OpenZeppelin | security

Lockx.io Audit

Lockx

September 30, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-09-08 To 2025-09-17
Languages	Solidity

Total Issues	22 (22 resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	6 (6 resolved)
Notes & Additional Information	13 (13 resolved)

Scope

OpenZeppelin audited the Lockx-Security/Lockx-Ethereum-Contracts repository at commit 3828518.

In scope were the following files:

```
contracts
├── Deposits.sol
├── Lockx.sol
├── SignatureVerification.sol
└── Withdrawals.sol
```

System Overview

Lockx is a smart contract wallet framework that leverages NFT-based access control to manage user-owned wallets, referred to as Lockboxes. A Lockbox can be created by calling one of the `createLockBox` functions of the `Lockx` contract. Four variations of this function exist, depending on the type of assets deposited at creation time:

- Native ETH
- ERC-20 tokens
- ERC-721 tokens
- Mixed assets (batch variant)

When a new Lockbox is created, the protocol mints a non-transferable ERC-721 token (Lockbox NFT) to the user. These tokens are *soulbound*: they cannot be transferred, and the only way to remove them is by destroying the associated Lockbox. Destruction is only permitted if the Lockbox has no remaining funds.

Each Lockbox is initialized with a user-supplied public key (`lockboxPublicKey`). This key is used to verify signatures on all restricted wallet actions, such as funds transfers and swaps. As a result, Lockbox operations require both of the following:

- For the Lockbox NFT owner to call the function on-chain
- A valid signature from the registered `lockboxPublicKey`

This effectively creates a 2-of-2 multisig model, where both the NFT owner and the external signing key must participate to authorize transactions.

For added security, Lockx supports public key rotation. If the signing key is compromised (e.g., through a phishing approval attack), the Lockbox owner can update the registered key. Since all actions must still be invoked by the NFT holder, a compromised signing key alone is insufficient to seize control of the wallet.

Security Model and Trust Assumptions

The proper functioning of the system under review is based on the trust of privileged roles, trust boundaries, and various security assumptions, such as:

- The owner of the system will honestly handle the default `metadataURI` of the Lockbox NFTs.
- Whitelisted routers are not malicious, and only swap-related functions are invoked.
- The system owner and whitelisted routers are trusted entities and behave as expected.
- The system is properly set up and will be appropriately initialized upon deployment.

Critical Severity

C-01 Treasury Lockbox Ownership Vulnerability

The `Withdrawals` contract hard-codes `TREASURY_LOCKBOX_ID = 0` and credits all swap fees to that bookkeeping slot. However, the first Lockbox minted via any `createLockbox` function is assigned `tokenId == 0` because `_nextId` is declared but not initialized. As a result, the first user to mint a Lockbox becomes the owner of the treasury Lockbox and can subsequently withdraw all accumulated protocol fees. This exposes all treasury funds to potential exploitation by malicious actors, with there being no mechanism to recover the funds if they get withdrawn.

Consider creating the treasury Lockbox (`tokenId == 0`) during the `Lockx` contract's construction, ensuring that ownership of the treasury is securely assigned to the deployer (or another designated entity). Doing so will prevent unintended user control over treasury funds.

Update: *Resolved in pull request #15 at commit 821bb91.*

Medium Severity

M-01 Lack of `calldata` Validation Allows Owner to Invoke Arbitrary Router Functions

The `swapInLockbox` function only verifies that the target address is an allowed router, but does not verify the `calldata` selector or the function signature being executed on that router. A malicious owner (or an owner account that is compromised) can craft calldata that calls any public function on the router contract. This could potentially be very dangerous given the fact that there is no control over what router functions are available, plus the potential "hijacked" call will be from the `Lockx` contract with all user assets inside. For example, calling the `sweep` function of a Uniswap router could result in stuck ERC-20 tokens in the `Lockx` contract. More serious attack vectors could be crafted based on the specific router allowance and potential future updates.

Consider constructing the `calldata` inside the `swapInLockbox` function or validating the function signature if it is pre-packed.

Update: *Resolved in pull request #9 at commit 062b6ef.*

M-02 Incompatibility With Rebasing Tokens

The system tracks ERC-20 balances for each Lockbox using the `_erc20Balances` mapping. Balances are updated only on deposit and withdrawal events, while the actual ERC-20 token contract is queried via `balanceOf` only during deposits. However, this approach fails to account for rebasing tokens, whose balances may change independently of user actions. As a result, the `_erc20Balances` mapping can become inaccurate.

This discrepancy may negatively affect the entire system:

- If a rebasing token increases in value, excess funds could become locked within the contract.
- If a rebasing token decreases in value, multiple Lockbox owners of the same token may compete to withdraw more than what is available.

