MiloTruck

LUKSO

Security Review Report

July, 2023

Table of Contents

Introduction

About MiloTruck

MiloTruck is an independent security researcher who specializes in smart contract audits. Having won multiple audit contests, he is currently one of the top wardens on Code4rena.

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

Disclaimer

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

Executive Summary

This review was completed as part of an audit contest on Code4rena.

About LUKSO

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

Repository Details

Repository	https://github.com/code-423n4/2023-06-lukso
Commit Hash	09bbdd68eeba6ed4dd624286c94a1947f79c195
Language	Solidity

Scope

The scope of this review can be found here.

Issues Found

Severity	Count
High	0
Medium	6
Low	13
Non-Critical	11

Findings

Summary

ID	Description	Severity
M-01	The owner of a LSP0ERC725Account can become the owner again after renouncing ownership	Medium
M-02	Two-step ownership transfer process in LSP0ERC725AccountCore can be bypassed	Medium
M-03	LSP8 and LSP9's ERC-165 interface ID differs from their specification	Medium
M-04	LSP8Burnable extension inherits the wrong contract	Medium
M-05	LSP8CompatibleERC721's approve() function deviates from ERC-721 specification	Medium
M-06	combinePermissions() handles duplicate permissions incorrectly	Medium
L-01	Users with ADDEXTENSIONS/CHANGEEXTENSIONS permissions can indirectly allow anyone to re-enter KeyManager	Low
L-02	Misleading comment in LSP7DigitalAssetCore.sol	Low
L-03	_fallbackLSP17Extendable() doesn't forward msg.value	Low
L-04	Consider validating _beforeTokenTransfer() in LSP8IdentifiableDigitalAssetCore.sol	Low
L-05	Incorrect NatSpec @dev in LSP6Utils.sol	Low
L-06	isEncodedArray() could revert or return true incorrectly	Low
L-07	Extensions should ensure that msg.value is 0 to prevent user mistakes	Low
L-08	Data passed to lsp20VerifyCallResult() is different in execute() and executeBatch()	Low
L-09	Extensions with a 0x00000000 function selector could create phantom functions	Low
L-10	LSP-17 extensions in LSP0 accounts are vulnerable to metamorphic contracts	Low
L-11	Allowed calls in LSP6ExecuteModule isn't compatible with some function selectors	Low

L-12	`LSP7DigitalAssetCore`'s `_burn()` function shouldn't have an allowance check	Low
L-13	`_getPermissionToSetControllerPermissions()` might return incorrect permissions for malformed data	Low
N-01	Import statement in `LSP0ERC725AccountCore.sol` can be more succinct	Non-Critical
N-02	Typos	Non-Critical
N-03	Unreachable if-statement in `generateSentVaultKeys()` can be removed	Non-Critical
N-04	Code in `_whenReceiving()` and `_whenSending()` can be shorter	Non-Critical
N-05	Unnecessary use of ternary operator	Non-Critical
N-06	Code in `lsp20VerifyCall()` can be more succinct	Non-Critical
N-07	Use `type(uint128).max` instead of `~uint128(0)`	Non-Critical
N-08	`_setupLSP6ReentrancyGuard()` is redundant	Non-Critical
N-09	Missing check in `_deployCreate2()`	Non-Critical
N-10	Document `_INTERFACEID_LSP20_CALL_VERIFICATION` in the LSP-20 specification	Non-Critical
N-11	`LSP8CompatibleERC721.sol` doesn't have a `_safeMint()` function	Non-Critical

Medium Severity Findings

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

Bug Description

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()`, perform a delegate call that overwrites `_pendingOwner` to his own address.
3. Call `renounceOwnership()` again to set the owner to `address(0)`.

