RENASCENCE

Locksmith SDK Audit Report

Version 2.0

Audited by:

MiloTruck

bytes032

January 27, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Locksmith SDK

Locksmith is an open-source on-chain permission primitive designed to provide security composability across smart contracts, applications, and ecosystems. The Locksmith SDK can be used by dApp and wallet developers alike to add fully decentralized, abstracted account permissions that are fully interoperable across on-chain applications.

2.2 Overview

Project	Locksmith SDK
Repository	locksmith-core
Commit Hash	109e33db3fec...
Mitigation Hash	474ee72ead9c...
Date	January 2024

2.3 Issues Found

Severity	**Count**
High Risk	1
Medium Risk	3
Low Risk	1
Informational	2
Total Issues	**7**

3 Findings Summary

ID	Description	Status
H-1	Soulbound checks and `_manageIndexes()` can be bypassed by specifying duplicate key IDs in `ids`	Resolved
M-1	`KeyLocker.redeemKeys()` can get permanently bricked	Resolved
M-2	`_manageIndexes()` updates `addressKeys` and `keyHolders` for transfers where `value = 0`	Resolved
M-3	`KeyLocker.supportsInterface()` returns `false` for `type(IERC1155Receiver).interfaceId`	Resolved
L-1	KeyLocker's checks on `onERC1155Received` can easily be circumvented	Resolved
I-1	Events should be named using the CapWords style	Resolved
I-2	Using-for declarations should be declared in the `Locksmith` contract	Resolved

4 Findings

4.1 High Risk

[H-1] Soulbound checks and `_manageIndexes()` can be bypassed by specifying duplicate key IDs in `ids`

Context:

- Locksmith.sol#L588-L594
- Locksmith.sol#L616-L619

Impact: Users who have more keys for a certain key ID than `soulboundKeyAmounts` can transfer all their keys to another address.

Additionally, `addressKeys` and `keyHolders` will contain key IDs and addresses respectively that should have been removed after a transfer.

Description: The `_update()` function enforces soulbound checks for users by ensuring that the remaining balance after the transfer is not less than `soulboundKeyAmounts` for each ID in `ids`:

```
for(uint256 x = 0; x < ids.length; x++) {
    // we need to allow address zero during minting,
    // and we need to allow the locksmith to violate during burning
    if ( (from != address(0)) && (to != address(0)) &&
         (balanceOf(from, ids[x]) - values[x]) < soulboundKeyAmounts[from][ids[x]]) {
            revert SoulboundTransferBreach();
    }
```

However, this check can be bypassed by specifying duplicate key IDs in `ids`. For example:

- Alice has two keys of `id = 1`.
- `soulboundKeyAmounts[alice][1] = 1`, which means that one of Alice's keys should not be transferable.
- Alice calls `safeBatchTransferFrom()` with:
  - `ids = [1, 1]`
  - `values = [1, 1]`
  - The check above passes as `balanceOf(alice, ids[x]) - values[x] = 1` for all `x` in `ids`.
  - Therefore, both of Alice's keys are transferred to another address.

Similarly, `_manageIndexes()` removes from `addressKeys` and `keyHolders` under the following condition:

```
// lets keep track of each key that is moving
if(balanceOf(from, id) == value) {
    addressKeys[from].remove(id);
    keyHolders[id].remove(from);
}
```

However, if `ids` contains duplicate key IDs as shown in the example above, `balanceOf()` will not be equal to `value`. Therefore, `addressKeys` and `keyHolders` will not be updated even when the user has transferred all his keys.

Recommendation: In `_update()`, consider ensuring that `ids` does not contain duplicate key IDs:

```
        for(uint256 x = 0; x < ids.length; x++) {
+           for (uint256 y = 0; y < x; y++) {
+               if (ids[x] == ids[y]) {
+                   revert DuplicateKeyID();
+               }
+           }

            // we need to allow address zero during minting,
            // and we need to allow the locksmith to violate during burning
            if ( (from != address(0)) && (to != address(0)) &&
                 (balanceOf(from, ids[x]) - values[x]) <
                 soulboundKeyAmounts[from][ids[x]]) {
                    revert SoulboundTransferBreach();
            }
```

Locksmith: Fixed in commit 3205440 by performing the transfer first and checking if the remaining balance of the `from` address is less than soulbound amount. `_manageIndexes()` was also modified to accommodate this change.

Renascence: Verified. Since the remaining balance is checked against the soulbound amount, the exploit described above is no longer possible.

4.2 Medium Risk

[M-1] `KeyLocker.redeemKeys()` can get permanently bricked

Context: KeyLocker.sol

Impact: `KeyLocker.redeemKeys()` will permanently revert when the Locksmith's minted keys reach a certain value.

