Lido Finance

Easy Track Contract Review

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Easy Track smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the EasyTrack smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/-closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the EasyTrack smart contracts.

Overview

Lido is a staking solution for ETH 2.0 built to solve illiquidity, immovability and accessibility by making staked ETH liquid and allowing for participation with any amount of ETH to improve security of the Ethereum network.

Lido DAO governance currently relies on the Aragon voting model, in which proposals are approved or rejected via direct governance voting. Lido looks to introduce a complimentary voting model that is efficient for decisions only affecting small groups of Lido DAO members.

> EasyTrack is a voting model where a motion is considered to have passed if the minimum objections threshold hasn't been exceeded [within the configured time frame]. With EasyTrack motions there's no need to vote "pro", token holders only have to vote "contra" if they have objections. There is also no requirement to ask the broader DAO community to vote on proposals that spark no debate, making it easier to manage. [1]

Security Assessment Summary

This review was conducted on the files hosted on the EasyTrack repository and were assessed at commit bc0904c.

Retesting activities targeted commit d7c0b24.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [2, 3].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 9 issues during this assessment. Categorized by their severity:

- High: 1 issue.
- Low: 3 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the EasyTrack smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
LET-01	DOS via Uninitialized `EasyTrack` Implementation Contract	High	Resolved
LET-02	LDO Holder DOS Potential	Low	Resolved
LET-03	Motion Creators Can Block Creation of Other Motions	Low	Resolved
LET-04	Sub-Optimal Reentrancy Protections	Low	Resolved
LET-05	Objections Allowed After Voting Duration Elapsed	Informational	Closed
LET-06	`EVMScriptPermissions` Gas Efficiency Considerations	Informational	Closed
LET-07	EasyTrack Gas Saving Considerations	Informational	Resolved
LET-08	Testing Findings And Suggestions	Informational	Resolved
LET-09	Miscellaneous Findings	Informational	Resolved

LET-01	DOS via Uninitialized `EasyTrack` Implementation Contract		
Asset	`EasyTrack.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The `EasyTrack` implementation contract is left uninitialized during deployment, allowing an attacker to execute `__EasyTrackStorage_init()` to take control of the underlying implementation contract and exploit a vulnerability in the OpenZeppelin upgrade mechanism to destroy the implementation and render the Easy Track system unusable.

It is considered best-practice to initalize the implementation, but it can often be left uninitialized without issue. While an attacker could then initialize the implementation, changes to the implementation's storage have no effect on the proxy contract's state.

In this case, however, the OpenZeppelin `UUPSUpgradeable` contract contains a vulnerability that enables the `DEFAULT_ADMIN_ROLE` to destroy the contract via `selfdestruct()`.[1] When exploited on the underlying implementation, an attacker who initalized the contract to assume the `DEFAULT_ADMIN_ROLE` can then destroy it to render the proxy contract unusable.

Given the publicly announced advisory, this exploit can be readily identified by a reasonably skilled attacker. As such, this issue has been deemed exploitable with a fairly high likelihood (provided it is not mitigated).

While an exploit would render the current EasyTrack system unusable, this would not halt the functioning of the Lido protocol as a whole. To recover, the Lido ecosystem could continue to operate as it does currently (via Aragon DAO voting) and deploy a fresh Easy Track system with the implementation properly initialized and exploit resolved. Although not critical, this attack would likely cause reasonable disruption and incur costs in needing to re-deploy the system.

Recommendations

Ensure `__EasyTrackStorage_init()` has been called on any existing deployments of the `EasyTrack` contract.

Upgrade OpenZeppelin to v4.3.2 or later. While this is sufficient to protect against the current vulnerability, it is preferable to disable functionality on the underlying implementation to prevent this class of issues.

Ensure the EasyTrack implementation contract is initialized during deployment. Preferable is to add a constructor that executes `__EasyTrackStorage_init()`, so the implementation is deployed and initialized in the same transaction. This protects against any risk of an attacker front-running the initialize should the deployment script deploy and execute in separate transactions.

