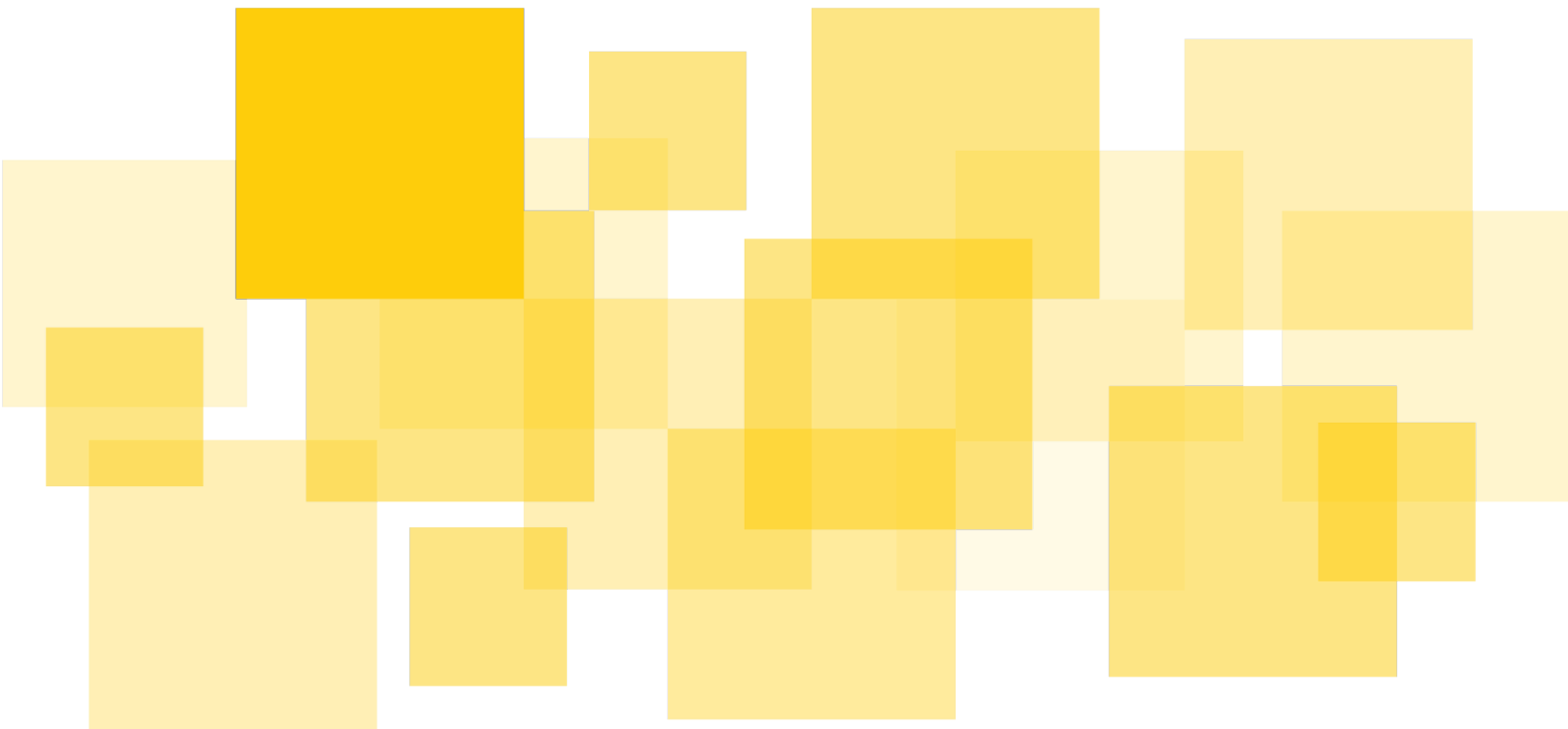


Formal Verification Report

LUKSO

Delivered: 2023-02-20



Prepared for LUKSO by Runtime Verification, Inc.





[Disclaimer](#)

[Introduction](#)

[Foundry+KEVM Workflow](#)

[Methodology](#)

[Properties](#)

[General Caveats](#)

[Uninitialized State](#)

[Maximal Permissions](#)

[Balance Transfers](#)

[Excluded Addresses](#)

[Fixed Bytes](#)

[Abstraction of Compact Bytes Array Checks](#)

[Test Descriptions](#)

[LSP0ERC725AccountTest.testCannotTransferOwnershipNotOwner](#)

[LSP0ERC725AccountTest.testCannotAcceptOwnershipNotOwner](#)

[LSP0ERC725AccountTest.testIsValidSignature](#)

[ERC725XTest.testCannotExecutelfNotOwner](#)

[ERC725YTest.testSetData1NotOwner](#)

[FallbackTest.testCallExtension](#)

[LSP6KeyManagerTest.testCannotTransferOwnership](#)

[LSP6KeyManagerTest.testCannotAcceptOwnership](#)

[LSP6KeyManagerTest.testCannotAddNewPermissions](#)

[LSP6KeyManagerTest.testCannotChangePermissions](#)

[LSP6KeyManagerTest.testCannotAddExtensions](#)

[LSP6KeyManagerTest.testCannotChangeExtensions](#)

[LSP6KeyManagerTest.testCannotAddUniversalReceiverDelegate](#)

[LSP6KeyManagerTest.testCannotChangeUniversalReceiverDelegate](#)

[LSP6KeyManagerTest.testCannotAddMappedDelegate](#)

[LSP6KeyManagerTest.testCannotChangeMappedDelegate](#)

[LSP6KeyManagerTest.testCannotAddNewAddress](#)

[LSP6KeyManagerTest.testCannotChangeAddress](#)

[LSP6KeyManagerTest.testCannotIncrementLength](#)

[LSP6KeyManagerTest.testCannotDecrementLength](#)

[LSP6KeyManagerTest.testCannotSetData](#)

[LSP6KeyManagerTest.testCannotTransferValue](#)

[LSP6KeyManagerTest.testCannotMakeCall](#)

[LSP6KeyManagerTest.testCannotMakeStaticCall](#)

[LSP6KeyManagerTest.testCannotDeployWithCreate](#)

[LSP6KeyManagerTest.testCannotDeployWithCreate2](#)

[LSP6KeyManagerTest.testCannotAddNewAllowedCalls](#)

[LSP6KeyManagerTest.testCannotChangeAllowedCalls](#)

[LSP6KeyManagerTest.testCannotAddNewAllowedDataKeys](#)

[LSP6KeyManagerTest.testCannotChangeAllowedDataKeys](#)

[LSP6KeyManagerTest.testCannotReenter](#)

[LSP6KeyManagerTest.testCannotSign](#)

[LSP6KeyManagerTest.testKeyManagerForwardsAllBalance](#)

[LSP6KeyManagerTest.testUniversalReceiverDelegateUP](#)

Verification Results

Findings

A01: supportsERC165InterfaceUnchecked is called without checking if address supports IERC165

Scenario

Recommendation

Status

A02: Compilation errors when using Solidity 0.8.13

Recommendation

Status

A03: LSP1UniversalReceiverDelegateUP calls target() on non-LSP6 accountOwner

Scenario

Recommendation

Status

A04: Payload is sliced without checking bounds first

Scenario

Recommendation

Status

A05: Empty call can be made via the key manager without any permissions

Recommendation

Status

A06: PERMISSION_CALL can be bypassed by constructing an adversarial payload

Details

Scenario



[Recommendation](#)

[Similar Findings](#)



Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.