As `_pendingOwner` is still set to the owner's address, he can call `acceptOwnership()` at any time 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

[Link to PoC]()

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();
```

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

Bug Description

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 directly
        if (msg.sender == currentOwner) {
            // set the pending owner
            LSP14Ownable2Step._transferOwnership(pendingNewOwner);
            emit OwnershipTransferStarted(currentOwner, pendingNewOwner);

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

            // Require that the owner didn't change after the LSP1 Call
            // (Pending owner didn't automate the acceptOwnership call through LSP1)
            require(
                currentOwner == owner(),
                "LSP14: newOwner MUST accept ownership in a separate transaction"
            );
        } 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 he wants, such as transferring the account's entire LYX balance to himself.
- 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

[Link to PoC]

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 CannotAcceptOwnershipDuringTransfer();

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

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

Bug Description

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.

M-04: `LSP8Burnable` extension inherits the wrong contract

Bug Description

The `LSP8Burnable` contract inherits from `LSP8IdentifiableDigitalAssetCore`:

LSP8Burnable.sol#L15

```
abstract contract LSP8Burnable is LSP8IdentifiableDigitalAssetCore {
```

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 LSP8IdentifiableDigitalAsset {
```

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

Impact

As `LSP8Burnable` does not inherit `LSP8IdentifiableDigitalAsset`, a developer who implements his 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 LSP8IdentifiableDigitalAssetCore {
+   abstract contract LSP8Burnable is LSP8IdentifiableDigitalAsset {
```

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

M-05: `LSP8CompatibleERC721`'s `approve()` function deviates from ERC-721 specification

Bug Description

The `LSP8CompatibleERC721` contract is a wrapper around LSP8 that 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 virtual {
        authorizeOperator(operator, bytes32(tokenId));
        emit Approval(tokenOwnerOf(bytes32(tokenId)), operator, tokenId);
    }
```

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.sender);
        }
```

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 NFT
    /// @dev The zero address indicates there is no approved address.
    ///  Throws unless `msg.sender` is the current NFT owner, or an authorized
    ///  operator of the current owner.
    /// @param _approved The new approved NFT controller
    /// @param _tokenId The NFT to approve
    function approve(address _approved, uint256 _tokenId) external payable;
```

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

ERC721.sol#L121-L123

```
        if (_msgSender() != owner && !isApprovedForAll(owner, _msgSender())) {
            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 virtual {
    bytes32 tokenIdBytes = bytes32(tokenId);
    address tokenOwner = tokenOwnerOf(tokenIdBytes);

    if (tokenOwner != msg.sender && !isApprovedForAll(tokenOwner, msg.sender)) {
        revert LSP8NotTokenOwner(tokenOwner, tokenIdBytes, msg.sender);
    }

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

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

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

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

M-06: `combinePermissions()` handles duplicate permissions incorrectly

Bug Description

In `LSP6Utils.sol`, `combinePermissions()` is a helper function for combining multiple permissions into a single `bytes32`:

LSP6Utils.sol#L169-L177

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

However, as it uses addition to combine the permissions, the result will be incorrect when the `permissions` array contains two or more of the same permissions.

For example, if `combinePermissions()` is called with an array containing two `CHANGEOWNER` (`0x1`) permissions, it will return `0x2`, which is the `ADDCONTROLLER` permission.

Impact

Contracts could end up assigning users with wrong permissions when using `combinePermissions()`.

Recommended Mitigation

Consider using bitwise OR to combine permissions instead:

LSP6Utils.sol#L169-L177

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

Low Severity Findings

L-01: Users with `ADDEXTENSIONS`/`CHANGEEXTENSIONS` permissions can indirectly allow anyone to re-enter KeyManager

Bug Description

`LSP6KeyManager` has in-built reentrancy protections. It works by setting `_reentrancyStatus` to `true` using `_nonReentrantBefore()` before a call to `LSP0ERC725Account` is made, and then calling `_nonReentrantAfter()` to set `_reentrancyStatus` back to `false` afterwards.

However, this reentrancy protection can be bypassed by adding the `LSP6KeyManager`'s `lsp20VerifyCallResult()` function as an extension to the LSP0 account:

LSP6KeyManagerCore.sol#L303-L313