A suitable constructor could be added to `EasyTrackStorage` as follows:

[1] Relevant OpenZeppelin advisory: https://forum.openzeppelin.com/t/security-advisory-initialize-uups-implementation-contracts/15301.

```
constructor () {
    __EasyTrackStorage_init(address(0), address(0));
}
```

Though this could reasonably be added to `EasyTrack`, keeping it in the same contract as top-level initializer leave less room for error.

Resolution

In PR #7, the Lido development team entirely removed the upgradability functionality from `EasyTrack`. That is, the `EasyTrack` contract is no longer derived from `UUPSUpgradeable` and will be deployed and used directly (rather than as the implementation for a `ContractProxy`).

This removes any vulnerabilities associated with the feature, resolving this issue.

Future upgrades will involve deployment of a new version of the `EasyTrack` contract, with the Lido DAO voting to replace the old one.

Functionality has been introduced to the `EVMScriptExecutor` to allow the Lido DAO to change the address of the `EasyTrack` contract approved to execute scripts.

Resolution Details

The testing team note some interesting details arising from the implemented resolution that have no bearing with regards to the correctness or validity of the resolution:

- The new upgrade process involves some complications and increased gas costs. The replacement contract will need to have all relevant storage changes made prior to being proposed to the DAO as a replacement (such as the `AccessControl` roles, factory permissions).
- There is no potential for bugs involved with inconsistent storage layouts across upgrades.
- Although upgrades are more gas expensive, normal use of the `EasyTrack` contract should involve less gas (as EasyTrack function calls no longer involve the layer of indirection provided by passing through a `ContractProxy`).

LET-02	LDO Holder DOS Potential		
Asset	`EasyTrack.sol` & `MotionSettings.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Holders of the LDO governance token may submit objections to proposed motions, which block the motion once a suitable threshold of objections have been reached. If this threshold is configured too low, a single malicious entity may block operation of the EasyTrack system by objecting to every proposed motion.

The likelihood of such an attack is primarily dependent on associated crypto-economic incentives, the distribution of LDO governance tokens, and the threshold required to object. As disrupting EasyTrack may devalue LDO, an attack would likely incur significant cost – both in gas fees to submit the objections, and costs to accumulate sufficient LDO.

In terms of impact, any DOS of the EasyTrack is limited in severity. While the `EasyTrack` system can be subject to denial-of-service, it can always be overridden by the Aragon voting DAO. The DAO can directly execute scripts on `EVMScriptExecutor` to bypass any blockage, and may increase the `objectionsThreshold` (within limit) or punish a malicious minority via other means. As such, this DOS can only delay motions and increase their cost, not block them entirely.

Note that too high an `objectionsThreshold` can limit EasyTrack's safety properties with regards to protections against malicious motions. A careful balance is important.

Also consider whether any motions are more urgent or time-critical, allowing the attacker to cause more significant disruption to the Lido ecosystem by introducing delays.

Recommendations

If not yet carefully evaluated, consider how feasible it is for a single malicious entity to acquire an `objectionsThreshold` proportion of the total LDO balance.

Consider whether `MAX_OBJECTIONS_THRESHOLD` should be increased past 5%, to account for future potential LDO distributions. Any increase in the `objectionsThreshold` may need an associated increase in the `motionDuration`, to provide a suitable voting window for objections from good-natured LDO holders (accounting for network congestion and potential for limited miner censorship).

It would be beneficial to include any relevant analysis in associated documentation.

Resolution

The Lido development team have explained the careful reasoning behind a `MAX_OBJECTIONS_THRESHOLD` of 5%, stating that:

> With LDO total supply of $1\,000\,000\,000$, 5% is $50\,000\,000$ tokens. Costs to obtain such amount of LDO by one malicious entity would potentially outweigh destructive impact on the project. But, even if that happens, we can update our implementation of EasyTrack (redeploy it) with a larger threshold limit.

The testing team note that the existing price history of LDO enables a more accurate `objectionsThreshold` configuration than if EasyTrack had been introduced in the initial deployment (while the LDO valuation was relatively uncertain).

