MiloTruck

LUKSO

Security Review Report

October, 2023

Table of Contents

Table of Contents **1**
Introduction **2**
About MiloTruck 2
Disclaimer 2
Executive Summary **3**
About LUKSO 3
Repository Details 3
Scope 3
Issues Found 3
Findings **4**
Summary 4
Medium Severity Findings 5
M-01: LSP0 accounts can be transferred with a “poisoned” `_renounceOwnershipStartedAt` 5
M-02: Invalid validation for LSP17 extensions in `_getPermissionRequiredToSetDataKey()` 7
M-03: `_extractCallType()` returns `0x00000000` as the required call type for empty calls 9
M-04: Missing universal receiver callback in `renounceOwnership()` when called by the owner 11
Low Severity Findings 13
L-01: Return data from `execute()` and `executeRelayCall()` is corrupted when calling `LSP0ERC725AccountCore::executeBatch()` 13
L-02: Functions with a `0x00000000` selector cannot be added to allowed calls 15
Informational Findings 16
I-01: Unnecessary initialization to 0 in for-loop 16
I-02: `_isValidNonce()` can be simplified 16

Introduction

About MiloTruck

MiloTruck is an independent security researcher who specializes in smart contract audits. Having won multiple audit contests, he is currently one of the top wardens on Code4rena. He is also a Senior Auditor at Trust Security and Associate Security Researcher at Spearbit.

For security consulting, reach out to him on Twitter - *@milotruck*

Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

Executive Summary

About LUKSO

LUKSO is the digital base layer for the New Creative Economies. It provides creators and users with future-proof tools and standards to unleash their creative force in an open interoperable ecosystem.

Repository Details

Repository	https://github.com/lukso-network/lsp-smart-contracts
Commit Hash	0c1738619818bcf2e01636f31782886b26fb91d7
Language	Solidity

Scope

contracts/LSP0ERC725Account/

- ILSP0ERC725Account.sol
- LSP0Constants.sol
- LSP0ERC725Account.sol
- LSP0ERC725AccountCore.sol
- LSP0ERC725AccountInit.sol
- LSP0ERC725AccountInitAbstract.sol

contracts/LSP6KeyManager/

- ILSP6KeyManager.sol
- LSP6Constants.sol
- LSP6Errors.sol
- LSP6KeyManager.sol
- LSP6KeyManagerCore.sol
- LSP6KeyManagerInit.sol
- LSP6KeyManagerInitAbstract.sol
- LSP6Utils.sol
- LSP6Modules/
  - LSP6ExecuteModule.sol
  - LSP6ExecuteRelayCallModule.sol
  - LSP6OwnershipModule.sol
  - LSP6SetDataModule.sol

Issues Found

Severity	Count
High	0
Medium	4
Low	2
Informational	2

Findings

Summary

ID	Description	Severity
M-01	LSP0 accounts can be transferred with a “poisoned” `_renounceOwnershipStartedAt`	Medium
M-02	Invalid validation for LSP17 extensions in `_getPermissionRequiredToSetDataKey()`	Medium
M-03	`_extractCallType()` returns `0x00000000` as the required call type for empty calls	Medium
M-04	Missing universal receiver callback in `renounceOwnership()` when called by the owner	Medium
L-01	Return data from `execute()` and `executeRelayCall()` is corrupted when calling `LSP0ERC725AccountCore::executeBatch()`	Low
L-02	Functions with a `0x00000000` selector cannot be added to allowed calls	Low
I-01	Unnecessary initialization to 0 in for-loop	Informational
I-02	`_isValidNonce()` can be simplified	Informational

Medium Severity Findings

M-01: LSP0 accounts can be transferred with a "poisoned" `_renounceOwnershipStartedAt`

Bug Description

Ownership of LSP0 accounts are transferred through a two-step process.

First, the owner calls `transferOwnership()` to nominate a `pendingOwner`:

LSP14Ownable2Step.sol#L165-L166

```
        _pendingOwner = newOwner;
        delete _renounceOwnershipStartedAt;
```

Afterwards, the pending owner calls `acceptOwnership()` to becomes the new owner:

LSP14Ownable2Step.sol#L176-L177

```
        _setOwner(msg.sender);
        delete _pendingOwner;
```

However, as `_renounceOwnershipStartedAt` is only deleted when `transferOwnership()` is called, LSP0 accounts can be transferred to new owners with a non-zero `_renounceOwnershipStartedAt` by doing the following:

1. Call `transferOwnership()` to nominate a pending owner.
2. Using `execute()`, perform a delegate call that overwrites `_renounceOwnershipStartedAt` to any value.
3. The pending owner calls `acceptOwnership()` to gain ownership of the LSP0 account.