Description: The `redeemKeys` function is designed to redeem keys left in the locker. An issue arises from the `inspectKey` call within this function.

```
    function redeemKeys(address locksmith, uint256 rootKeyId, uint256 keyId, uint256
    amount) external {
        ILocksmith l = ILocksmith(locksmith);

        // can't redeem zero
        if(amount < 1) {
            revert InvalidInput();
        }

        // make sure the key used is actually a root key
>        (,,uint256 rootRing,bool isValidRoot,) = l.inspectKey(rootKeyId); // @audit
if the set grows too big this will make the function uncallable
```

The problem lies in the `Locksmith.inspectKey` call, where an attempt is made to figure out the keys associated with the ring by invoking `EnumerableSet.values()`.

```
function inspectKey(uint256 keyId) public view returns (bool, bytes32, uint256, bool,
uint256[] memory) {
        uint256[] memory empty = new uint256[](0);

        // the key is a valid key number
        return ((keyId < keyCount),
            // the human readable name of the key
            keyData[keyId].name,
            // ring Id of the key
            keyRingAssociations[keyId],
            // the key is a root key
            _isRootKey(keyId),
            // the keys associated with the ring
            keyId >= keyCount ? empty :
            ringRegistry[keyRingAssociations[keyId]].keys.values());
    }
```

As shown in OZ's documentation, this might be a problem:

```
 * WARNING: This operation will copy the entire storage to memory, which can be quite
 expensive. This is designed
 * to mostly be used by view accessors that are queried without any gas fees.
 Developers should keep in mind that
 * this function has an unbounded cost, and using it as part of a state-changing
 function may render the function
 * uncallable if the set grows to a point where copying to memory consumes too much
 gas to fit in a block.
 */
function values(UintSet storage set) internal view returns (uint256[] memory) {
```

Even though `inspectKey` is a view function, `redeemKeys` is state-changing. Then, because keys are not removed from the set when burned, we can assume that the set will always increase.

As a result, when the data gets to a point where its too big, any keys in `KeyLocker.sol` will be forever locked.

Recommendation: Given that only two return values from `inspectKeys` are used by `redeemKeys`, the easiest thing to do is expose a getter function from Locksmith that would allow the fetching of the root ring that needs to be used later and use it and `isRootKey` instead.

```
### Locksmith.sol

+    function getRootRingId(uint256 keyId) public view returns (uint256) {
+        return keyRingAssociations[keyId];
+    }
```

```
### Keylocker.sol
 function redeemKeys(address locksmith, uint256 rootKeyId, uint256 keyId, uint256
 amount) external {
-        (,,uint256 rootRing,bool isValidRoot,) = l.inspectKey(rootKeyId);
+        uint256 rootRing = l.getRootRingId(rootKeyId);
+        uint256 isValidRoot = l.isRootKey(rootKeyId)
```

Locksmith: Fixed in commit `474ee72`.

Renascence: Verified, the recommended fix was implemented. An additional `keyId < keyCount` check was also added in `getRingId()` to prevent the function from being called with a non-existent `keyId`.

[M-2] `_manageIndexes()` updates `addressKeys` and `keyHolders` for transfers where `value = 0`

Context: Locksmith.sol#L620-L623

Impact: Users can add themselves to `addressKeys` and `keyHolders` even if they do not hold the corresponding key, causing `getKeysForHolder()` and `getHolders()` to return incorrect values.

Description: The `_manageIndexes()` function adds to `addressKeys` and `keyHolders` when the `to` address of a transfer is non-zero:

```
if(address(0) != to) {
    addressKeys[to].add(id);
    keyHolders[id].add(to);
}
```

However, since the function does not check if `value` is non-zero, `_manageIndexes()` will still update both mappings for transfers with zero value.

This allows users to add themselves to both mappings for any arbitrary key ID by performing a self-transfer with zero value. For example, a user can call `safeTransferFrom()` with:

- `from` and `to` as their own address
- `id` as the key ID they want to add
- `value = 0`

After the transfer is performed, `getKeysForHolder()` will include the new key ID and `getHolders()` will include their address, even though they do not own a key with that ID.

Recommendation: Only update `addressKeys` and `keyHolders` if `value` is non-zero:

```diff
- if(address(0) != to) {
+ if (address(0) != to && value != 0) {
    addressKeys[to].add(id);
    keyHolders[id].add(to);
}
```

Locksmith: Fixed in commit `fd32f9c`.

Renascence: Verified, the recommended fix was implemented.

[M-3] `KeyLocker.supportsInterface()` returns `false` for `type(IERC1155Receiver).interfaceId`

Context: KeyLocker.sol#L26-L30

Impact: `KeyLocker.supportsInterface()` incorrectly returns `false` when called with the interface ID of `IERC1155Receiver`, which violates the ERC-1155 specification and could break composability with other contracts.