LET-03	Motion Creators Can Block Creation of Other Motions		
Asset	`EasyTrack.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `EasyTrack` contract enforces a global limit to the number of currently active motions. Once reached, no new motions can be created. If a malicious actor is able to create new motions, they can flood the contract with motions to fill up this list and effectively halt the operation of `EasyTrack`.

The likelihood of any associated attack is fairly low. The current factories correctly restrict authorized creators to a small number of `trustedCaller` and node operator accounts, with the `trustedCaller` accounts expected to be multisigs controlled by a small committee. These should all be highly trusted and well-known entities, so it is unlikely that any of them are malicious.

In the event that one of these accounts is compromised by a malicious entity, the severity of this attack is also limited. While the `EasyTrack` system can be subject to denial-of-service, it can always be overridden by the Aragon voting DAO. The DAO can directly execute scripts on `EVMScriptExecutor` to bypass any blockage, and can replace the factory which authorizes the malicious account (via `EVMScriptFactoriesRegistry.removeEVMScriptFactory()`).[2]

As such, the testing team deems the associated severity to be *Low*.

Recommendations

Be conscious about these risks when adding new factories, and otherwise authorizing new entities to create motions. Ensure that any updates continue to prevent unauthenticated attackers from creating motions.

Consider introducing clear, visible documentation to highlight this for any future maintainers and relevant stakeholders; both in the NatSpec comments for `EVMScriptFactoriesRegistry.addEVMScriptFactory()` and in external documentation like the project README or specification.

One could also consider enforcing a separate motion limit per factory, but this is likely not worth the increased gas costs.

Resolution

Relevant documentation has been introduced as part of PR #6. In particular, the README contains documentation describing appropriate guidelines and best practices for those creating a new EVMScript factory, and an appropriate warning was added to the NatSpec for `EVMScriptFactoriesRegistry.addEVMScriptFactory()`.

[2] Not to mention any other controls the DAO may have to act directly on the relevant contracts like the finance contract.

LET-04	Sub-Optimal Reentrancy Protections		
Asset	`EasyTrack.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The functions `createMotion()` and `enactMotion()` are sub-optimal with regards to resilience against reentrancy attacks. They have no check to prevent reentrancy (e.g. a `noReentrancy` modifier based on a storage flag) and do not properly apply the Checks-Effects-Interactions pattern.

The external contracts accessed are relatively trustworthy (`EVMScriptExecutor` , and the factory) and no impactful exploits were identified. A clearly malicious factory is not part of a reasonable threat model, as these are approved by the DAO. However, it is more feasible to consider a scenario in which the DAO is tricked into accepting an obfuscated or vulnerable factory.

Recommendations

Restructure `createMotion()` and `enactMotion()` to better follow the checks-effects-interactions pattern.

In `enactMotion()` , execute `_deleteMotion()` and emit `MotionEnacted()` before making any external contract calls.

In `createMotion()` , increment `lastMotionId` and save as much of the `Motion` struct to storage as possible before executing `_createEVMScript()` (it should be possible to store all but `evmScriptHash` prior).

Alternatively, these functions could be protected via a `noReentrancy` modifier, but this is likely not worth the additional storage costs.

Resolution

This has been fixed in PR #8.

LET-05	Objections Allowed After Voting Duration Elapsed
Asset	`EasyTrack.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

LDO holders can submit a “late” objection to a motion, where its duration has already elapsed (provided the motion has not yet been enacted and deleted).

This may introduce some risk of front-running abuse, where a malicious whale slightly wastes the enacting account’s gas by waiting for the `enactMotion` () transaction to be released before front-running an objection sufficient to reach the threshold.

Recommendations

Consider requiring that `objectToMotion()` can only succeed for open motions whose voting duration has yet to elapse (the opposite of the check in `enactMotion()` at line [104]).

Resolution

The Lido team have acknowledged the issue and deemed there to be sufficient risk mitigation without code changes.

Any exploit would require the attacker to control a similar amount of LDO as for LET-02, and also pay higher gas fees to reliably front-run the enacting account.

