Mimic v3-core code review

by Nicolás Venturo (nicolas.venturo@gmail.com)

This document contains a review of smart contracts developed by Mimic Finance. Their source code can be found in the `v3-core` GitHub repository.

The review was conducted on commit `53c2a5159d515f9a458b01d367ead3f2c670ddf2`, from October 31st, 2023. The contracts under review were `Relayer`, `Authorizer` and `SmartVault`.

`Relayer`

Strong suggestions

- In L99 `setSmartVaultCollector` forbids a collector being the zero address. However, on L73 a zero address is used to signal a smart vault wishes to use the default collector. This means that once a smart vault sets any collector, it is impossible to revert to the default mechanism. Remove L99, or alternatively add a `clearSmartVaultCollector` function.
- In L237 task execution halts if a task call fails and `continueIfFailed` is not set. However, the `taskResults` return value does not indicate which tasks were skipped: the last array values will be left uninitialized. While it is possible to determine which tasks were called and which were not, this logic can be a bit convoluted and error-prone (e.g. if `continueIfFailed` is not set, then the first failed task was executed but the following ones were not). Return a second numeric value with the number of tasks that were actually called, or alternatively add a `called` flag to the `TaskResult` struct.
- In L227 `task` is validated by calling `hasPermissions` on the smart vault. Current smart vaults return true if the task has any single permission over them, meaning that granting permissions to an account has the implicit side effect of allowing it to spend the smart vault's gas. Remove `hasPermissions` (or rename it to e.g. `hasAnyPermission`), and instead of calling that in L227 call a new `isTaskAllowed` function or similar. Note that this can be made backwards compatible with existing smart vaults by calling into a new task registry contract instead of the smart vaults themselves.

Comments on the `_execute` function

The main purpose of the relayer is to call other contracts associated with a smart vault via `_execute`, paying the smart vault's collector for the gas spent. No issues were found in this function, but it is important to point out that there are many moving pieces here, and small refactors could easily lead to issues. It would be good to document or otherwise bring attention to the following points, hopefully reducing the likelihood of errors in the future.

- L227 is the only validation that is made on `task`, by checking that the smart vault that will pay for the gas is somehow related to it. It might not be immediately obvious that this check is critical since the smart vault is not referenced again in the `_execute` function (only inside `_payTransasctionGasToRelayer`).
- The logic in `_payTransactionGasToRelayer` is relatively complicated, and as a result a critical property is difficult to prove: the relayer never sends any funds in excess of the smart vault's balance, i.e. `amount`

`- quota <= balance`. If this did not hold, the owner would be able to steal funds by setting arbitrarily large values for `maxQuota`.

- In L231 there's a raw call to `task` with no value. If `task` were an EOA, this would succeed but achieve nothing, a false positive that might be difficult to detect. This is correctly but indirectly prevented by L224, where `smartVault` is called on `task`, which will revert unless `task` is a contract.
- The relayer itself should never have permissions over anything, since it can be made to call (almost) arbitrary contracts. There's some mitigation in the fact that `task` must provide a meaningful return value for `smartVault`, but it'd still be good to make sure that relayers are not given any special treatment.
- Related to the comment above, the relayer is a sort of special account as it is expected to hold native tokens. These could be trivially stolen by any executor if value was sent in the call in L231.
- Similarly, any tokens held by the relayer could be stolen by transferring them in L231. This is prevented as it is expected that a token contract will revert when the `smartVault` function is called on them, which is a side-effect of the intended behavior (checking that the smart vault will pay for the gas of a call to `task`).

Minor comments

- The gas spent in emitting the `TaskExecuted` event is not repaid, as `gasleft` is called before it (to compute `gasUsed`). The cost of the event could be non-negligible if either `data` or `result` are long.
- On L258 the `quota` variable is left uninitialized if `balance < amount` and the `else` branch is taken. Solidity initializes variables to their default value (0 for numeric values), but it'd be good to do this explicitly.
- There's an unnecessary command to disable solhint on L240, likely a remnant from a previous refactor.

`Authorizer`

Strong suggestions

- The functions `hasPermissions(who, where)` and `hasPermission(who, where, what)` look very similar and can easily lead to errors that would be undetected by most tests. Rename `hasPermissions` to `hasAnyPermission`, or remove it.

Comments on the permission scheme

The system seems to trade a large amount of complexity for little gain in terms of usability. This added complexity makes it hard to verify correctness, and at times introduces difficulty when attempting to understand basic behavior.

An example of this is the `authorize` function, with arguments `who`, `where`, `what` and `params`, plus a `how` variable. This calls `_authenticate` with some values, which in turn calls `isAuthorized`. `isAuthorized` also has `who`, `where`, `what` and `how` parameters, but none of these four values are the same as the ones with the same names in the original `authorize` call context. It is easy to get lost and to have to resort to note-taking to follow what is going on, greatly reducing the reader's confidence in their own understanding.

Another function that is hard to follow is `_evalParams`, with the `if` statement in L257 being very logic-dense. Once unrolled, certain aspects of this check remain puzzling. Examples are the scenario in which `params` is longer than `how`, but then all further params must equal `Op.NONE`, or the one in which `how` is longer than `params` causing some arguments to not be evaluated.

The benefits provided by this scheme are also quite limited and don't provide a lot of functionality. Parameter evaluation is limited to uints, making other types (including ints) cumbersome to use. This required the introduction of the `AuthorizedHelpers` contract, which provides multiple conversion functions, but is itself very error-prone.

Additionally, even if type support was not an issue, the operations that `_evalParam` can evaluate are quite limited. Examples of common use cases that are not possible to implement using this scheme include comparisons between parameters (e.g. two paramters must be the same, or one must be larger than another), comparisons to some global state (e.g. a timestamp must be in the future), and frequency restrictions (e.g. a permission can only be used once a week).

The system as a whole would likely benefit from reduced complexity in `Authorizer`, with the removal of the generic `_evalParams` and the introduction of specialized checks where required (e.g. in `authorize`). Permissions that require conditions on the parameters of the function call can be more cleanly implemented by introducing permissioned contracts with ad-hoc checks that restrict their behavior.

`SmartVault`

Strong suggestions

- In L76 the constructor does not call the `_disableInitializers` function. This is not an issue as the parent contract `Authorized` already calls it, but changes to either `Authorized` or the inheritance tree could easily result in the initializers not being disabled on the implementation contract. Add this missing call, since as of v5.0.0 of the OpenZeppelin library disabling initializers multiple times is not an error.
- The `hasPermissions` function is error-prone and easy to misuse. Rename it to `hasAnyPermission` or remove it entirely.

Minor comments

- In L106 `__SmartVault_init_unchained` ignores its first parameter, making the reader wonder why it's there in the first place. It can be safely removed.
- In L206 the delegate call can trivially override all of the other security mechanisms in the contract, including disabling the reentrancy guard, withdrawing funds, upgrading the contract, and even do things that are otherwise not possible, such as changing the authorizer. This makes the permission to call `execute` much more powerful than any of the other permissions, and results in `_validateConnector` being a critical safeguard and `overrideConnectorCheck` very dangerous.
- While `pause` does correctly prevent calls to all state changing functions, including `execute`, it does not prevent contract upgrades. This behavior must be explicitly included in the upgrade mechanism.
- Most functions are non-reentrant, but this seems unnecessary as all of them are quite simple and trivially follow the checks-effects-interactions pattern.