Introduction

LUKSO has engaged Runtime Verification for a pilot project on using mechanized formal verification on the LUKSO code base using KEVM. The goal is to create Foundry fuzz tests for desirable properties of some core contracts, then use KEVM's Foundry integration to prove that these tests will pass under all circumstances. Normally, formal verification with KEVM requires building out a set of assumptions and theorems that allow the underlying K prover to automatically prove the desired specifications (or disprove them and find a bug). KEVM-Foundry is beta software which requires the engagement to both fine-tune the tooling as well as performing the normal proving tasks. Therefore, the engagement has been focused on both providing assumptions and theorems useful for the LUKSO codebase, as well as improving the KEVM-Foundry integration.

The targets for this engagement have been the LSP0ERC725Account, LSP6KeyManager and LSP1UniversalReceiverDelegateUP contracts. Although the code has gone through several iterations over the course of the engagement, the version on which KEVM was run was [version 0.8.0](#).

Foundry+KEVM Workflow

The KEVM-Foundry integration re-uses the property tests written for Foundry as formal specifications for KEVM proofs. The main advantage of this approach is that specifications can be written and read in Solidity and no further knowledge of a dedicated specification language is required to follow along. Another benefit of this method is that we can run the property tests before we try to prove them symbolically. This gives us a faster feedback cycle. The flowchart below visualizes the high-level workflow, which can be split into three stages.

Specification


The initial step of the workflow consists in writing property tests that capture the properties that we wish to verify. These tests assert that, for every test input that satisfies some assumptions, the results follow some expected behavior (revert with some error message, emit an event, return a value that satisfies some conditions, etc.). These tests serve as our formal specification.

Fuzzing

The first feedback cycle then consists in running the tests as fuzz tests using Foundry, by randomly-generating concrete inputs and executing the test over them. This allows us to quickly identify issues on the code or on the test itself, and either fix the code or refine the test (for example by adding missing assumptions). In this way, this step is important not only to find an initial set of bugs, but also to iterate on the formal specification and ensure that it accurately reflects the desired property.

Symbolic Execution

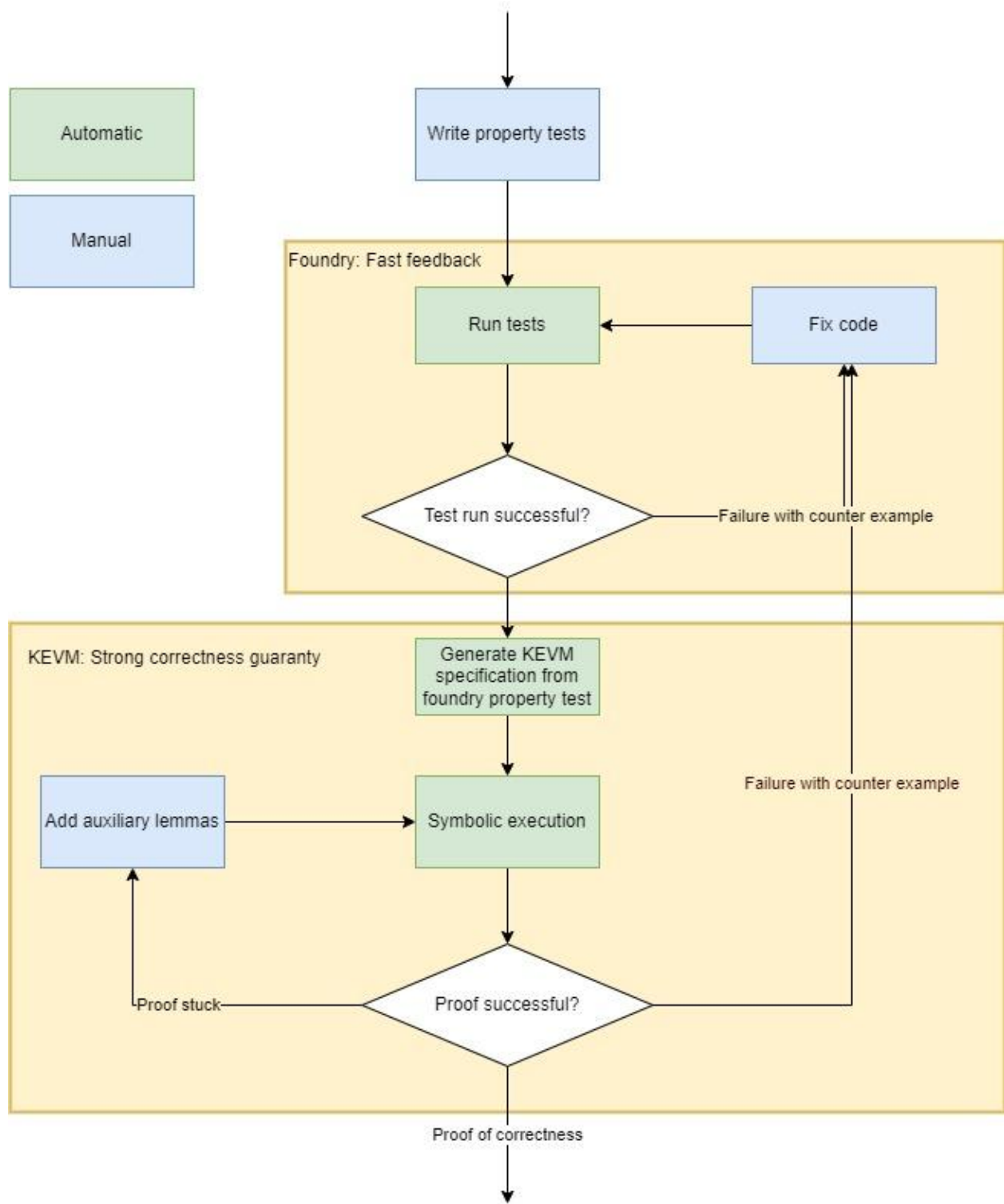
After all tests are passing, the process can move on to symbolic execution using KEVM. In contrast to fuzzing, symbolic execution uses symbolic rather than concrete inputs. This means that the input values are interpreted as mathematical variables that can contain any value that satisfies the assumptions. The symbolic execution engine then executes the code over these symbolic variables, building a mathematical expression capturing the set of possible values that they can take at each point in the execution. In this way, KEVM is able to explore the entire input space of the test without having to run it concretely for every input. At the end of the execution, the engine checks if every possible final state passes the test. If so, then we can state that the test passes for every possible input, and therefore the code has been formally verified to satisfy the specification. If there is a final state where the test fails, then this might indicate a bug in the code, in which case it needs to be fixed and the process repeated.