If abused, the Lido DAO can increase the objections threshold such that the attack becomes increasingly economically infeasible (if it were not already).

The Lido team also point out that the existing functionality is useful in scenarios where the motion can no longer be enacted successfully (e.g. due to some conflicting state change). Here, the smaller threshold of LDO holders can clean up (delete) the outstanding motion without resorting to an expensive Aragon DAO vote.

The testing team also note that the gas costs alone expended to continually execute this attack would likely greatly outweigh any impact on the entity enacting motions (not to mention the large LDO prerequisite).

LET-06	`EVMScriptPermissions` Gas Efficiency Considerations
Asset	`contracts/EVMScriptFactoriesRegistry.sol` & `contracts/libraries/EVMScriptPermissions.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The current implementation of factory permissions introduces efficiency and complexity tradeoffs that may be undesirable, depending on the usual number of scripts associated with each factory, and the expected number of calls in associated scripts.

Each EVMScript Factory registered with the EasyTrack system has an associated set of permissions restricting which contract functions ("methods") may be called in their scripts. These are represented in the form of a series of `(bytes20 address, bytes4 function_selector)` tuples, concatenated into a `bytes` value.[3]

Note that, for the purposes of EasyTrack, an EVMScript is simply a series of external contract calls.

The current implementation is most problematic with regards to gas efficiency when a factory has a large permissions set, associated scripts involve many calls, and the associated motions are enacted regularly.

Consider the following points regarding the current implementation:

1. Permissions lookup via `_hasPermission()` occurs in $O(n)$ time and gas (where $n =$ number of permissions associated with that factory).
2. To validate a script containing m calls via `canExecuteEVMScript()` will take $O(n \times m)$ time and gas.
3. The entire set of permissions is always loaded from storage, even if the relevant permission is the first one in the list.

The current implementation appears to prioritize storage space efficiency, which is most pronounced when the permission set contains only one permission tuple (the entire `bytes` value can be packed into a single storage slot[4]), or many tuples (it is only for values with 4 or more tuples does the space packing become more efficient than the more simply encoded `bytes24[]` [5])

The current implementation may be often reasonable, as storage access costs can usually heavily outweigh other execution costs. However, ensure these other scenarios are considered:

A large permission set and large script:

For a large script with distinct calls, we can expect that the entire permission set would need to be loaded from storage (so we can assume all permissions are available via `bytes memory` or `bytes24[] memory`). If gas efficiency were a significant concern, an implementation could improve lookup from $O(n)$ by relying on ordering of permissions to perform a binary search in $O(\log n)$. However, because Solidity includes no "builtin" `memory`

[3] Equivalent to a packed encoding of `bytes20, bytes4, [bytes20, bytes4, ...]`.

[4] The encoded permissions uses 24 bytes, which is short enough to be stored together with the array length. See https://docs.soliditylang.org/en/v0.8.7/internals/layout_in_storage.html#bytes-and-string.

[5] An encoded `bytes` representation with 4 tuples can be stored in the same space as a `bytes24[]` of length 3: $4 \times \texttt{tuple\_length} = 4 \times 24 = 96 = 3 \times 32$. So, for permissions with 2-3 entries, the storage used is equivalent.

collections with better than $O(n)$ lookup, this would involve more complicated and potentially error-prone logic that would be less efficient for small permissions sets.

A large permission set and a small script:
(Or also a larger script containing only repeated calls to the same method.)

Loading the entire permissions set into memory can dominate in this case, particularly if regularly executed. A more efficient implementation here could involve mappings such that only a single `SLOAD` is involved (e.g. `address => bytes` instead of the tuple, with the `bytes` containing concatenated `bytes4` function selectors).

Permission sets usually containing 2-3 tuples and not often used:

Here, gas is less of a concern, and it may be worth considering a more gas expensive but simple solution that avoids the technical complexity and risk potential associated with the low level assembly needed to handle the permissions decoding.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing any relevant suggestions as appropriate for the expected use-case.

Also take into account the costs associated with needing to deploy an updated implementation, if this use-case changes drastically.