```
    function lsp20VerifyCallResult(
        bytes32 /*callHash*/,
        bytes memory /*result*/
    ) external returns (bytes4) {
        // If it's the target calling, set back the reentrancy guard
        // to false, if not return the magic value
        if (msg.sender == _target) {
            _nonReentrantAfter();
        }
        return _LSP20_VERIFY_CALL_RESULT_MAGIC_VALUE;
    }
```

Where:

- `_target` is set to the `LSP0ERC725Account`'s address.

As `lsp20VerifyCallResult()` is called through the LSP0 account, `msg.sender == _target` will be true, thereby calling `_nonReentrantAfter()` to reset `_reentrancyStatus`.

Therefore, after `lsp20VerifyCallResult()` is added as an extension, anyone can then call this before performing a reentrant call to `LSP6KeyManager` to bypass reentrancy checks.

Impact

Anyone with `ADDEXTENSIONS`/`CHANGEEXTENSIONS` permissions can allow users to bypass `LSP6KeyManager`'s reentrancy protections by adding `lsp20VerifyCallResult()` as an extension.

Recommended Mitigation

Disallow the `lsp20VerifyCallResult()` function in `LSP6KeyManager`. One way of achieving this could be to blacklist its function selector. However, this could potentially cause problems if a function in another extension happens to have the same selector.

L-02: Misleading comment in `LSP7DigitalAssetCore.sol`

Bug Description

The following comment states that ERC20's approval race condition issue can be avoided by calling LSP7's `revokeOperator()` before `authorizeOperator()` to update a spender's allowance:

LSP7DigitalAssetCore.sol#L78-L90

```
    /**
     * @inheritdoc ILSP7DigitalAsset
     *
     * @dev To avoid front-running and Allowance Double-Spend Exploit when
     * increasing or decreasing the authorized amount of an operator,
     * it is advised to:
     *     1. call {revokeOperator} first, and
     *     2. then re-call {authorizeOperator} with the new amount
     *
     * for more information, see:
     * https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/
     *
     */
```

This is incorrect as `revokeOperator()` simply sets the spender's allowance back to 0:

LSP7DigitalAssetCore.sol#L101-L103

```
    function revokeOperator(address operator) public virtual {
        _updateOperator(msg.sender, operator, 0);
    }
```

Therefore, someone could still spend double of his allowance by front-running the call to `revokeOperator()`.

Recommended Mitigation

Change the comment to recommend using `decreaseAllowance()` instead.

L-03: `_fallbackLSP17Extendable()` doesn't forward `msg.value`

Bug Description

In `LSP17Extendable.sol`, `_fallbackLSP17Extendable()` is used in fallback functions to forward calls to their respective extensions, similar to how proxies work. However, it uses a low-level call instead of `delegatecall`:

LSP17Extendable.sol#L108-L116

```
           let success := call(
            gas(),
            extension,
            0, // value is set to 0
            0,
            add(calldatasize(), 52),
            0,
            0
          )
```

As seen from above, none of `msg.value` will be forwarded to the extension contract in the low-level call.

Impact

This might cause several problems:

1. The functionality of LSP-17 is limited; developers might want to receive native tokens in their extension contracts but will be unable to do so.
2. This behavior isn't mentioned anywhere in the documentation or LSP specification. Developers might expect LSP-17 to behave similarly to proxies and write extensions that require native tokens, causing their code to be incorrect.

Recommended Mitigation

Consider mentioning that no native tokens will be forwarded to the extension using the LSP-17 standard.

L-04: Consider validating `_beforeTokenTransfer()` in `LSP8IdentifiableDigitalAssetCore.sol`

In `LSP8IdentifiableDigitalAssetCore.sol`, consider adding the following checks after various `_beforeTokenTransfer` hooks:

- `_mint()` - Check that `tokenId` was not minted in `_beforeTokenTransfer`:

LSP8IdentifiableDigitalAssetCore.sol#L329

```
        _beforeTokenTransfer(address(0), to, tokenId);

+       // Check that `tokenId` was not minted by `_beforeTokenTransfer` hook
+       if (_exists(tokenId)) {
+           revert LSP8TokenIdAlreadyMinted(tokenId);
+       }
```

- `_burn()` - Update `tokenOwner` if `tokenId` was transferred in `_beforeTokenTransfer`:

LSP8IdentifiableDigitalAssetCore.sol#L363