In some cases, however, KEVM might not be able to complete verification, but also not find an actual bug in the code. There are a few different scenarios that can happen:

- The engine reaches a final state where the test fails, but this state cannot be reached in practice. This happens because KEVM was not able to prove that this execution path is impossible.
- The engine gets stuck in a non-final state of the execution where it is unable to make progress. KEVM works by matching the current symbolic expression to a set of internal rules, and if it cannot find a rule that matches the current expression it cannot continue execution. This usually means that it was not able to simplify the expression to the form it needs.
- The symbolic execution branches too much, creating too many execution paths that the engine cannot complete in a reasonable time. Like before, this usually means that KEVM was not able to prove that some of these paths are impossible. In some cases, however, the branches are valid execution paths. If there are too many paths that need to be considered, it might be necessary to make simplifying assumptions to narrow the scope of the verification and focus on only a single, main path or small set of paths.

When the symbolic execution gets stuck or is unable to prune a spurious execution path, KEVM requires manual intervention in order to make progress. This is done by adding lemmas, auxiliary rules that can be fed into the tool to tell it how an expression can be simplified. KEVM ships with a body of lemmas, but new lemmas might need to be added for a specific proof. Any lemmas that we have written in the process of the verification are provided along with the tests.




Methodology

The first step of the verification process consisted in identifying important properties of the contracts in scope and formalizing these properties in the form of Foundry property tests. We identified many such properties, wrote tests for them, and ran these tests as fuzz tests in order to generate confidence that they hold. Running the tests also helped us identify necessary preconditions that must be assumed in order for the properties to hold, allowing us to refine the tests appropriately to reflect those.

As it would be infeasible to symbolically execute all the tests, we asked the LUKSO team for input on the most important properties that they would like verified. These are described in section [Properties](#) below. We wrote additional tests for those properties that were not already covered by the initial test suite and ensured that they passed fuzzing. We then attempted to formally verify them using KEVM. The results are shown in section [Verification Results](#). Due to limitations of our verification infrastructure, some of the tests required simplifications in order to make verification feasible (see [General Caveats](#) for more details).

Most of the Foundry tests written for these properties follow a similar structure:

1. Make explicit assumptions on the values of the test inputs using Foundry's `assume` cheatcode. These can include, for example, that two addresses are different, or that a value is not zero.
2. Initialize the relevant contracts. For the LSP0 tests, this is an `LSP0ERC725Account` with an arbitrary owner (provided as one of the test inputs). For the LSP6 tests, the ownership of the `LSP0ERC725Account` is given to an `LSP6KeyManager`, and the Foundry test contract is given all permissions on this key manager. This is done to allow the test contract to easily set up each test, as some of this setup requires specific permissions.
3. Set up the test. This involves making changes to the state of the account or key-manager contracts in order to simulate the conditions that the test is trying to exercise. For example:
 - In the LSP6 permissions tests, the controller is given a subset of the permissions.
 - In tests like `testCannotDecrementLength`, where the controller is trying to modify a previous value, the storage needs to be set to that initial value.
 - In tests that include a transfer of balance, the appropriate addresses need to be given a sufficient balance.
4. Call a Foundry cheatcode like `expectEmit`, `expectCall` or `expectRevert` to preempt some expected effect of the function call, for example emitting an event, making a call or reverting.
5. Impersonate the caller using the `prank` or `startPrank` cheatcodes.

- 
6. Call the function that is being tested.
 7. Make assertions about the expected result of the function.

Properties

We describe here the main properties that we have specified for the LSP contracts, along with the assumptions that we had to make in the process of formally verifying them.

We describe the properties and assumptions in this section in plain English to make them easy to understand, but this also means that the descriptions can be imprecise and ambiguous. If there are any doubts about the precise meaning of a property or assumption, the reader should refer to the Foundry tests, which serve as the formal specification.

The description of each test contains the following information:

- **Property:** A description in plain English of the property being tested and verified.
- **Preconditions:** A list of preconditions on the test parameters that must hold in order for the test to pass. Parameters that don't satisfy the preconditions fall outside the scope of the property being tested. For example, `testCannotDecrementLength` has the precondition that the initial length is non-zero (otherwise it could not be decremented).
- **Caveats:** A test might not always correspond 100% to the property that it is trying to express, either because a fully generic test for the property would be hard to specify in Foundry or because simplifications need to be made in order to make verification feasible. Such factors are listed as caveats.

General Caveats


These are caveats that apply to all tests or to a subset of the tests.

Correctness of Lemmas

The correctness of all the proofs depends on the correctness of the lemmas added during the verification process to help the symbolic execution make progress (see [Workflow](#)). We make an effort to be careful and only add lemmas that we have confidence are correct, but human errors are always a possibility. The lemmas introduced are provided along with the tests and can be inspected for correctness.

Uninitialized State

When formally verifying a property, we would like to show that for every state that satisfies the preconditions, calling the function under verification will give the expected result (emitting an event, reverting with a certain message, returning an appropriate value, etc.). In our Foundry



tests, however, we initialize the states of the contracts by a sequence of operations that is not completely generic. In particular, we set whatever state is relevant to the execution of the function, but leave the rest of the state uninitialized. For example, in `testCannotDecrementLength` we ensure that:

- The key manager is the owner of the account.
- The controller has a certain subset of the permissions.
- The length of the `AddressPermissions` array is set to an unspecified number greater than 0.

But at the same time we leave other parts of the state uninitialized:

- There are no LSP17 extensions or LSP1 universal receiver delegates set.
- The data values for other data keys are empty.
- There are no other addresses (besides the controller and the Foundry test contract) with permissions.
- There is no `pendingOwner` (since it has been reset when ownership passed to the key manager).

This means that the formal verification is not necessarily considering every state that satisfies the preconditions of the property. For example, for `testCannotDecrementLength`, it is not considering the case where there is an LSP17 extension set, even though the property should still be satisfied in that case (if the other preconditions hold). This is because having to consider every possible state would potentially make the property much harder to formally verify.

Therefore, we instead make the assumption that any uninitialized state either a) is required by the preconditions to be uninitialized (for example, `testCannotAddUniversalReceiverDelegate` has as a precondition that no delegate has been set), or b) does not affect the execution of the function under verification. When writing the tests, we have tried to ensure by manual inspection of the code that this holds, that is, that every piece of state that does affect the execution of the function has been properly initialized.

Maximal Permissions

In the LSP6 permissions tests, our goal is to check that a controller cannot perform an action via the key manager without the required permission. For simplicity, instead of trying every possible combination of permissions, we give the controller all permissions except the required ones. This simplification is sound if we assume that giving *fewer* permissions to a controller will never allow it to perform *more* actions than it would be able to otherwise.