This could be problematic depending on what `_renounceOwnershipStartedAt` is set to.

If it is set to an extremely large value, such as close to `type(uint256).max`, `renounceOwnership()` will always revert due to this check, making the function DOSed.

If it is set to `block.number - RENOUNCE_OWNERSHIP_CONFIRMATION_DELAY`, `renounceOwnership()` will be in the confirmation period for the next 200 blocks. If the new owner accidentally calls `renounceOwnership()`, he will instantly lose ownership of the account rather than initiate the process.

Impact

By abusing delegate call, a previous owner can transfer ownership of an LSP0 account with `_renounceOwnershipStartedAt` as a malicious value. This could cause:

- `renounceOwnership()` to be DOSed for the new owner.
- Renouncing ownership to be a single-step process for the next 200 blocks.

Note that the value of `_renounceOwnershipStartedAt` is not permanent; the new owner can call `transferOwnership()` with `newOwner = address(0)` to reset `_renounceOwnershipStartedAt` to `0`.

Recommended Mitigation

Consider deleting `_renounceOwnershipStartedAt` when `acceptOwnership()` is called:

LSP14Ownable2Step.sol#L176-L177

```
        _setOwner(msg.sender);
        delete _pendingOwner;
+       delete _renounceOwnershipStartedAt;
```

Team Response

Fixed in PR #775.

M-02: Invalid validation for LSP17 extensions in `_getPermissionRequiredToSetDataKey()`

Bug Description

When setting data for LSP17 extensions, `_getPermissionRequiredToSetDataKey()` checks that the data values are either 20 bytes long or empty:

LSP6SetDataModule.sol#L322-L330

```
            // LSP17Extension:<bytes4>
        } else if (bytes12(inputDataKey) == _LSP17_EXTENSION_PREFIX) {
            // CHECK that `dataValue` contains exactly 20 bytes, which corresponds to an address for
a LSP17 Extension
            if (inputDataValue.length != 20 && inputDataValue.length != 0) {
                revert InvalidDataValuesForDataKeys(
                    inputDataKey,
                    inputDataValue
                );
            }
```

However, for LSP0 accounts, `_getExtensionAndForwardValue()` also considers data that is 21 bytes long as valid:

LSP0ERC725AccountCore.sol#L841-L851

```
        // CHECK if the `extensionData` is 21 bytes long
        // - 20 bytes = extension's address
        // - 1 byte `0x01` as a boolean indicating if the contract should forward the value to the
extension or not
        if (extensionData.length == 21) {
            // If the last byte is set to `0x01` (`true`)
            // this indicates that the contract should forward the value to the extension
            if (extensionData[20] == 0x01) {
                // Return the address of the extension
                return (address(bytes20(extensionData)), true);
            }
        }
```

More specifically, a `LSP17Extension:<bytes4>` key can contain an extension's address as its first 20 bytes, with `0x01` appended at the end to indicate that `msg.value` should be forwarded to the extension.

Impact

LSP0 accounts that are managed by LSP6 key managers will not be able to whitelist extensions that forward `msg.value`.

Recommended Mitigation

In `_getPermissionRequiredToSetDataKey()`, consider allowing input data to be 21 bytes long for LSP17 keys:

LSP6SetDataModule.sol#L322-L330

```
            // LSP17Extension:<bytes4>
        } else if (bytes12(inputDataKey) == _LSP17_EXTENSION_PREFIX) {
            // CHECK that `dataValue` contains exactly 20 bytes, which corresponds to an address for
a LSP17 Extension
-           if (inputDataValue.length != 20 && inputDataValue.length != 0) {
+           if (inputDataValue.length != 20 && inputDataValue.length != 21 && inputDataValue.length
!= 0) {
                revert InvalidDataValuesForDataKeys(
                    inputDataKey,
                    inputDataValue
                );
            }
```

Team Response

Fixed in PR #777.

M-03: `_extractCallType()` returns `0x00000000` as the required call type for empty calls

Bug Description

The `_extractCallType()` function is used to determine which allowed call permission is required:

LSP6ExecuteModule.sol#L309-L329

```
    function _extractCallType(
        uint256 operationType,
        uint256 value,
        bool isEmptyCall
    ) internal pure returns (bytes4 requiredCallTypes) {
        // if there is value being transferred, add the extra bit
        // for the first bit for Value Transfer in the `requiredCallTypes`
        if (value != 0) {
            requiredCallTypes |= _ALLOWEDCALLS_TRANSFERVALUE;
        }

        if (!isEmptyCall) {
            if (operationType == OPERATION_0_CALL) {
                requiredCallTypes |= _ALLOWEDCALLS_CALL;
            } else if (operationType == OPERATION_3_STATICCALL) {
                requiredCallTypes |= _ALLOWEDCALLS_STATICCALL;
            } else if (operationType == OPERATION_4_DELEGATECALL) {
                requiredCallTypes |= _ALLOWEDCALLS_DELEGATECALL;
            }
        }
    }
```

However, if `value = 0` and `isEmptyCall = true`, both if-statements will be skipped, causing `requiredCallTypes` to be `bytes4(0)`.