```
        _beforeTokenTransfer(tokenOwner, address(0), tokenId);

+       // Update `tokenOwner` in case `tokenId` was transferred
+       tokenOwner = tokenOwnerOf(tokenId);
```

- `_transfer()` - Check that `tokenId` was not transferred to a new owner in `_beforeTokenTransfer`:

LSP8IdentifiableDigitalAssetCore.sol#L416

```
        _beforeTokenTransfer(from, to, tokenId);

+       // Check that `tokenId` was not transferred by `_beforeTokenTransfer` hook
+       if (tokenOwner != tokenOwnerOf(tokenId)) {
+           revert ERC721IncorrectOwner(from, tokenId, owner);
+       }
```

If a contract inheriting `LSP8IdentifiableDigitalAssetCore` has a bug in its `_beforeTokenTransfer` hook (eg. reentrancy), these checks will prevent tokens from being duplicated.

L-05: Incorrect NatSpec `@dev` in `LSP6Utils.sol`

Bug Description

The following comment states each element in the array passed to `isCompactBytesArrayOfAllowedCalls()` must be 28 bytes long:

[LSP6Utils.sol#L85-L86]()

```
    /*
     * @dev same as LSP2Utils.isCompactBytesArray with the additional requirement that
each element must be 28 bytes long.
```

However, `isCompactBytesArrayOfAllowedCalls()` actually validates that each element is 32 bytes long:

[LSP6Utils.sol#L106-L107]()

```
            // each entries in the allowedCalls (compact) array must be 32 bytes long
            if (elementLength != 32) return false;
```

Impact

This might mislead developers who only read the NatSpec, causing them to utilize the `isCompactBytesArrayOfAllowedCalls()` function incorrectly.

L-06: `isEncodedArray()` could revert or return `true` incorrectly

Bug Description

In `LSP2Utils.sol`, the `isEncodedArray()` function is used to check if `data` is an encoded array. However, its implementations has several issues:

1. If `offset` or `arrayLength` is too large, the function might revert due to an arithmetic overflow in the following lines:

LSP2Utils.sol#L236

```
        if (nbOfBytes < offset + 32) return false;
```

LSP2Utils.sol#L242

```
        if (nbOfBytes < (offset + 32 + (arrayLength * 32))) return false;
```

2. As there is no `offset >= 32` check, the function considers `bytes32(0)` as a valid encoded array.

3. The function only ensures that `data.length` is not smaller than the length calculated with `offset` and `arrayLength`. Therefore, even if `data` contains more bytes than the correct length, `isEncodedArray()` will still return `true`.

Impact

Should a developer inherit the `LSP2Utils` contract and use `isEncodedArray()` to validate user input, an attacker could intentionally craft malformed data to bypass `isEncodedArray()` or cause it to revert.

L-07: Extensions should ensure that `msg.value` is 0 to prevent user mistakes

Bug Description

The fallback function of the `LSP0ERC725AccountCore` contract is declared as `payable`:

LSP0ERC725AccountCore.sol#L151-L161

```
    fallback() external payable virtual {
        if (msg.value != 0) {
            emit ValueReceived(msg.sender, msg.value);
        }

        if (msg.data.length < 4) {
            return;
        }

        _fallbackLSP17Extendable();
    }
```

This is meant to support LSP17 extensions that require transfers of LYX. However, this allows users to accidentally transfer LYX to the fallback function while calling an extension that doesn't use `msg.value`.

Recommended Mitigation

In the documentation, state that developers should ensure `msg.value == 0` if their extensions do not use `msg.value`.

L-08: Data passed to `lsp20VerifyCallResult()` is different in `execute()` and `executeBatch()`

Bug Description

In `execute()`, the result from `ERC725XCore._execute()` is abi-encoded as `bytes` and passed to `_verifyCallResult()`, which calls the owner's `lsp20VerifyCallResult()` function:

LSP0ERC725AccountCore.sol#L243-L254