Balance Transfers

Some of the tests make calls that transfer value between addresses. In this case, we use Foundry's `deal` cheatcode to make sure that the sender has enough balance to transfer. For simplicity, we only give the sender the exact amount necessary for the transfer, but it is likely that the property would still hold if the sender had additional balance.

Excluded Addresses

To simplify the proofs, any addresses received as test inputs (such as the owner in the LSP0 tests, and the controller in the LSP6 tests) are assumed to be different from the following addresses:

- The Foundry test contract.
- The Foundry cheatcodes contract.
- The LSP0ERC725Account contract.
- The LSP6KeyManager contract.

Fixed Bytes

For formal verification, the tests that take a parameter of type `bytes` were changed to a fixed-length parameter of type `bytes32`. This is because, since a value of type `bytes` can have arbitrary length, the bytecode generated to perform the ABI encoding of such value requires iterating over the byte array. Loops are challenging to symbolic execution tools such as KEVM, making formal verification significantly more difficult. Using a `bytes32` parameter is equivalent to proving that the property holds when the size of the data is fixed at 32 bytes.

Abstraction of Compact Bytes Array Checks

The tests `testCannotAddNewAllowedCalls`, `testCannotChangeAllowedCalls`, `testCannotAddNewAllowedDataKeys` and `testCannotChangeAllowedDataKeys` test the permissions to add and modify the list of allowed calls and data keys for an address. Because these lists are represented in the compact bytes array format, the data being set is checked using the functions `LSP6Utils.isCompactByteArrayOfAllowedCalls` and `LSP6Utils.isCompactByteArrayOfAllowedERC725YDataKeys`. The problem is that these functions contain fairly complex loops, which as mentioned before are challenging to handle by formal verification. Although the call to the key manager is expected to revert due to insufficient permissions, the permissions check occurs after the compact bytes array check, meaning that the loop is in the execution path exercised by the test.

To handle this, we have chosen to verify a modified version of the code, with the following simplifications:

- Instead of a well-formed compact bytes array of arbitrary size, the test uses an arbitrary `bytes32` value (similar to above).
- The implementation of the `isCompactByteArray` functions is replaced with a function that always returns `true`.

These simplifications allow us to simulate what the execution of the test on a true compact bytes array would be like without having to deal with an arbitrary-length bytes value or with the loops in the `isCompactByteArray` functions. However, the following two conditions are necessary to guarantee that the execution is correctly simulated:

1. The `isCompactByteArray` functions must have no side effects.
2. Other than for the `isCompactByteArray` check, the execution must not depend on the length or format of the data.

If the first condition did not hold, then the simplified versions of the functions (that only return `true`) would not correctly reproduce the behavior of the original, as they would be missing the side-effect. Since both functions have the `pure` modifier, however, this condition is guaranteed. If the second condition did not hold, then it is possible that the fixed-length `bytes32` value would cause the execution to diverge from that of a well-formed compact bytes array. This condition was checked via manual inspection of the code.

Test Descriptions

`LSP0ERC725AccountTest.testCannotTransferOwnershipNotOwner`

Property: If `LSPERC725Account.transferOwnership` is called by an address that is not the account's owner, it reverts with the message "Ownable: caller is not the owner".

`LSP0ERC725AccountTest.testCannotAcceptOwnershipNotOwner`

Property: If `LSPERC725Account.acceptOwnership` is called by an address that is not the pending owner, it reverts with the message "LSP14: caller is not the pendingOwner".

`LSP0ERC725AccountTest.testIsValidSignature`

Property: If `LSPERC725Account.isValidSignature` is called with a hash correctly signed by the account's owner, it returns the magic value `_INTERFACEID_ERC1271`.

Preconditions:

- The owner's private key is valid (greater than 0 and less than the secp256k1 curve order).

ERC725XTest.testCannotExecuteIfNotOwner

Property: If `LSP0ERC725Account.execute` is called by an address who is not the account's owner, it reverts with the message "Ownable: caller is not the owner".

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

ERC725YTest.testSetData1NotOwner

Property: If the single-key version of `LSP0ERC725Account.setData` is called by an address who is not the account's owner, it reverts with the message "Ownable: caller is not the owner".

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

FallbackTest.testCallExtension

Property: If a call is made to an `LSP0ERC725Account` with a selector that does not correspond to any of the contract's externally-callable functions, then the account forwards the call to the LSP17 extension.

Preconditions:

- The address of the extension is greater than 0x9 (needed to avoid precompiled contracts - see [A01](#)).
- The function selector called is non-zero and does not correspond to any of the externally-callable functions of `LSP0ERC725Account`.
- If some `msg.value` is sent along with the call, the caller has enough balance to cover that amount.

Caveats:

- To simplify the verification, we assume that the address of the extension is neither of the following:
 - The account's owner.
 - The Foundry test contract.
 - The Foundry cheatcodes contract.
- Likewise, we assume that the caller is neither the account's owner nor the extension itself.
- We don't give the caller any more balance than the `msg.value` being sent (see [Balance Transfers](#), under General Caveats).
- The call's `msg.value` is non-zero.

LSP6KeyManagerTest.testCannotTransferOwnership

Property: If an address controller with all permissions except `_PERMISSION_CHANGEOWNER` tries to call `LSP0ERC725Account.transferOwnership` via the key manager, the call reverts with the error `NotAuthorised(controller, "TRANSFEROWNERSHIP")`.

LSP6KeyManagerTest.testCannotAcceptOwnership

Property: If an address controller with all permissions except `_PERMISSION_CHANGEOWNER` tries to call `LSP0ERC725Account.acceptOwnership` via the key manager, the call reverts with the error `NotAuthorised(controller, "TRANSFEROWNERSHIP")`.

Preconditions:

- The address of the previous owner is greater than `0x9` (needed to avoid precompiled contracts - see [A01](#)).
- The key manager is the pending owner.

LSP6KeyManagerTest.testCannotAddNewPermissions

Property: If an address controller with all permissions except `_PERMISSION_ADDCONTROLLER` tries to give permissions to a new address via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDCONTROLLER")`.

Preconditions:

- The new address has no permissions set (otherwise, the scenario is covered by `testCannotChangePermissions`).

LSP6KeyManagerTest.testCannotChangePermissions

Property: If an address controller with all permissions except `_PERMISSION_CHANGEPERMISSIONS` tries to change the permissions of an address via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDCONTROLLER")`.

Preconditions:

- The address whose permissions are being changed is not the same as the controller itself.
- The address already has some permissions set (otherwise, the scenario is covered by `testCannotAddNewPermissions`).

LSP6KeyManagerTest.testCannotAddExtensions

Property: If an address controller with all permissions except `_PERMISSION_ADDEXTENSIONS` tries to add an LSP17 extension via the key manager for a new function selector, the call reverts with the error `NotAuthorised(controller, "ADDEXTENSIONS")`.

Preconditions:

- There is no LSP17 extension set for that function selector (otherwise, the scenario is covered by `testCannotChangeExtensions`).