Resolution

The Lido team have acknowledged this and confirmed that the tradeoffs described where carefully considered for the current `EVMScriptPermissions` implementation, which is intentionally designed to minimize gas use for the current factory implementations.

Each of these initial factories only require a single permission entry and the resulting EVMScripts only contain a small number of function calls.

Should the frequently executed scripts change such that the existing the library implementation is inefficient, it can be updated (replaced) as needed.

LET-07	EasyTrack Gas Saving Considerations
Asset	`EasyTrack.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details findings relating to gas savings or optimizations which do not have direct security implications:

1. There is on-chain reason for `objectionsAmountPct` to be saved to storage (it's recalculated each time, and the stored value is never used)

   Though it may be useful to off-chain users via `getMotion`. Consider removing it from the `Motion` struct to save on storage gas fees.

2. It is possible to make large storage cost savings by storing the majority of the Motion struct on-chain as a hash only. While `objectionsAmount` changes regularly, so should be stored on-chain, the remaining fields are immutable (not changing for the life of the motion) and could feasibly be condensed into a hash.

   These values would need to be passed as parameters during relevant function calls, and compared against the stored hash to verify them.

   While this would save on gas, it may be a problematic user experience for those now needing knowledge of the historical state in order to object. Data availability issues may also be a consideration.

   A middle ground could also be considered, where some values are stored on-chain for improved data availability and user-experience, but others are kept only in the hash.

3. It is unnecessary to create the EVM script during `enactMotion()`. It may be preferable and cheaper to pass the entire script as a parameter and validate it against the stored hash.

   Note that an external call to the factory may still be needed if it's possible for the creator to *no longer* be authorized by the factory (where it was allowed to create a script at the time of `createMotion()` but not later).

4. It would be more gas efficient to avoid checking the permissions for an EVM script in `enactMotion()`, and instead simply checking the script against the stored hash (a hash collision should be infeasible).

   There would be some complications, however, in safely handling factories that have been removed (or removed and re-added with different permissions) since the motion was created. This may be possible by storing with each factory the timestamp or block number when it was added (or its permissions had last changed).

5. Depending on how gas-expensive `totalSupplyAt(snapshotBlock)` is, it may be desirable to cache this in storage.

   A negative is that the entity saving it to the cache would incur higher gas costs (either the proposer in `createMotion()` or the first objector).

   Saving it during `createMotion()` also incurs extra gas costs in the common, "happy" scenario in which no objections are raised.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The above comments have been considered and relevant fixes were introduced in PR #6.

The Lido team have addressed these comments as follows:

1. The `objectionsAmountPct` is no longer saved to storage in the `Motion` struct.
2. This involves large changes to the existing UI, and is inconvenient from a UX perspective. The team will consider relevant changes in more detail in subsequent upgrades.
3. It is important that a factory can validate the script at the point of execution/enactment (as well as when created), to account for state changes such that the script is no longer valid.

   As an example, "a motion to increase the node operator staking limit might become non-executable if the node operator who created it was removed from the node operators registry after the creation of the motion."

   The suggestion, to pass the EVMScript as a parameter, greatly complicates this validation (and likely makes it infeasible, such that no gas is saved).
4. This has been considered, and the increased logical complexity was deemed to outweigh the potential gas savings.

   The testing team concurs that a simple implementation is less likely to contain bugs and more easily understood by third parties.
5. The development team intends designed EasyTrack to be used by common, non-controversial proposals which, in the usual "happy" scenario, should have no objections.

   As such, they prioritize the gas efficiency of motion creation over objection. Similarly, making the first objection more expensive than subsequent ones may introduce minor but undesirable incentives.

LET-08	Testing Findings And Suggestions
Asset	`tests/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details findings relating to the test implementation, using the brownie framework, which do not have direct security implications.

Also refer to additions and modifications to the tests provided to the development team alongside this report.