Consider clearly documenting the fact that rebasing tokens are not supported and should not be deposited into a Lockbox. Alternatively, consider implementing explicit handling for rebasing tokens (e.g., updating balances on every withdrawal using `balanceOf`).

Update: *Resolved in pull request #13 at commit 69db56f. Explicit warning in NatSpec documentation has been added, which states that rebasing tokens are not supported.*

Low Severity

L-01 Inconsistent Recipient Validation Across Withdrawal Functions

The withdrawal logic in `Lockx.sol` applies inconsistent validation rules depending on the asset type:

- ERC-721 withdrawals explicitly require that the recipient is not the contract itself (`recipient != address(this)`).

- ETH and ERC-20 withdrawals do not perform the `recipient != address(this)` check, allowing ETH to be sent back to the `Lockx` contract.
- Batch withdrawals do not perform the `recipient != address(this)` check for all three asset types (ETH, ERC-20, and ERC-721).

Specifically for ETH withdrawals, when the recipient is the contract itself, the call returns the ETH right back. As a result, the real ETH balance remains unchanged, but the internal mapping is already decremented. This inconsistency creates divergent behavior between the withdrawal functions.

Consider always validating that the recipient address is not the contract itself to make the validation requirements consistent and prevent funds from getting stuck in the contract.

Update: *Resolved in pull request #6 at commit 8a5549c.*

L-02 Insufficient Input Validation Enables Creation of Empty Lockboxes

The `createLockboxWithBatch` function in `Lockx.sol` allows users to create a Lockbox without providing any assets (ETH, ERC-20, or ERC-721). Since there is no enforced requirement to supply a minimum asset amount, an "empty" Lockbox can be created. However, this design contradicts the logic contained in the asset-type-specific functions (`createLockboxWithETH`, `createLockboxWithERC20`, and `createLockboxWithERC721`). While each of these functions includes explicit checks to ensure that a non-zero asset amount is provided, these checks can be bypassed simply by calling `createLockboxWithBatch` with no assets.

Consider implementing non-zero-amount checks in the `createLockBoxWithBatch` function to prevent the creation of empty Lockboxes, ensuring consistency with the intended system design.

Update: *Resolved in pull request #7 at commit fcdba44.*

L-03 Swap Fee Rounding Mechanism Favors Users Over Protocol

In the `Withdrawals` contract, the swap logic rounds down, which favors the user. This means that the protocol consistently collects slightly less than the intended fee. While this does not pose a direct security risk, it is generally considered bad practice. In Defi protocols, rounding should typically favor the protocol (rounding up) to ensure that no attack vectors can be composed out of it.

Consider rounding in favor of the protocol to eliminate any potential edge cases that could be exploited.

Update: *Resolved in pull request #8 at commit 32f0145.*

L-04 Orphaned ETH on Direct Transfers to `Lockx`

If ETH is sent directly to the `Lockx` contract (e.g., via plain transfer, selfdestruct, or misrouted router call), the ETH is accepted but not accounted to any Lockbox or user balance. The `Lockx` contract lacks `receive()`/`fallback()` logic to properly account for such deposits, and there is no explicit recovery mechanism that is tied to ownership or Lockbox state. As a result, this ETH becomes orphaned (i.e., visible in `address(this).balance` but inaccessible through protocol logic), effectively leaving the funds stuck.

Consider implementing logic to handle unintended ETH transfers, for example making the `receive()` function revert ETH coming from any sender other than the whitelisted routers.

Update: *Resolved at pull request #10 at commit 881969c.*

L-05 Potentially Misleading Event Emission

The functionality for rotating a `LockBox` public key does not validate whether the newly registered key differs from the existing one. As a result, the contract may emit an event even though no actual change has occurred. Such misleading events could confuse integrators or off-chain systems that rely on event logs to track meaningful state changes.

Consider introducing a check to ensure that the new public key is different from the current one before updating state and emitting the event. This will prevent unnecessary or misleading event emissions.

Update: *Resolved at pull request #11 at commit 758bac8.*

L-06 Unverified `referenceId` Could Mislead Off-Chain Integrators

When a new NFT is minted, the `Minted` event is emitted with a `referenceId`. This value is passed as an argument to the `createLockBox` functions and is documented for off-chain tracking, but it is not stored in the contract's storage. Similarly, for access-controlled actions that require signature verification (e.g., key rotation, withdrawal, swap), the `referenceId` for the specified `tokenId` is included in the signed message, and an event containing the `referenceId` is emitted.

However, since the contract does not store or verify `referenceId` values, any arbitrary value can be provided. As a result, external integrators relying on `referenceId` may draw incorrect conclusions, because a token owner could perform a valid action using an incorrect `referenceId` either intentionally or accidentally.