LSP6KeyManagerTest.testCannotChangeExtensions

Property: If an address controller with all permissions except `_PERMISSION_CHANGEEXTENSIONS` tries to change the LSP17 extension for a function selector via the key manager, the call reverts with the error `NotAuthorised(controller, "CHANGEEXTENSIONS")`.

Preconditions:

- There is already an extension set for that function selector (otherwise, the scenario is covered by `testCannotAddExtensions`).

LSP6KeyManagerTest.testCannotAddUniversalReceiverDelegate

Property: If an address controller with all permissions except `_PERMISSION_ADDUNIVERSALRECEIVERDELEGATE` tries to add an LSP1 universal receiver delegate via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDUNIVERSALRECEIVERDELEGATE")`.

Preconditions:

- There is no universal receiver delegate set in the account (otherwise, the scenario is covered by `testCannotChangeUniversalReceiverDelegate`).

LSP6KeyManagerTest.testCannotChangeUniversalReceiverDelegate

Property: If an address controller with all permissions except `_PERMISSION_CHANGEUNIVERSALRECEIVERDELEGATE` tries to change the LSP1 universal receiver delegate of the account via the key manager, the call reverts with the error `NotAuthorised(controller, "CHANGEUNIVERSALRECEIVERDELEGATE")`.

Preconditions:

- There is already a universal receiver delegate set in the account (otherwise, the scenario is covered by `testCannotAddUniversalReceiverDelegate`).

LSP6KeyManagerTest.testCannotAddMappedDelegate

Property: If an address controller with all permissions except `_PERMISSION_ADDUNIVERSALRECEIVERDELEGATE` tries to add a mapped LSP1 universal receiver delegate for a new `typeId` via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDUNIVERSALRECEIVERDELEGATE")`.

Preconditions:

- The account has no mapped delegate set for the `typeId` (otherwise, the scenario is covered by `testCannotChangeMappedDelegate`).

LSP6KeyManagerTest.testCannotChangeMappedDelegate

Property: If an address controller with all permissions except `_PERMISSION_CHANGEUNIVERSALRECEIVERDELEGATE` tries to change the mapped LSP1 universal receiver delegate for a `typeId` via the key manager, the call reverts with the error `NotAuthorised(controller, "CHANGEUNIVERSALRECEIVERDELEGATE")`.

Preconditions:

- The address already has a mapped delegate set for that `typeId` (otherwise, the scenario is covered by `testCannotAddMappedDelegate`).

LSP6KeyManagerTest.testCannotAddNewAddress

Property: If an address controller with all permissions except `_PERMISSION_ADDCONTROLLER` tries to add an address to a new index of the `AddressPermissions` array via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDCONTROLLER")`.

Preconditions:

- There is no previous address stored in that index of the `AddressPermissions` array (otherwise, the scenario is covered by `testCannotChangePermissions`).

LSP6KeyManagerTest.testCannotChangeAddress

Property: If an address controller with all permissions except `_PERMISSION_CHANGEPERMISSIONS` tries to change the address at an index of the `AddressPermissions` array via the key manager, the call reverts with the error `NotAuthorised(controller, "CHANGEPERMISSIONS")`.

Preconditions:

- There is already an address stored in that index of the `AddressPermissions` array (otherwise, the scenario is covered by `testCannotAddNewAddress`).

LSP6KeyManagerTest.testCannotIncrementLength

Property: If an address controller with all permissions except `_PERMISSION_ADDCONTROLLER` tries to set the length of the `AddressPermissions` array via the key manager to a higher number than the current one, the call reverts with the error `NotAuthorised(controller, "ADDCONTROLLER")`.

Preconditions:

- The old length is less than $2^{256} - 1$ and the new length is greater than the old length.

LSP6KeyManagerTest.testCannotDecrementLength

Property: If an address controller with all permissions except `_PERMISSION_CHANGEPERMISSIONS` tries to set the length of the `AddressPermissions` array via the key manager to a lower number than the current one, the call reverts with the error `NotAuthorised(controller, "CHANGEPERMISSIONS")`.

Preconditions:

- The old length is greater than 0 and the new length is smaller than the old length.

LSP6KeyManagerTest.testCannotSetData

Property: If an address controller with all permissions except `_PERMISSION_SUPER_SETDATA` and `_PERMISSION_SETDATA` tries to set a data key via the key manager, the call reverts with the error `NotAuthorised(controller, "SETDATA")`.

Preconditions:

- The key being set is not any of the special keys or prefixes handled by other permissions:
 - `_LSP6KEY_ADDRESSPERMISSIONS_ARRAY_PREFIX`
 - `_LSP6KEY_ADDRESSPERMISSIONS_PREFIX`
 - `_LSP1_UNIVERSAL_RECEIVER_DELEGATE_KEY`
 - `_LSP1_UNIVERSAL_RECEIVER_DELEGATE_PREFIX`
 - `_LSP17_EXTENSION_PREFIX`

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotTransferValue

Property: If an address controller with all permissions except `_PERMISSION_SUPER_TRANSFERVERVALUE` and `_PERMISSION_TRANSFERVERVALUE` tries to make a call via the key manager that transfers some of its balance, the call reverts with the error `NotAuthorised(controller, "TRANSFERVERVALUE")`.

Preconditions:

- The value transferred is greater than 0.
- The key manager has the value to be transferred in its balance.

Caveats:

- We don't give the key manager any more balance than necessary to perform the transfer (see [Balance Transfers](#), under General Caveats).
- For simplicity, we assume that the key manager already has the full balance necessary, and the controller doesn't need to send any additional value when making the call.
- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotMakeCall

Property: If an address controller with all permissions except `_PERMISSION_SUPER_CALL` and `_PERMISSION_CALL` tries to make a non-static call via the key manager, the call reverts with the error `NotAuthorised(controller, "CALL")`.

Preconditions:

- The data sent is not empty.
- The key manager has the value to be transferred in its balance.

Caveats:

- We don't give the key manager any more balance than the value being sent along with the call (see [Balance Transfers](#), under General Caveats).
- For simplicity, we assume that the key manager already has the full balance necessary, and the controller doesn't need to send any additional value when making the call.
- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotMakeStaticCall

Property: If an address controller with all permissions except `_PERMISSION_SUPER_STATICCALL` and `_PERMISSION_STATICCALL` tries to make a static call via the key manager, the call reverts with the error `NotAuthorised(controller, "STATICCALL")`.

Preconditions:

- The data sent is not empty.

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotDeployWithCreate

Property: If an address controller with all permissions except `_PERMISSION_DEPLOY` tries to deploy a contract via the key manager using the `CREATE` opcode, the call reverts with the error `NotAuthorised(controller, "DEPLOY")`.

Preconditions:

- The data sent is not empty.

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotDeployWithCreate2

Property: If an address controller with all permissions except `_PERMISSION_DEPLOY` tries to deploy a contract via the key manager using the `CREATE2` opcode, the call reverts with the error `NotAuthorised(controller, "DEPLOY")`.

Preconditions:

- The data sent is not empty.