This becomes problematic as `_isAllowedCallType()` validates allowed call permissions using `&`:

LSP6ExecuteModule.sol#L394-L395

```
        bytes4 allowedCallType = bytes4(allowedCall);
        return (allowedCallType & requiredCallTypes == requiredCallTypes);
```

When `requiredCallTypes = bytes4(0)`, the check above will always pass, regardless of what `allowedCallType` actually is.

This allows the fallback functions of contracts to be called through the wrong permissions:

- Assume a contract has the following functions:
  - Some `view` functions.
  - A state-changing (non-view) function.
  - A fallback function.
- Assume that Bob has an LSP0 account, and wants to allow Alice to perform some calls on his behalf:
  - Alice is given the permissions `CALL` and `STATICCALL`.

- Bob wants to allow Alice to call all view functions in the contract through his LSP0 account. He whitelists the following allowed call:
    - `allowedCallType = _ALLOWEDCALLS_STATICCALL`
    - `allowedAddress` is set to the contract.
    - `allowedStandard = 0xffffffff`
    - `allowedFunction = 0xffffffff`
- Alice, however, is malicious. She wants to call the contract's fallback function through Bob's LSP0 account, even though it is not a whitelisted allowed call.
- She calls `execute()` with payload such that:
    - `operationType = OPERATION_0_CALL`
    - `to` is the contract
    - `data`, the underlying calldata, is empty
    - `msg.value = 0`
- In `_extractCallType()`, since `value = 0` and `isEmptyCall = true`, `bytes4(0)` is returned as the required call type.
- In `_verifyAllowedCall()`, `_isAllowedCallType()` incorrectly returns `true` when checking against the allowed call whitelisted above.
- Therefore, even though Bob never added the contract's fallback function to his allowed calls, Alice manages to call it.

Impact

Empty calls with no value will unintentionally be permitted through incorrect allowed calls. This can be used to call fallback functions even if they are not whitelisted.

Recommended Mitigation

Consider modifying `_extractCallType()` to account for empty calls when `value == 0`:

LSP6ExecuteModule.sol#L320-L328

```
-       if (!isEmptyCall) {
+       if (!isEmptyCall || (isEmptyCall && value == 0)) {
            if (operationType == OPERATION_0_CALL) {
                requiredCallTypes |= _ALLOWEDCALLS_CALL;
            } else if (operationType == OPERATION_3_STATICCALL) {
                requiredCallTypes |= _ALLOWEDCALLS_STATICCALL;
            } else if (operationType == OPERATION_4_DELEGATECALL) {
                requiredCallTypes |= _ALLOWEDCALLS_DELEGATECALL;
            }
        }
```

Team Response

Fixed in PR #776.

M-04: Missing universal receiver callback in `renounceOwnership()` when called by the owner

Bug Description

When `LSP14Ownable2Step::renounceOwnership()` is called for a second time to renounce ownership, the owner receives a universal receiver callback:

LSP14Ownable2Step.sol#L144-L149

```
        if (owner() == address(0)) {
            previousOwner.tryNotifyUniversalReceiver(
                _TYPEID_LSP14_OwnershipTransferred_SenderNotification,
                ""
            );
        }
```

However, in `LSP0ERC725AccountCore::renounceOwnership()`, there is no such callback when renouncing ownership if `msg.sender == owner()`:

LSP0ERC725AccountCore.sol#L632-L642

```
    function renounceOwnership()
        public
        virtual
        override(LSP14Ownable2Step, OwnableUnset)
    {
        address accountOwner = owner();

        // If the caller is the owner perform renounceOwnership directly
        if (msg.sender == accountOwner) {
            return LSP14Ownable2Step._renounceOwnership();
        }
```

Impact

If `renounceOwnership()` is called by the LSP0 account's owner, he will not receive a universal receiver callback when completing the process.