```
        // Perform the execution
        bytes memory result = ERC725XCore._execute(
            operationType,
            target,
            value,
            data
        );

        // if verifyAfter is true, Call {lsp20VerifyCallResult} on the owner
        if (verifyAfter) {
            LSP20CallVerification._verifyCallResult(_owner, abi.encode(result));
        }
```

However, in `executeBatch()`, the data is abi-encoded as a `bytes` array instead:

LSP0ERC725AccountCore.sol#L307-L321

```
        // Perform the execution
        bytes[] memory results = ERC725XCore._executeBatch(
            operationsType,
            targets,
            values,
            datas
        );

        // if verifyAfter is true, Call {lsp20VerifyCallResult} on the owner
        if (verifyAfter) {
            LSP20CallVerification._verifyCallResult(
                _owner,
                abi.encode(results)
            );
        }
```

Impact

This creates two issues:

1. The owner's `lsp20VerifyCallResult()` has no way to differentiate between a `bytes` result from `execute()` and a `bytes` array from `executeBatch()`.
2. When calling `execute()`, a malicious attacker could potentially manipulate `ERC725XCore._execute()` into returning `bytes` that resembles a `bytes` array in data, which would trick `lsp20VerifyCallResult()` into thinking the result came from `executeBatch()`.

Recommended Mitigation

Consider calling `_verifyCallResult()` in a loop with each result in the `results` array individually:

[LSP0ERC725AccountCore.sol#L316-L321]()

```diff
        // if verifyAfter is true, Call {lsp20VerifyCallResult} on the owner
        if (verifyAfter) {
-           LSP20CallVerification._verifyCallResult(
-               _owner,
-               abi.encode(results)
-           );
+           for (uint256 i; i < results.length; ++i) {
+               LSP20CallVerification._verifyCallResult(
+                   _owner,
+                   abi.encode(results[i])
+               );
+           }
        }
```

L-09: Extensions with a `0x00000000` function selector could create phantom functions

Bug Description

In the `LSP0ERC725AccountCore` contract, calls to the fallback function should revert if no matching function selector is found. However, the `0x00000000` function selector simply returns instead:

LSP0ERC725AccountCore.sol#L800-L801

```
        // if no extension was found for bytes4(0) return don't revert
        if (msg.sig == bytes4(0) && extension == address(0)) return;
```

Impact

This could create phantom functions for functions in extensions with the `0x00000000` selector.

For example, a contract might perform some kind of validation in an extension, such as checking for LSP6 permissions, and expect the function to revert if the user is not authorized.

However, if the function's selector is `0x00000000` and the LSP0 account doesn't have this extension, the contract's call to the function will return instead of reverting, giving the contract the impression that the user is authorized.

Recommended Mitigation

Consider treating reverting for the `0x00000000` function selector as well when no extension matches. Otherwise, warn developers that they should not rely on extensions to revert, but check its return value instead

L-10: LSP-17 extensions in LSP0 accounts are vulnerable to metamorphic contracts

Bug Description

To add an extension to a LSP0 account, the owner has to add its address to the list of extensions, similar to a whitelist. However, this pattern is vulnerable to metamorphic contracts. For example:

- Attacker deploys an extension contract with `CREATE2` to a predetermined address.
  - This extension contract is not able to do anything malicious, but has the ability to `selfdestruct` itself.
- As the extension looks harmless, the owner of an LSP0 account adds it.
- The attacker selfdestructs the contract and deploys a new one with different runtime bytecode using `CREATE2`.
  - As long as the same `salt` and initialization code was provided to `CREATE2`, this new contract will have the same address as the destructed extension contract.
- The attacker can now use this contract to perform malicious actions on the LSP0 account.

This attack was famously used in the Tornado Cash Governance Hack.

Recommended Mitigation

Warn users about the risk of adding extensions with the ability to `selfdestruct`.

L-11: Allowed calls in `LSP6ExecuteModule` isn't compatible with some function selectors

Bug Description

Whenever a controller attempts to call a LSP0 account's `execute()` function without the relevant SUPER permissions, `LSP6ExecuteModule` will check that the call is one of the whitelisted allowed calls.

However, in `_isAllowedFunction()`, the function selectors `0x00000000` and `0xffffffff` have special meanings:

LSP6ExecuteModule.sol#L410-L415

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

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

`0x00000000` represents a call with empty calldata, while `0xffffffff` means that all function selectors are permitted. This makes `_isAllowedFunction()` unable to handle functions that have either of these selectors.

Impact

Functions with either of these selectors cannot be called by users with only `SETDATA` permissions.

L-12: `LSP7DigitalAssetCore`'s `_burn()` function shouldn't have an allowance check

Bug Description

In `LSP7DigitalAssetCore`, the `_burn()` function checks that the caller has a sufficient allowance from the `from` address:

[LSP7DigitalAssetCore.sol#L352-L366]