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotAddNewAllowedCalls

Property: If an address controller with all permissions except `_PERMISSION_ADDCONTROLLER` tries to set the `AllowedCalls` array for a new address via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDCONTROLLER")`.

Preconditions:

- The new address doesn't have the `AllowedCalls` array set (otherwise, the scenario is covered by `testCannotChangeAllowedCalls`).

Caveats:

- The `isCompactByteArrayOfAllowedCalls` function is abstracted away (see [Abstraction of Compact Bytes Array Checks](#), under General Caveats).

LSP6KeyManagerTest.testCannotChangeAllowedCalls

Property: If an address controller with all permissions except `_PERMISSION_CHANGEPERMISSIONS` tries to change the `AllowedCalls` array for an address via

the key manager, the call reverts with the error `NotAuthorised(controller, "CHANGEPERMISSIONS")`.

Preconditions:

- The new address already has the `AllowedCalls` array set (otherwise, the scenario is covered by `testCannotAddNewAllowedCalls`).

Caveats:

- The `isCompactByteArrayOfAllowedCalls` function is abstracted away (see [Abstraction of Compact Bytes Array Checks](#), under General Caveats).

LSP6KeyManagerTest.testCannotAddNewAllowedDataKeys

Property: If an address controller with all permissions except `_PERMISSION_ADDCONTROLLER` tries to set the `AllowedDataKeys` array for a new address via the key manager, the call reverts with the error `NotAuthorised(controller, "ADDCONTROLLER")`.

Preconditions:

- The new address doesn't have the `AllowedDataKeys` array set (otherwise, the scenario is covered by `testCannotChangeAllowedDataKeys`).

Caveats:

- The `isCompactByteArrayOfAllowedCalls` function is abstracted away (see [Abstraction of Compact Bytes Array Checks](#), under General Caveats).

LSP6KeyManagerTest.testCannotChangeAllowedDataKeys

Property: If an address controller with all permissions except `_PERMISSION_CHANGEPERMISSIONS` tries to change the `AllowedDataKeys` array for an address via the key manager, the call reverts with the error `NotAuthorised(controller, "CHANGEPERMISSIONS")`.

Preconditions:

- The new address already has the `AllowedDataKeys` array set (otherwise, the scenario is covered by `testCannotAddNewAllowedDataKeys`).

Caveats:

- The `isCompactByteArrayOfAllowedCalls` function is abstracted away (see [Abstraction of Compact Bytes Array Checks](#), under General Caveats).

LSP6KeyManagerTest.testCannotReenter

Property: If an address controller with all permissions except `_PERMISSION_REENTRANCY` tries to make a reentrant call via the key manager, the call reverts with the error `NotAuthorised(controller, "REENTRANCY")`.

Caveats:

- For simplicity, the test uses a mock contract `CallerContract` with a `callExecute(LSP6KeyManager, bytes memory)` function. This function asks the given key manager to execute the given payload. The test calls the `callExecute` function of the `CallerContract` via the key manager, triggering a reentrant call. Although this setup is not general enough to encompass all possible reentrancy scenarios, it provides some assurance that the permission check for reentrancy works in the most common cases.
- For simplicity, we use `CallerContract` itself as the controller address, and we don't transfer any value along with the call.

Preconditions:

- The payload passed to `callExecute` is at least 4 bytes long (otherwise it would trigger the `InvalidPayload` error instead).

Caveats:

- The data parameter is fixed to be 32 bytes in length (see [Fixed Bytes](#), under General Caveats).

LSP6KeyManagerTest.testCannotSign

Property: If an address controller with all permissions except `_PERMISSION_SIGN` tries to call `LSP6KeyManager.isValidSignature`, the call returns `_ERC1271_FAILVALUE`.

Preconditions:

- The controller's private key is valid (greater than 0 and less than the secp256k1 curve order).
- The signature is correct.

LSP6KeyManagerTest.testKeyManagerForwardsAllBalance

Property: If a controller makes a call via the key manager, transferring some value, the balance of the key manager after the call will be the same as before (the key manager forwards the entire value to the account).

Preconditions:

- The address of the called contract is greater than 0x9 (needed to avoid precompiled contracts - see [A01](#)).
- The controller has the value to be transferred in its balance.
- The balance of the key manager, plus the value transferred by the controller, is less than 2^{256} (to avoid an overflow).

Caveats:

- We don't give the controller any more balance than the value being sent along with the call (see [Balance Transfers](#), under General Caveats).
- For simplicity, the test is set up so that the account transfers the entire value that it receives from the key manager to the contract being called. This could be generalized so that some of the value remains in the account (regardless, it would not remain in the key manager).

LSP6KeyManagerTest.testUniversalReceiverDelegateUP

Property: If LSP1UniversalReceiverDelegateUP is called as the universal receiver delegate of an account, only this account will be modified, and not any other account that the contract is serving as a delegate for. Additionally, no other storage slots besides the LSP5 and LSP10 keys of the account will be modified.

Preconditions:

- The address of the called contract is greater than 0x9 (needed to avoid precompiled contracts - see [A01](#)).
- The delegate has the permission to set the account's data.

Caveats:

- This property cannot currently be expressed in Foundry, as there is no easy way to check what addresses or storage positions have been modified by a test. We have implemented custom cheatcodes in KEVM in order to check this property.

- As an approximation that *can* be executed as a Foundry test, we created an alternative version of the test where the delegate is only given permissions to the account whose `universalReceiver` function is called. This way, if it tries to write to any other account, the test should revert and fail.
- Because it is hard to write a test considering an arbitrary number of accounts, all using the same universal receiver delegate, we simplify the scenario to consider only a single other account sharing the same delegate.
- We considered the following range of scenarios, in all possible combinations, which determine the keys that must be set beforehand and the type ID passed to the `universalReceiver` function:
 - Whether the `universalReceiver` function is being called due to the transfer of an LSP7 token, an LSP8 token or an LSP9 vault.
 - Whether the account is the sender or the receiver of the asset.
 - Whether the asset is already registered in the account.
- In our tests, no other assets have been registered to the account.
- Similarly to `testCannotReenter`, we use a specific mock contract to simulate an LSP7 or LSP8 token. This contract only has a `balanceOf` function, which returns a predefined balance. As this is the only function called by `LSP1UniversalReceiverDelegateUP`, it is enough to simulate the behavior of the asset. For LSP9 vaults, we use the standard `LSP9Vault` contract.

Verification Results

In the time span of the engagement, we were able to formally verify a majority of the tests described in the [Properties](#) section. Formal verification is challenging, and our tooling is still in active development, so some proofs could not be completed within the time frame of the engagement. However, all tests have also been run as regular Foundry fuzz tests (#runs = 256) and passed (in the case of [testUniversalReceiverDelegateUP](#), the alternative version described under “Caveats”).

The following table shows the verification results for each of the tests:

- **PASSING** indicates that the test completed symbolic execution and the property was satisfied, with the simplifications and caveats listed in [Properties](#).
- **STUCK** indicates that symbolic execution of the test could not be completed due to technical obstacles. This does not mean that a bug has been found, and it is possible that with more manual work to guide the prover the test could pass verification.