1. **Missed edge cases/coverage:** The following useful test cases were identified that are not currently covered:
    - Cases for `EVMScriptPermissions` with valid and invalid EVM Script inputs of different lengths. In particular, for `isValidPermissions` tests, a valid `permissions` list containing more than 1 permission, invalid `permissions` longer than 1 permission but shorter than 2, and `permissions` lists containing duplicate entries (to be explicit duplicate entries should be treated as valid or not).
    - The tests currently don't exercise the `ContractProxy` by default. Arguably, the default tests and fixtures implemented in the `conftest.py` should mimic the real-world deployment. While unit tests against the underlying `EasyTrack` contract are reasonable, the vast majority of tests use interact directly with the `EasyTrack` contract, with only a few specific tests using the `ContractProxy`. An alternative option would be to have 2 different test directories, with the `conftest.py` for "unit" test using mocks for contracts not in focus, and the "integration" `conftest.py` providing contract fixtures mimicking the intended deployment.
2. **Test isolation & EVM Snapshots:**

    The tests fail to make use of the powerful and developer–friendly brownie isolation fixtures, greatly limiting performance, and increasing the risk of tests affecting other tests' results and validity.

    In order to prevent tests from affecting each other, most fixtures in the current test implementation are "function scoped" (meaning they execute at the start of every test function), so new contracts are re-deployed at the start of every test function. This greatly increases execution time, and it's still prone to developer error (e.g. with account balances needing to be manually reset before each test).

    The brownie isolation fixtures provide a developer–friendly means to snapshot and rollback the underlying test EVM between test functions and modules.[6] This makes it possible for slow contract deployments (and other setup) to only be done once per test module, and developers can be confident that tests are properly isolated from each other. Similarly, this makes it safe for tests to be run in parallel with pytest-xdist, allowing execution time to be greatly reduced.

    The performance limitations can also encourage less exhaustive tests, sub-optimal test style (in which a several independent checks are made in a single test), and can otherwise hinder the development workflow.
3. **Opportunities to use pytest fixtures that return constants instead of global variables:**

[6] Explained in more detail: `https://eth-brownie.readthedocs.io/en/stable/tests-pytest-intro.html#isolation-fixtures`. A detailed example at `./tests/sigp-easy-track/tests/isolation_example/` was provided to the development team alongside this report.

For example, in `test_evm_script_permissions.py` the `VALID_PERMISSIONS` global variable can be replaced with a fixture that returns a valid permission.

The more powerful fixtures can then be parameterized to return one of several values, immediately covering more edge cases wherever used, or overridden at different levels to modify initial test state without needing to redefine a whole hierarchy of fixtures.[7]

4. **Tests making multiple checks that should be separated:**

   The testing team notes that it is preferred practice that test functions check only one conceptual "action". While this check may involve multiple `assert` statements, they should all be about the same action (e.g. that both a transaction's return value and emitted logs are as expected).

   The current test functions will often make multiple independent asserts within a single test, which should preferably be split into separate test functions. For example, `test_is_valid_permissions()`, it makes 3 independent calls to `EVMScriptPermissions.isValidPermissions()` with different inputs. If the first `assert` fails, the later checks are not evaluated; so it is not clear from the test results whether whether 1 or 3 things are wrong with the `isValidPermissions()`.

   It is likely that this was done for performance reasons (to avoid the slow setup between tests) but the EVM snapshots (discussed in item 2) can avoid this problem.

   Test parameterization can make it easy to implement separate tests with various inputs without rewriting the code.

5. **Tests asserting the order in which inputs are validated:**

   Several tests were identified that, likely unintentionally, make assumptions about the order in which the contracts verify input. This can be an internal implementation detail that, if modified, would cause these tests to fail unnecessarily.

   For example, in `test_add_evm_script_factory_called_without_permissions()`, there is more than one thing wrong with the input (empty `permissions`, and an invalid sender). The test then asserts that an error relating to the invalid sender is returned, when it could easily be one relating to the invalid `permissions`. Unless intentional and the order of these requirements is important, it is usually preferable to provide invalid input where only 1 thing is wrong with it.

6. At `test_evm_script_permissions.py:62`, the input to `isValidPermissions()` (shown below) is likely a different value than intended.