```
        address operator = msg.sender;
        if (operator != from) {
            uint256 authorizedAmount = _operatorAuthorizedAmount[from][
                operator
            ];
            if (amount > authorizedAmount) {
                revert LSP7AmountExceedsAuthorizedAmount(
                    from,
                    authorizedAmount,
                    operator,
                    amount
                );
            }
            _operatorAuthorizedAmount[from][operator] -= amount;
        }
```

However, this check could limit the functionality of contracts that inherit from `LSP7DigitalAsset`. Some protocols might want to allow other addresses, such as a contract trusted by the protocol, to burn the LSP7 tokens of the user.

Furthermore, this is inconsistent with LSP8's `_burn()` function, which does not contain any checks for the allowance of `msg.sender`.

Recommended Mitigation

Consider removing the allowance check in `_burn()` and adding it to the [`LSP8Burnable`] extension instead.

L-13: `_getPermissionToSetControllerPermissions()` might return incorrect permissions for malformed data

Bug Description

The `_getPermissionToSetControllerPermissions()` function is used to check if the `ADDCONTROLLER` or `EDITPERMISSIONS` permission is required when adding a controller using `setData()` or `setDataBatch()`:

LSP6SetDataModule.sol#L385-L397

```
    function _getPermissionToSetControllerPermissions(
        address controlledContract,
        bytes32 inputPermissionDataKey
    ) internal view virtual returns (bytes32) {
        return
            // if there is nothing stored under the data key, we are trying to ADD a new
controller.
            // if there are already some permissions set under the data key, we are
trying to CHANGE the permissions of a controller.
            bytes32(
                ERC725Y(controlledContract).getData(inputPermissionDataKey)
            ) == bytes32(0)
                ? _PERMISSION_ADDCONTROLLER
                : _PERMISSION_EDITPERMISSIONS;
    }
```

However, instead of checking if the data's length is 0, it checks if the data under the `inputPermissionDataKey` key is empty by comparing it to `bytes32(0)`. This will return `true` as long as the first 32 bytes are `0x00`, regardless of whether there is data after the first 32 bytes.

This could potentially cause problems if a contract sets permissions to `bytes32(0)` temporarily to ensure that users with `EDITPERMISSIONS` are still able to edit them.

Recommended Mitigation

Check if the data is empty using by comparing its length to 0 instead:

LSP6SetDataModule.sol#L392-L396

```
-           bytes32(
-               ERC725Y(controlledContract).getData(inputPermissionDataKey)
-           ) == bytes32(0)
+           ERC725Y(controlledContract).getData(inputPermissionDataKey).length == 0
                ? _PERMISSION_ADDCONTROLLER
                : _PERMISSION_EDITPERMISSIONS;
```

Non-Critical Findings

N-01: Import statement in `LSP0ERC725AccountCore.sol` can be more succinct

Consider modifying the following import statement as such:

LSP0ERC725AccountCore.sol#L35-L43