Consider storing the `referenceId` for each minted `tokenId` in contract storage. This allows for the validation of `referenceId` during restricted actions, ensuring that the emitted events accurately reflect the correct off-chain tracking identifiers.

***Update:** Resolved in pull request #12 at commit 89046e7.*

Notes & Additional Information

N-01 Inefficient O(n) Duplicate Detection

The `batchWidraw` function of the `Withdrawals` contract performs O(n) duplicate detection for ERC-20 and ERC-721 arrays using a “seen” structure that relies on storage reads/writes. While functionally correct, this approach is gas-inefficient and increases the "state-touching" complexity during a single call that only needs transient checks. Since the arrays are user-supplied, duplicates can be prevented more efficiently by requiring sorted input and enforcing a strictly increasing order in a single pass. This maintains O(n) complexity but relies only on calldata/memory comparisons, avoiding unnecessary storage writes. It also guarantees both sortedness and non-duplication in a simpler way.

Consider refactoring the duplicate-detection logic in `batchWithdraw` to require user-supplied arrays to be sorted, and enforcing strictly increasing checks during iteration. This optimization reduces gas costs and simplifies state interactions while ensuring correctness.

Update: *Resolved at pull request #14 at commit 3c8b351.*

N-02 Lack of Exact Token-Out Support in `swapInLockbox` Function

The `swapInLockbox` function currently only supports swaps where the input amount is specified (`exactTokenIn`). It does not support the complementary `exactTokenOut` mode, where the caller specifies the desired output amount and the contract supplies whatever maximum input is necessary.

Consider adding support for `exactTokenOut`. Supporting both modes is a common practice in swap integrations because it increases flexibility for callers.

Update: *Resolved in pull request #27 at commit 03ed8a2.*

N-03 Redundant Zero-Address Check in `swapInLockbox`

In line 419 of `Withdrawals.sol`, the `swapInLockbox` function checks `if (target == address(0))` before calling `_isAllowedRouter(target)` in line 424. However, `_isAllowedRouter` already rejects the zero address, making the explicit check redundant and unnecessary.

Consider removing the redundant `target == address(0)` check to simplify the logic and avoid duplicating validations.

Update: *Resolved in pull request #26 at commit 8bed22e.*

N-04 Inefficient Validation Order in Withdrawal Functions

Deferring validations leads to unnecessary gas consumption on reverts and makes it harder to identify the root cause of failures.

Throughout the codebase, multiple instances of withdrawal functions deferring basic, deterministic validations until after signature verification were identified:

- line 106, in the `withdrawETH` function
- line 170, in the `withdrawERC20` function
- line 239, in the `withdrawERC721` function

Consider rendering the logic so that deterministic checks (e.g., input validation, state conditions, etc.) occur before signature verification. This ensures that failures are caught earlier, reduces wasted gas, and provides clearer revert reasons.

Update: *Resolved in pull request #23 at commit 51cb11d.*

N-05 Code Duplication Across Withdrawal, Lockbox Creation, and Swap Logic

Throughout the codebase, multiple instances of duplicated logic were identified:

- In the `batchWithdraw` function, in line 318, ETH withdrawal code that is already present in line 105 is duplicated. Same for ERC-20 and ERC-721 withdrawal paths, where the same validation and transfer logic are duplicated.
- These four functions (`createLockboxWithETH`, `createLockboxWithERC20`, `createLockboxWithERC721`, and `createLockboxWithBatch`) share the same verification and creation boilerplate.
- In `swapInLockbox`, repeated logic appears in line 538 and line 561.

Consider refactoring these cases of duplicated code for clarity, maintainability, and reduced attack surface.

Update: *Resolved in pull request #24 at commit 4b6fec9.*

N-06 Redundant Use of `forceApprove`

The `swapInLockbox` function contains redundant logic around the use of `forceApprove`:

- Line 478: The function calls `forceApprove` to reset allowance to 0. This is unnecessary, since calling `approve(spender, 0)` on an ERC-20 token never reverts — even for tokens with the "zero-first" quirk (e.g., USDT). Using `forceApprove` here only adds gas overhead. A direct `safeApprove` call would be sufficient.

- Line 477: The function checks if allowance is non-zero, and if so, calls `forceApprove(0)` before setting a new allowance. This pre-check is unnecessary because `forceApprove` already includes logic to handle the “zero-first” quirk. The code could simply call `forceApprove` directly without the additional condition.

Consider simplifying the allowance handling in `swapInLockbox` by using `safeApprove(spender, 0)` when resetting to zero and removing the redundant non-zero allowance check before calling `forceApprove`. This reduces gas overhead and avoids duplicating logic already covered by `forceApprove`.