   ```
   assert not evm_script_permissions_wrapper.isValidPermissions(
     b"11223344556677889911"
   )
   ```

   When passed to the contract, the value `b"11223344556677889911"` is actually 20 bytes long and equivalent to the solidity `bytes` value `hex"3131323233333434353536363737383839393131"` (the ASCII/Unicode encoding of the numbers `112...`), not the 10 byte `hex"11223344556677889911"`. It appears that the string value `"0x11223344556677889911"` is actually the intended input.[8]

   In this case, the test is still ok because either the input is invalid but this misunderstanding could lead to bugs elsewhere. Confirm whether this is intended and adjust accordingly.

7. The imports in the brownie `conftest.py` at lines [5–21] are unnecessary, as all brownie `ContractContainer` instances are available as pytest fixtures.

   Instead, these can be listed as fixture dependencies in any fixtures that require them.

[7] Refer to the modified `test_evm_script_permissions.py` provided alongside this report, and the relevant pytest documentation: `https://docs.pytest.org/en/latest/how-to/fixtures.html#parametrizing-fixtures` and `https://docs.pytest.org/en/latest/how-to/fixtures.html#overriding-fixtures-on-various-levels`.

[8] This is converted by brownie to the equivalent python byte value `b"\x11\x22\x33\x44\x55\x66\x77\x88\x99\x11"`

8. A nitpick at `utils/evm_script.py:21-22`. The current code (shown below) works with `int256` because the input to `encode_single` is always positive:

   ```
   length = eth_abi.encode_single("int256", len(calldata_bytes) // 2).hex()
   result += addr_bytes + length[56:] + calldata_bytes
   ```

   but is more clearly written using `uint32`:

   ```
   length = eth_abi.encode_single("uint32", len(calldata_bytes) // 2).hex()
   result += addr_bytes + length + calldata_bytes
   ```

   This more clearly reflects the CallsScript encoding.[9]

9. The brownie report exclusion settings can be more easily and reliably configured in `brownie-config.yaml` via the `exclude_paths` field (shown below), rather than needing to manually add all test contract names to the `exclude_contracts` field.

   ```
   exclude_paths:
   - contracts/test/**/*
   ```

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

This has been addressed (primarily in PR #9) as follows:

1. - ee5835d — More tests cases added.
   - 96479e8 — `ContractProxy` removed (see LET-01), so no need for tests.
2. adc4473 — resolved, using isolation fixtures and improved fixture scoping.
3. Used where relevant and more convenient than global variables.
4. ee5835d — Tests split up and parameterization used.
5. ee5835d.
6. d7c0b24.
7. adc4473.
8. ee5835d.
9. ee5835d.

[9] Defined at AragonOS CallsScript.sol.

LET-09	Miscellaneous Findings
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. Although the specification (`specification.md`) discusses upgradability, it makes no direct mention of the `ContractProxy` ERC1967 proxy.
2. **Minor Gas Optimizations:**
   - In `BytesUtils.sol` the `div` can be replaced with the more gas-efficient `shr` instruction (and an appropriately adjusted value), though the real-world gas savings would be minimal unless used in bulk.
   - Duplicate entries in a permissions list are allowed by `EVMScriptFactoriesRegistry.addEVMScriptFactory()` , allowing unnecessary gas costs when processing associated scripts.
     This is noted purely to ensure awareness. Any deduplication is likely not worth the additional gas costs, particularly as the entity proposing new factories to the DAO is likely also involved in enacting associated motions.
3. **List of Typos:**
   - At `RewardProgramsRegistry.sol:86` , `_gerRewardProgramIndex` should be `_getRewardProgramIndex` .
   - In `utils/evm_script.py` , the function `encode_call_script` is more appropriately named `encode_calls_scri` (to reflect the `CallsScript.sol` naming).
   - At `AddRewardProgram.sol:65` , the NatSpec comment for `decodeEVMScriptCallData()` is incomplete (shown below):

     ```
     /// @param _evmScriptCallData Encoded tuple: (address _rewardProgram)
     ```

     According to the spec[10] and code, this should be `(address _rewardProgram, string memory _title)`
   - The `EvmScriptExecutor.sol` file should be renamed to `EVMScriptExecutor.sol` to match the capitalization of its contained contract.
4. **Comment on EasyTrackStorage state variable ordering:** Note that the current organization of state variables within `EasyTrackStorage` is unsustainable with future upgrades.