Test	Result
LSP0ERC725AccountTest. testCannotTransferOwnershipNotOwner	PASSING
LSP0ERC725AccountTest. testCannotAcceptOwnershipNotOwner	STUCK
LSP0ERC725AccountTest. testIsValidSignature	STUCK
ERC725XTest. testCannotExecuteIfNotOwner	PASSING
ERC725YTest. testSetData1NotOwner	PASSING
FallbackTest. testCallExtension	PASSING
LSP6KeyManagerTest. testCannotTransferOwnership	PASSING
LSP6KeyManagerTest. testCannotAcceptOwnership	STUCK
LSP6KeyManagerTest. testCannotAddNewPermissions	PASSING

LSP6KeyManagerTest. testCannotChangePermissions	PASSING
LSP6KeyManagerTest. testCannotAddExtensions	PASSING
LSP6KeyManagerTest. testCannotChangeExtensions	PASSING
LSP6KeyManagerTest. testCannotAddUniversalReceiverDelegate	PASSING
LSP6KeyManagerTest. testCannotChangeUniversalReceiverDelegate	PASSING
LSP6KeyManagerTest. testCannotAddMappedDelegate	PASSING
LSP6KeyManagerTest. testCannotChangeMappedDelegate	PASSING
LSP6KeyManagerTest. testCannotAddNewAddress	PASSING
LSP6KeyManagerTest. testCannotChangeAddress	PASSING
LSP6KeyManagerTest. testCannotIncrementLength	STUCK
LSP6KeyManagerTest. testCannotDecrementLength	STUCK
LSP6KeyManagerTest. testCannotSetData	PASSING
LSP6KeyManagerTest. testCannotTransferValue	PASSING
LSP6KeyManagerTest. testCannotMakeCall	PASSING
LSP6KeyManagerTest. testCannotMakeStaticCall	PASSING
LSP6KeyManagerTest. testCannotDeployWithCreate	PASSING
LSP6KeyManagerTest. testCannotDeployWithCreate2	PASSING
LSP6KeyManagerTest. testCannotAddNewAllowedCalls	PASSING

LSP6KeyManagerTest. testCannotChangeAllowedCalls	PASSING
LSP6KeyManagerTest. testCannotAddNewAllowedDataKeys	PASSING
LSP6KeyManagerTest. testCannotChangeAllowedDataKeys	PASSING
LSP6KeyManagerTest. testCannotReenter	PASSING
LSP6KeyManagerTest. testCannotSign	STUCK
LSP6KeyManagerTest. testKeyManagerForwardsAllBalance	STUCK
LSP6KeyManagerTest. testUniversalReceiverDelegateUP	STUCK

Findings

In the course of writing, fuzzing and formally verifying tests, the following findings have been identified.

A01: `supportsERC165InterfaceUnchecked` is called without checking if address supports `IERC165`

[Severity: Low | Difficulty: High | Category: Validation]

The `ERC165Checker.supportsERC165InterfaceUnchecked` function checks if an address supports a specific ERC165 interface identifier. To be able to perform this check, it assumes that the address implements `IERC165`, and if this assumption is violated the behavior of the function is undefined. However, in many parts of the code this function is called on an address without checking if it implements the interface first.


In particular, when called on the addresses `0x2`, `0x3` or `0x4`, which correspond to Ethereum precompiled contracts, this function always returns `true` (for any interface ID) even though those are not true contracts and do not implement `IERC165`. It is unclear whether some interface IDs might also return `true` for other precompiled contracts.

Scenario

We have not identified any potential vulnerabilities that can emerge from this behavior. In the contracts in scope, `supportsERC165InterfaceUnchecked` is called in the following circumstances:

- On the new owner and previous owner during the process of ownership transfer of an `LSP0ERC725Account`.
- On the universal receiver delegate when the account's `universalReceiver` function is called.
- On the account's owner in `LSP1UniversalReceiverDelegateUP.universalReceiver`.
- On the notifier of a call to `LSP1UniversalReceiverDelegateUP.universalReceiver` resulting from a vault transfer.

In the first three cases, to trigger the issue a controller would need to set either the account's owner or universal receiver delegate to one of the precompiled contracts. The last case should



not be possible for an account deployed as a `LSP0ERC725Account` contract, since the implementation of `LSP0ERC725Account.universalReceiver` passes `msg.sender` to `LSP1UniversalReceiverDelegateUP.universalReceiver` as the notifier. Since a precompiled contract should not be able to make a call, it cannot be the `msg.sender`.

Therefore, this scenario should be easy to prevent as part of users doing their due diligence of controlling access to the account and making sure that components such as `LSP1UniversalReceiverDelegateUP` are used properly as intended. An attacker that gains this level of control over the account would have the ability to do far more damage by other means.

Recommendation

The first option is to change `supportsERC165InterfaceUnchecked` to `supportsInterface`, which does check that the contract supports ERC165 before checking the interface ID. Otherwise, this behavior should be documented and developers should be aware of it when adding to or modifying the code to avoid introducing more serious issues. Of course, users should also be careful when transferring ownership and adding delegates to make sure that they are using the correct address.

Status

Tests reproducing this behavior were added in PR [#388](#).

A02: Compilation errors when using Solidity 0.8.13

[Severity: Low | Difficulty: N/A | Category: Compiler]

LSP7CompatibleERC20, LSP7CompatibleERC20InitAbstract, LSP8CompatibleERC721 and LSP8CompatibleERC721InitAbstract have compiler errors when compiled using version 0.8.13 or higher of the Solidity compiler. The reason for these errors is that the supportsInterface function from ERC725YCore (inherited via LSP4Compatibility) is not being properly overridden.

Recommendation

- Add an override for the supportsInterface function to LSP7CompatibleERC20 and LSP7CompatibleERC20InitAbstract.
- Add ERC725YCore to the overrides clause of supportsInterface in LSP8CompatibleERC721 and LSP8CompatibleERC721InitAbstract.

Status

Fixed in PR [#389](#).

A03: LSP1UniversalReceiverDelegateUP calls target() on non-LSP6 accountOwner

[Severity: Medium | Difficulty: Low | Category: Usability]

The `_validateCallerViaKeyManager` function in `LSP1UniversalReceiverUP` includes the following code:

```
if (accountOwner.supportsERC165InterfaceUnchecked(_INTERFACEID_LSP6))
    ownerIsKeyManager = true;

address target = ILSP6KeyManager(accountOwner).target();
```

This code converts `accountOwner` to the `ILSP6KeyManager` type and calls the `target()` function on it even if it does not implement the LSP6 interface, as per the preceding `if` statement.

Scenario

If the `accountOwner` is not a key manager, then the call to `target()` will likely either revert or call the owner's `fallback()` function if one exists. Even in the later case, the target will be invalid and will cause the validation to revert. This means that `LSP1UniversalReceiverUP` can never be used when the account owner is not a key manager, as it will cause calls of `universalReceiver` to always revert.