Recommended Mitigation

Consider refactoring `renounceOwnership()` to include this universal receiver callback. For example:

```
    function renounceOwnership()
        public
        virtual
        override(LSP14Ownable2Step, OwnableUnset)
    {
        address accountOwner = owner();

        bool verifyAfter;

        if (msg.sender != accountOwner) {
            // If the caller is not the owner, call {lsp20VerifyCall} on the owner
            // Depending on the returnedStatus, a second call is done after renouncing ownership
            verifyAfter = _verifyCall(accountOwner);
        }

        LSP14Ownable2Step._renounceOwnership();

        if (owner() == address(0)) {
            accountOwner.tryNotifyUniversalReceiver(
                _TYPEID_LSP0_OwnershipTransferred_SenderNotification,
                ""
            );
        }

        // If verifyAfter is true, Call {lsp20VerifyCallResult} on the owner
        // The renounceOwnership function does not return, second parameter of {_verifyCallResult}
will be empty
        if (verifyAfter) {
            _verifyCallResult(accountOwner, "");
        }
    }
```

Team Response

Fixed in PR #783.

Low Severity Findings

L-01: Return data from `execute()` and `executeRelayCall()` is corrupted when calling `LSP0ERC725AccountCore::executeBatch()`

Bug Description

In `_executePayload()`, the return data from calling the target contract is decoded into bytes:

LSP6KeyManagerCore.sol#L509-L519

```
        (bool success, bytes memory returnData) = targetContract.call{
            value: msgValue,
            gas: gasleft()
        }(payload);
        bytes memory result = Address.verifyCallResult(
            success,
            returnData,
            "LSP6: failed executing payload"
        );

        return result.length != 0 ? abi.decode(result, (bytes)) : result;
```

This works for all functions in `LSP0ERC725AccountCore.sol` except `executeBatch()`, which returns a `bytes[]`:

LSP0ERC725AccountCore.sol#L256-L261

```
    function executeBatch(
        uint256[] memory operationsType,
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory datas
    ) public payable virtual override returns (bytes[] memory) {
```

Impact

If `executeBatch()` is called through a LSP6 key manager's functions, such as `execute()` or `executeRelayCall()`, the returned data will be corrupted since it decodes a `bytes[]` into `bytes`.

Recommended Mitigation

In `_executePayload()`, consider directly returning result instead of attempting to decode it:

LSP6KeyManagerCore.sol#L509-L519

```
        (bool success, bytes memory returnData) = targetContract.call{
            value: msgValue,
            gas: gasleft()
        }(payload);
        bytes memory result = Address.verifyCallResult(
            success,
            returnData,
            "LSP6: failed executing payload"
        );

-       return result.length != 0 ? abi.decode(result, (bytes)) : result;
+       return result;
```

By doing so, it becomes the caller's responsibility to decode the return data into the appropriate data type.

This is because it is impossible to know during runtime what data type is returned from the called function, especially if the LSP6 key manager is used for a `targetContract` that isn't a LSP0 account.

Team Response

Fixed in PR #784.

L-02: Functions with a `0x00000000` selector cannot be added to allowed calls

Bug Description

In `_isAllowedFunction()`, the `0x00000000` function selector is used as a special value to represent calldata with less than 4 bytes:

LSP6ExecuteModule.sol#L375-L380

```
        bool isFunctionCall = requiredFunction != bytes4(0);

        // ANY function = 0xffffffff
        return
            allowedFunction == bytes4(type(uint32).max) ||
            (isFunctionCall && (requiredFunction == allowedFunction));
```

As seen from above, if `isFunctionCall` is false, `_isAllowedFunction()` will return `false` unless the allowed call allows any function.

However, if a function's selector happens to clash with `0x00000000`, the only way to call it through allowed calls is by listing `0xffffffff` as the function selector, since `0x00000000` isn't allowed.

Recommended Mitigation

Consider refactoring `_isAllowedFunction()` to allow the `0x00000000` selector.

This can be done by passing the calldata as a parameter to `_isAllowedFunction()`, and changing `isFunctionCall` to `data.length >= 4` instead.

Team Response

Fixed in PR #776.

Informational Findings

I-01: Unnecessary initialization to 0 in for-loop

LSP6KeyManagerCore.sol#L615

```
-           for (uint256 ii = 0; ii < operationTypes.length; ii++) {
+           for (uint256 ii; ii < operationTypes.length; ii++) {
```

Team Response

Fixed in PR #782.

I-02: `_isValidNonce()` can be simplified

Instead of using a bitmask, consider casting `idx` to `uint128` to retrieve the nonce:

LSP25MultiChannelNonce.sol#L138-L147

```
    function _isValidNonce(
        address from,
        uint256 idx
    ) internal view virtual returns (bool) {
-       uint256 mask = ~uint128(0);
-       // Alternatively:
-       // uint256 mask = (1<<128)-1;
-       // uint256 mask = 0xffffffffffffffffffffffffffffffff;
-       return (idx & mask) == _nonceStore[from][idx >> 128];
+       return uint128(idx) == _nonceStore[from][idx >> 128];
    }
```

Team Response

Fixed in PR #782.