```
    import {
        _INTERFACEID_LSP0,
        _INTERFACEID_ERC1271,
        _ERC1271_MAGICVALUE,
        _ERC1271_FAILVALUE,
        _TYPEID_LSP0_OwnershipTransferStarted,
        _TYPEID_LSP0_OwnershipTransferred_SenderNotification,
        _TYPEID_LSP0_OwnershipTransferred_RecipientNotification
-   } from "../LSP0ERC725Account/LSP0Constants.sol";
+   } from "./LSP0Constants.sol";
```

N-02: Typos

"loose" should be "lose":

LSP0ERC725AccountCore.sol#L616

```
     * - the current {`owner()`} will loose access to the functions restricted to the
{`owner()`} only.
```

Missing ")" in the comments below:

LSP5Utils.sol#L136

```
        // Updating the number of the received assets (decrementing by 1
```

LSP10Utils.sol#L139

```
        // Updating the number of the received vaults (decrementing by 1
```

"substractedAmount" should be "subtractedAmount":

LSP7DigitalAssetCore.sol#L234

```
        uint256 substractedAmount
```

This occurs on LSP7DigitalAssetCore.sol#L237 and LSP7DigitalAssetCore.sol#L245 as well.

N-03: Unreachable if-statement in `generateSentVaultKeys()` can be removed

The if-statement below is unreachable as `oldArrayLength` is a `uint128` and will never be larger than `type(uint128).max`:

LSP10Utils.sol#L132-L137

```
        // Updating the number of the received vaults
        uint128 oldArrayLength = uint128(bytes16(lsp10VaultsCountValue));

        if (oldArrayLength > type(uint128).max) {
            revert VaultIndexSuperiorToUint128(oldArrayLength);
        }
```

N-04: Code in `_whenReceiving()` and `_whenSending()` can be shorter

The `_whenReceiving()` function is implemented as such:

```
    function _whenReceiving(
        ...
    ) internal virtual returns (bytes memory) {
        bytes32[] memory dataKeys;
        bytes[] memory dataValues;

        // if it's a token transfer (LSP7/LSP8)
        if (typeId != _TYPEID_LSP9_OwnershipTransferred_RecipientNotification) {
            // Some code here...
            return "";
        } else {
            // Some code here...
            return "";
        }
    }
```

The code can be shortened by using one return statement at the end of the function:

```
        // if it's a token transfer (LSP7/LSP8)
        if (typeId != _TYPEID_LSP9_OwnershipTransferred_RecipientNotification) {
            // Some code here...
-           return "";
        } else {
            // Some code here...
-           return "";
        }
+       return "";
```

This also applies to `_whenSending()`, which follows the same pattern.

N-05: Unnecessary use of ternary operator

In `_verifyCall()`, return `bytes1(magicValue[3]) == 0x01` instead of using a ternary operator with `true` and `false`:

LSP20CallVerification/LSP20CallVerification.sol#L42

```
-       return bytes1(magicValue[3]) == 0x01 ? true : false;
+       return bytes1(magicValue[3]) == 0x01;
```

In `getTransferDetails()`, set `isReceiving` to `typeId == CONSTANT` directly:

LSP1Utils.sol#L84-L107

```
        if (
            typeId == _TYPEID_LSP7_TOKENSSENDER ||
            typeId == _TYPEID_LSP7_TOKENSRECIPIENT
        ) {
            mapPrefix = _LSP5_RECEIVED_ASSETS_MAP_KEY_PREFIX;
            interfaceId = _INTERFACEID_LSP7;
-           isReceiving = typeId == _TYPEID_LSP7_TOKENSRECIPIENT ? true : false;
+           isReceiving = typeId == _TYPEID_LSP7_TOKENSRECIPIENT;
        } else if (
            typeId == _TYPEID_LSP8_TOKENSSENDER ||
            typeId == _TYPEID_LSP8_TOKENSRECIPIENT
        ) {
            mapPrefix = _LSP5_RECEIVED_ASSETS_MAP_KEY_PREFIX;
            interfaceId = _INTERFACEID_LSP8;
-           isReceiving = typeId == _TYPEID_LSP8_TOKENSRECIPIENT ? true : false;
+           isReceiving = typeId == _TYPEID_LSP8_TOKENSRECIPIENT;
        } else if (
            typeId == _TYPEID_LSP9_OwnershipTransferred_SenderNotification ||
            typeId == _TYPEID_LSP9_OwnershipTransferred_RecipientNotification
        ) {
            mapPrefix = _LSP10_VAULTS_MAP_KEY_PREFIX;
            interfaceId = _INTERFACEID_LSP9;
-           isReceiving = (typeId ==
-               _TYPEID_LSP9_OwnershipTransferred_RecipientNotification)
-               ? true
-               : false;
+           isReceiving = typeId == _TYPEID_LSP9_OwnershipTransferred_RecipientNotification;
```

N-06: Code in `lsp20VerifyCall()` can be more succinct

The following code:

LSP6KeyManagerCore.sol#L256-L262

