix)**	A call validatePaymasterUserOp() and postOp() for one sender can never front-run another call for another sender and make it revert. ● See second assumption above.
lastOperationTimeIsInThePast	The rate, cost and total rate limits last operation time is never in the future.
postOpUpdatesLimits	The postOp() changes the user limits of the op input context (account and sender) only.
onlyOneAppBalanceChangeAtATime	Any operation may change the contract spent amount and balance for one app at a time.
contractSpentMustDecreaseBalance	The contract spent amount cannot decrease, and must increase by the same amount the balance of that contract decreases.

[5] Assuming the app which uses the paymaster charges enough ETH from its users.

Violated **cannotDos_withdrawTokensTo** **Issue: H-03 The withdrawTo() method does not update the Paymaster balance after withdrawal**	No operation can front-run and make a call to withdrawTokensTo() revert Violation reason: When the owner calls to withdrawTo(), the balance won't be updated. **Rule is verified after the latest commit fix.**
onlyUserCanChangeHisParameters	Only the user (or the EntryPoint) can change his own limits.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.
Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects,
and incident response.