   Although it is more legible to organize into different sections, this will break down if more state variables are added (because these need to be added to the bottom to retain a consistent storage layout). For example, a new variable relating to “motion setting” must be put at the bottom, not in the “motion setting” section.
5. **Constructor validation and sanity checks:**
   - There is no validation of `trustedCaller != 0` . Consider modifying the `TrustedCaller` constructor to include this check.

[10] See spec.

- Consider validating the `EVMScriptExecutor.constructor()` arguments `_callsScript`, `_easyTrack`, `_voting` to check that they aren't zero or are a contract.

6. **Small, Nitpick Naming Suggestions:**

    - The `EVMScriptExecutor` name could be slightly misleading. Although descriptive, the name could be misunderstood to suggest it implements the AragonOS IEVMScriptExecutor interface, which it does not.
      Consider renaming it to something like `EasyTrackScriptExecutor` to help distinguish this.
    - Consider adjusting the variable names in `RewardProgramsRegistry` to highlight that the expected `TrustedCaller` account is the `EVMScriptExecutor` instance (rather than an EOA like for the other contracts). For example, renaming the `_trustedCaller` constructor parameter to `_trustedScriptExecutor` would be more descriptive for readers.
    - In `EVMScriptPermissions._getNextMethodId()`, the `uint32 methodId` local variable name is confusing and clashes with the understanding that the function returns a `bytes24 methodId`.

      Consider renaming this to something like `functionSelector`, to highlight that it is the function selector and not the whole `methodId`.
    - At `EVMScriptPermissions.sol:35`, the use of the `PERMISSIONS_LENGTH` constant is somewhat incorrect.
      While the value of 24 is correct, the code is traversing along the evmscript, not the permissions. In this case, the length (24) is meant to represent the size of an `address + bytes4 calldata_size`, not a permission consisting of `address + selector`.
      Consider using a different constant name here, that is more descriptive for the context.
    - The `EVMScriptPermissions` contract contains a lot of "magic numbers".

      Consider replacing these numbers with more descriptively named constants, to better describe how calls scripts are encoded.

7. It may be desirable to add an extra `createEVMScript()` overload to `EVMScriptCreator` that allows multiple calls to different methods. That is, one with a signature like the following:

    ```
    function createEVMScript(
        address _to,
        bytes4[] _methodIds,
        bytes[] memory _evmScriptCallData
    ) internal pure returns (bytes memory _evmScript) {
    ```

    This can save needing to deploy a modified library should a later script factory wants to execute more than one different function.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

This has been addressed (primarily in PR #6) as follows:

1. 96479e8 — `ContractProxy` removed (see LET-01), so no need for inclusion in the specification.
2.
   - 1c1e140
   - New factories with duplicate permissions entries could be rejected by the voting DAO, like it should for the addition of a malicious factory.
3. Relevant fixes in a37531f & 39d635d.
4. With `EasyTrack` no longer upgradable, the `EasyTrackStorage` contract has been removed and storage ordering is no longer relevant.
5. Resolved in 4ef4b5a & aee9b60
6.
   - Magic numbers and relevant naming suggestions resolved in e8a0826.
   - The naming of `EVMScriptExecutor` will be retained. It has been chosen to highlight that EasyTrack uses the same format as the Aragon CallsScript. Renaming would involve significant changes across the project with minimal, if any, benefit.
   - `RewardProgramsRegistry` now inherits `AccessControl` instead of `TrustedCaller`.
7. 0d57936 — additional `createEVMScript()` overloads allowing the creation of scripts involving multiple calls to different functions in a contract, and multiple calls to different functions in different contracts.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Bogdan Kovtun and Gregory Stepanov. Easy Track Specification. Website, 2021, Available: https://github.com/lidofinance/easy-track/blob/master/specification.md. [Accessed September, 2021].

[2] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[3] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].