Recommendation

Move the call to `target()` inside the `if` condition.

Status

Fixed in PR [#448](#).

A04: Payload is sliced without checking bounds first

[Severity: Low | Difficulty: Low | Category: User experience]

When calling the `LSP0ERC725Account.execute` function via the key manager, the payload of the function is passed to `LSP6KeyManager._verifyCanExecute` in order to validate the controller permissions. `_verifyCanExecute` takes slices of the payload in order to check the different arguments, but it doesn't check that the length of the payload is large enough before slicing it.

Scenario

If a user accidentally sends an improperly-constructed payload that is too short, the slicing attempt will produce a revert with an unhelpful low-level EVM error message.

Recommendation

Check the length of the payload before slicing, or document thoroughly so that users are aware of the minimum payload length requirement.

Status

Partially fixed in PR [#449](#).

A05: Empty call can be made via the key manager without any permissions

[Severity: Medium | Difficulty: Low | Category: Permissions]

The function `_verifyCanExecute` in `LSP6KeyManager` is called to check if a controller has the right permissions when trying to make a call or transfer via the key manager. However, if the `calldata` is empty and no value is transferred, then the controller is allowed to make the call even if it does not have any permissions. This would allow controllers without `_PERMISSION_CALL` to call the fallback function of a contract.

Recommendation

Add a call to `_requirePermissions` covering the case where the `calldata` is empty and the value is 0.

Status

The following piece of code has been added to the `_verifyCanExecute` function in PR [#450](#):

```
if (!hasSuperOperation && !isCallDataPresent && value == 0) {
    _requirePermissions(
        from, permissions, _extractPermissionFromOperation(operationType));
}
```

A06: `_PERMISSION_CALL` can be bypassed by constructing an adversarial payload

[Severity: Medium | Difficulty: Low | Category: Permissions]

To perform an operation on an account via the key manager, a controller needs to have the appropriate permissions. One of these is `_PERMISSION_CALL`, which allows a controller to use the account to make non-static calls to other contracts. Transfers that send some value but have no calldata don't require `_PERMISSION_CALL`, only `_PERMISSION_TRANSFER`. However, if an attacker crafts an adversarial payload that is not well-formed, it is possible to bypass the `_PERMISSION_CALL` check and make a call with data having only `_PERMISSION_TRANSFER`.

Details

To make a call via the key manager, a controller passes to the `LSP6KeyManager.execute` function an ABI-encoded payload consisting of:

1. The function selector of the `LSP0ERC725Account.execute` function (4 bytes).
2. The `OPERATION_0_CALL` operation type identifier (first argument, 32 bytes).
3. The encoded address of the target contract being called (second argument, 32 bytes).
4. The `msg.value` that should be sent to the target (third argument, 32 bytes).
5. The encoded calldata (fourth argument, ? bytes).

The fourth argument of the `LSP0ERC725Account.execute` function is a value of type `bytes` containing the calldata sent to the target contract. Since the `bytes` type is a dynamic type, its ABI encoding is split into a "head" of fixed size and a "tail" of variable size. Therefore, a calldata with length K will be ABI encoded as follows:

1. Head: Offset pointing to the start of the tail (32 bytes).
2. Tail: Length of byte array = K (32 bytes).
3. Tail: Content of the byte array, right-padded with zeros into a multiple of 32 bytes (K rounded to the next multiple of 32 bytes).

From this, we can conclude that a well-formed payload will have a size of at least $164 + K$ bytes ($4 + 32 + 32 + 32 + 32 + 32 + K$ bytes). Following this, the `_verifyCanExecute` function in `LSP6KeyManager` considers the calldata to be empty if the payload is no larger than 164 bytes, since in this case K must be 0.

The problem is that there is no guarantee that the payload is well-formed, and in fact it is possible for an attacker to craft a specific payload that can send non-empty calldata in 164 bytes. As long as some value is transferred along with the call, they would be able to execute this calldata with only `_PERMISSION_TRANSFER`.

Scenario

An attacker can construct a 164-byte payload like the following:

```
0x44c028fe
0x00: 0x0000000000000000000000000000000000000000000000000000000000000000
0x20: 0x0000000000000000000000000000000000000000000000000000000000000000
0x40: 0x0000000000000000000000000000000000000000000000000000000000000024
0x60: 0x0000000000000000000000000000000000000000000000000000000000000040
0x80: 0xdeadbeef00000000000000000000000000000000000000000000000000000000
```

In the above, the first line is the 4-byte function selector for `LSP0ERC725Account.execute`, and the number next to each line after that is the byte offset where the line starts (counting from the end of the selector). The line starting at `0x00` is the `OPERATION_0_CALL` identifier, the line at `0x20` is the 20-byte address of the target contract, and the line at `0x40` is the value to be transferred with the call.

Note that the line at `0x60` should contain the offset where the tail of the ABI encoding of the calldata should start. Instead of pointing to an offset after the head, however, it points *back* to the offset `0x40`, which contains the value to be transferred. Therefore, when decoding, the value `0x24` will be interpreted as the length of the calldata: 36 bytes. The next 36 bytes after that (`0x0040deadbeef`) will be interpreted as the contents of the calldata. Because the first four bytes (which should contain the function selector) are `0x00000000`, the call will be forwarded to the fallback function of the target contract (as long as it is marked payable), with the remaining 32 bytes as the `msg.data`.

Note that the value at `0x40` can be adjusted to be higher, as long as it remains within the bounds of the payload (the attacker may need to send this amount along with the call, but it will always be small). Because the value at `0x60` must be `0x40`, however, the function selector will necessarily be `0x00000000`, and the first 28 bytes of the `msg.data` will be `0x0040`. However, the remaining bytes can contain any value that the attacker wants.

This attack would become more serious if an attacker was able to craft a similar payload that calls a function selector for an existing function of the target contract. One case in which this might be possible would be if the function selector happened to be a substring of the target contract's address, but this scenario is unlikely. It is not clear if constructing a payload that can call an arbitrary selector is possible.

Recommendation

Since only the last argument of `LSP0ERC725Account.execute` is a dynamic type, in a well-formed payload the offset will always point to position `0x80`. Checking this consistency condition will prevent using an invalid offset to bypass the `_PERMISSION_CALL` check.

Note that, as an additional consistency condition, if the payload is at most 164 then the length at position `0x80` should be zero. However, the code emitted by the Solidity compiler already includes consistency checks for the size of function arguments, and so any payload that violates this condition should anyway cause the call to `LSP0ERC725Account.execute` to revert.

Similar Findings

The `_verifyAllowedCall` function has a similar check based on the length of the payload, to decide whether there is a function selector included that must be matched to one of the allowed selectors for that controller. Since using the above attack to call a specific function selector is more challenging, it is unclear if this attack can similarly be used to call a selector that is not allowed. Regardless, adding the above consistency check should be able to prevent this case as well.

Status

Fixed in PR [#489](#).