```
        bool isSetData = false;
        if (
            bytes4(data) == IERC725Y.setData.selector ||
            bytes4(data) == IERC725Y.setDataBatch.selector
        ) {
            isSetData = true;
        }
```

can be rewritten as a single expression:

```
bool isSetData = bytes4(data) == IERC725Y.setData.selector || bytes4(data) ==
IERC725Y.setDataBatch.selector;
```

N-07: Use `type(uint128).max` instead of `~uint128(0)`

Consider changing the code below to use `type(uint128).max`, which is more readable and easier to understand:

LSP6KeyManagerCore.sol#L445

```
        uint256 mask = ~uint128(0);
```

N-08: `_setupLSP6ReentrancyGuard()` is redundant

In `LSP6KeyManagerCore.sol`, `_setupLSP6ReentrancyGuard()` is used to initialize `_reentrancyStatus` to `false`:

LSP6KeyManagerCore.sol#L515-L520

```
    /**
     * @dev Initialise _reentrancyStatus to _NOT_ENTERED.
     */
    function _setupLSP6ReentrancyGuard() internal virtual {
        _reentrancyStatus = false;
    }
```

However, this is redundant as `_reentrancyStatus` is set to `false` by default.

N-09: Missing check in `_deployCreate2()`

In `ERC725XCore.sol`, `_deployCreate()` ensures that `address(this).balance` is more than or equal to the `value` parameter:

ERC725XCore.sol#L225-L227

```
        if (address(this).balance < value) {
            revert ERC725X_InsufficientBalance(address(this).balance, value);
        }
```

However, this check is missing in `_deployCreate2()`. Currently, this isn't exploitable as Openzeppelin's Create2 library has the following check:

Create2.sol#L31

```
        require(address(this).balance >= amount, "Create2: insufficient balance");
```

Nevertheless, consider adding the `address(this) < value` check to `_deployCreate2()` to have consistent error handling.

N-10: Document `_INTERFACEID_LSP20_CALL_VERIFICATION` in the LSP-20 specification

In `LSP20Constants.sol`, there are two ERC165 interface IDs:

LSP20Constants.sol#L4-L8

```
// bytes4(keccak256("LSP20CallVerification"))
bytes4 constant _INTERFACEID_LSP20_CALL_VERIFICATION = 0x1a0eb6a5;

// `lsp20VerifyCall(address,uint256,bytes)` selector XOR
`lsp20VerifyCallResult(bytes32,bytes)` selector
bytes4 constant _INTERFACEID_LSP20_CALL_VERIFIER = 0x480c0ec2;
```

However, in the LSP-20 specification, only `_INTERFACEID_LSP20_CALL_VERIFIER` is mentioned. This might be confusing for developers that wish to implement LSP-20, but are unable to figure out which interface ID to use.

Recommendation

Consider adding `_INTERFACEID_LSP20_CALL_VERIFICATION` to the specification and explaining when each interface ID should be used.

N-11: `LSP8CompatibleERC721.sol` doesn't have a `_safeMint()` function

`LSP8CompatibleERC721.sol` is meant to resemble the ERC721 standard. However, although it has a `safeTransferFrom()` function, its contract does not implement a `_safeMint()` function.

This might confuse developers who are used to popular implementations of ERC721 (eg. Openzeppelin's ERC721), which usually have an in-built `_safeMint()` function.