Update: *Resolved in pull request #25 at commit f86f594.*

N-07 Redundant Data in Signed Messages

The system encodes both `tokenId` and `msg.sender` in signed messages. This encoding is redundant because the contract already enforces authorization through `tokenId` ownership checks, making the additional binding to `msg.sender` unnecessary from a security perspective. Similarly, `structHash` includes both `tokenId` and `dataHash`, despite `tokenId` already being part of the hashed data, which introduces unnecessary duplication.

Consider removing redundant fields (`msg.sender` and the duplicate `tokenId`) from the signed messages to simplify the implementation and reduce potential confusion without impacting security.

Update*: Resolved in pull request #22 at commit fa0739d.*

N-08 Unnecessary Parameters and Variables

Throughout the codebase, multiple instances of redundant parameters and variables were identified:

- The `createLockboxWithETH`, `createLockboxWithERC20`, `createLockboxWithERC721`, and `createLockboxWithBatch` functions all take a `to` parameter. However, Lockboxes can only ever be created for `msg.sender`, making `to` unnecessary and potentially confusing.
- In `createLockboxWithBatch`, the `amountETH` parameter is redundant. The contract should simply rely on `msg.value` to track the deposited ETH.
- In `verifySignature`, the `messageHash` parameter is unnecessary. The contract must reconstruct the hash internally from the signed payload, making an externally

supplied hash redundant and potentially misleading. In addition, the local variable `dataHash` is also unnecessary, as it only temporarily stores a reconstructed value that could be used inline.

Consider removing or refactoring the aforementioned instances of redundant variables and parameters for improved clarity and gas efficiency.

Update: *Resolved at pull request #16 at commit c3d6840.*

N-09 Unused Error Definition

Defining errors that are not referenced introduces unnecessary noise and may reduce code clarity. In the `Lockx` contract, the `UseDepositETH` error is defined but is never utilized within the entire codebase.

Consider removing the unused error to improve the maintainability and clarity of the codebase.

Update: *Resolved in pull request #17 at commit 1be6b18.*

N-10 Redundant Token Existence Check

In the `setTokenMetadataURI` function of the `Lockx` contract, line 268 verifies token ownership by calling `ownerOf` from OpenZeppelin's `ERC721.sol`. This function already ensures that the token with the specified `tokenId` exists, reverting with a custom error otherwise. Since line 267 independently checks for token existence and reverts if the token does not exist, this validation is redundant.

Consider removing the code in line 267 to improve code readability and achieve minor gas savings.

Update: *Resolved in pull request #18 at commit cc62ebc.*

N-11 Access-Control Function Could Be a Modifier

In the `Deposits` contract, the `_requireOwnsLockBox` `internal` function is used across multiple functions to restrict access exclusively to the owner of a specified `tokenId`. While functional, this pattern could be expressed more clearly as a modifier. Using a modifier for ownership validation improves readability and emphasizes intent.

Consider placing the call to `_requireOwnsLockBox` inside a modifier and using that modifier instead to improve code clarity and maintainability.

Update: *Resolved in pull request #19 at commit d8947b6.*

N-12 Inconsistent Naming Convention For Internal Functions

Throughout the codebase, most `internal` functions are prefixed with an underscore (_) to distinguish them from `public` and `external` functions. However, a few `internal` functions do not follow this convention:

- `initialize` in `SignatureVerification.sol`
- `verifySignature` in `SignatureVerification.sol`

This inconsistency reduces the overall code readability and can make it harder to quickly identify the visibility of functions when reviewing the code.

Consider adopting a consistent naming convention by prefixing all `internal` functions with an underscore. Doing so will improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #20 at commit 4d17936.*

N-13 Inconsistent Indexing Across the Codebase

Several data structures in the codebase require indexing, but there is no consistent approach to the starting index. Some structures begin indexing from 0, while others start from 1:

- The `_erc20TokenAddresses[tokenId]` array is indexed via the `_erc20Index` mapping starting from 1.
- `tokenId` of Lockx NFTs starts from 0.
- The `nonce` value for each Lockx NFT starts from 1.

Inconsistent indexing can lead to confusion, increase the likelihood of off-by-one errors, and reduce code readability.

Consider standardizing indexing across the codebase by starting all indices from 0, following common Solidity and general programming conventions. This will help improve code maintainability and reduce the potential for indexing errors.

Update: *Resolved in pull request #21 at commit b2732ac.*

Conclusion

The system under review is a smart contract wallet framework that leverages NFT-based access control to manage user-owned wallets, referred to as Lockboxes. The system was found to be well-structured and clearly documented, and the use of an NFT-based approach for token locking is innovative.

Nonetheless, the codebase would benefit from improved design patterns and best practices, particularly around the signature schemes that are heavily used throughout the system. These areas are detailed in the findings of this report. The Lockx team is appreciated for being highly responsive during the audit, promptly providing the necessary information and addressing all questions from the audit team.