Description: `supportsInterface()` in `KeyLocker.sol` checks the following interface IDs:

```
return interfaceId == type(IKeyLocker).interfaceId ||
       interfaceId == type(IERC165).interfaceId ||
       interfaceId == type(ERC1155Holder).interfaceId;
```

However, the function incorrectly checks for `type(ERC1155Holder).interfaceId` instead of `type(IERC1155Receiver).interfaceId`.

Recommendation: Instead of including the interface IDs of all inherited contracts, consider calling `super.supportsInterface()`:

```diff
  return interfaceId == type(IKeyLocker).interfaceId ||
-         interfaceId == type(IERC165).interfaceId ||
-         interfaceId == type(ERC1155Holder).interfaceId;
+         super.supportsInterface(interfaceId);
```

This will also call the `supportsInterface()` function of `ERC1155Holder`.

For consistency, consider modifying `Locksmith.supportsInterface()` as well:

```diff
  return
-     interfaceId == type(IERC1155).interfaceId ||
-     interfaceId == type(IERC1155MetadataURI).interfaceId ||
      interfaceId == type(ILocksmith).interfaceId ||
-     interfaceId == type(IERC165).interfaceId;
+     super.supportsInterface(interfaceId);
```

Locksmith: Fixed in commit `8d95848`.

Renascence: Verified, the recommended fix was implemented.

4.3 Low Risk

[L-1] KeyLockers checks on `onERC1155Received` can easily be circumvented

Context: KeyLocker.sol

Description: The `KeyLocker` contract implements `ERC1155Holder`.

```
contract KeyLocker is IKeyLocker, ERC1155Holder {
```

The `onERC1155Received` function is overridden to include a check ensuring that the caller is a valid `Locksmith` contract.

```
function onERC1155Received(address, address from, uint256 keyId, uint256 count, bytes
memory)
    public virtual override returns (bytes4) {
    // make sure the locksmith is a proper one
    // @audit-issue the same check is not applied for batch receive
    if(!IERC165(msg.sender).supportsInterface(type(ILocksmith).interfaceId)) {
        revert InvalidInput();
    }

    // we are going to accept this key no matter what.
    emit keyLockerDeposit(from, msg.sender, keyId, count);

    // success
    return this.onERC1155Received.selector;
}
```

However, it's crucial to note that this check can be bypassed by sending tokens as a batch, as it triggers the `onERC1155BatchReceived` callback.

Recommendation:

```
+    function onERC1155BatchReceived(address,address,uint256[] memory,uint256[]
memory, bytes memory)
+         public virtual override returns (bytes4) {
+        if(!IERC165(msg.sender).supportsInterface(type(ILocksmith).interfaceId)) {
+            revert InvalidInput();
+        }
+
+        // we are going to accept this key no matter what.
+        emit keyLockerDeposit(from, msg.sender, keyId, count);
+
+         // success
+         return this.onERC1155BatchReceived.selector;
+    }
```

Locksmith: Fixed in commit `11a50d5`.

Renascence: Verified, the recommended `onERC1155BatchReceived` callback was added.

4.4 Informational

[I-1] Events should be named using the CapWords style

Context: KeyLocker.sol, Locksmith.sol

Description: Across the codebase, all of the events are in camel case format, e.g.

```
emit keyBurned()
emit keyMinted()
emit keyLockerDeposit()
emit keyLockerWithdrawal()
```

However, as per Solidity's documentation, the names should be named using the CapWords style.

Recommendation:

```
+emit KeyBurned()
+emit KeyMinted()
+emit KeyLockerDeposit()
+emit KeyLockerWithdrawal()
```

Locksmith: Fixed in commit `2737d23`.

Renascence: Verified, all events now adhere to the CapWords style.

[I-2] Using-for declarations should be declared in the `Locksmith` contract

Context: Locksmith.sol#L24-L25

Description: In `LockSmith.sol`, the using-for declarations for `EnumerableSet` are declared in the global scope:

```
using EnumerableSet for EnumerableSet.UintSet;
using EnumerableSet for EnumerableSet.AddressSet;
```

It's best to declare them in contracts where they are needed, which would be the `LockSmith` contract.

Recommendation: Move both using-for declarations into the `LockSmith` contract:

```
  contract Locksmith is ILocksmith, ERC1155 {
+     using EnumerableSet for EnumerableSet.UintSet;
+     using EnumerableSet for EnumerableSet.AddressSet;
```

Locksmith: Fixed in commit `6017d97`.

Renascence: Verified, the recommended fix was implemented.