MultiversX Multisig

MultiversX smart contract - Security audit by Arda

Repository: https://github.com/multiversx/mx-contracts-rs
Smart contract path: *contracts/multisig*
Initial commit: *28d617fa761c7c8c057f56aa13eef0862fb2c47b* (repository link)
Final commit: *5478c6ed3c8cc998429b2d83f97347c553be8f87* (repository link)

Issues

A.1. Solved Major **Any board member can add actions to an already existing batch, allowing him to (1) prevent the batch from ever being executed (by adding actions to it) (2) possibly making unwanted action be executed (by front-running other board members and adding actions before they sign).**

Solution: Once a batch is created, we may forbid adding further actions to it.

A.2. Solved Medium **Any board member can prevent an accepted cross-shard action from being performed.** This is because if he performs the action and provides a very small gas quantity, the async call will fail on the other shard, and the call won't revert, so the action has been cleared from storage and can't be performed again.

Note that a non-malicious board member could also simply make a mistake in the gas quantity needed for the asynchronous call to be executed, leading to the same problem.

Solution: We suggest adding an optional minimum gas field in the action, which will be checked in `perform_action` before launching the asynchronous call. In particular, this would be useful for upgrading contracts, as upgrades are asynchronous calls as well.

A.3. Solved Medium **If the quorum is decreased, undesired and outdated actions can suddenly be performed.** Indeed, upon performing an action, `quorum_reached` checks that the number of signers is above the current value of the storage `quorum`.

So for example if an action which did not reach the quorum one year ago was not discarded since then and forgotten, and now the quorum is decreased, it would become possible to perform that action even if it is undesired.

Solution: We suggest making the quorum an internal field of the action, recorded when the action is created, and used when trying to perform it.

A.4. Solved Medium The endpoint `sign_batch_and_perform` will fail if it tries to execute the actions, because of the following iteration:

```
None
for action_id in self.action_groups(group_id).iter() {
   let _ = self.perform_action(action_id);
}
```

Namely, at each iteration, inside `perform_action` the action is cleared from `action_groups(group_id)`, reducing the size of this mapper, hence the iteration will reach some empty actions, and performing them will fail.

Solution: Before performing each action, we recommend storing all the `action_id` from `action_groups(group_id)` in a `ManagedVec` variable, and then in a second loop, we can perform them.

A.5. Solved Medium **Error-prone argument `group_id` of `propose_batch`.** Users must choose a `group_id` to create a batch, which may not be evident, and there is also the risk that the group id is mistakenly chosen to be the one of an existing batch even though the user wanted to create a new batch.

Solution: If the solution to issue A.1. is followed, i.e. users can't add actions to existing batches, then we can simply remove the `group_id` argument and automatically generate this id from a counter `counter_group_id` which gets incremented each time a batch is created.

In case the solution to issue A.1. is not followed and users can still add actions to existing batches, then we can make the `group_id` argument optional:

- If not provided, we automatically generate a new id using a counter as in the previous paragraph.
- If provided, we check that the `group_id` corresponds to an existing batch, and we add actions there.

A.6. Solved Medium **A `SendTransferExecuteEsdt` action will fail to be executed if no gas limit was provided,** because it consumes all the gas left as gas limit, which immediately leaves the contract since we are doing a `TransferExecute`, and thus there is no gas left to terminate the contract's call.

Solution: We suggest proceeding as for `SendTransferExecuteEgld`, i.e. when there is no gas limit provided, we use a default value of `gas_for_transfer_exec`. Moreover, we suggest adding a unit test where a `SendTransferExecuteEgld` action with no gas limit is performed.

A.7. Solved Medium **An endpoint required from the functional specifications is missing in the implementation.** Namely, it is expected that the multisig has an endpoint that board members can call to sign and, if possible, execute the action (or batch) in the same transaction. However, `sign_and_perform` and `sign_batch_and_perform` do not fulfill this purpose as they fail if the action or batch can't be performed.

Solution: In both `sign_and_perform` and `sign_batch_and_perform`, we suggest performing the execution only in case the caller has the right to perform actions and all actions have reached quorum.

A.8. Solved Medium **In `sign_batch_and_perform`, some actions of the batch might be executed and others not**, although the user expects that all actions of the batch are executed, or none of them. This is because the following iteration will execute actions which have reached quorum and do nothing for the others:

```
None
for action_id in self.action_groups(group_id).iter() {
    if self.quorum_reached(action_id) {
       let _ = self.perform_action(action_id);
    }
}
```

Solution: We recommend first verifying in a 1st iteration that all actions have reached quorum, and then, if it is the case, performing all the actions sequentially in a 2nd iteration.

A.9. Solved Medium **It is expected from functional specifications that no asynchronous call can be included in a batch, but an upgrade action can be any action of a batch.** This is problematic since contract upgrades are asynchronous calls. In particular, if an upgrade action is included in a batch, it would kill the execution in the middle of the batch.

Solution: When creating a batch in `propose_batch`, we suggest also checking that each action is not a contract upgrade.

A.10. Solved Medium `get_pending_action_full_info` **will fail if too many actions have been created in the multisig**, because this view must iterate over all past action indices:

```
None
let action_last_index = self.get_action_last_index();
for action_id in 1..= action_last_index { ... }
```

Solution: We suggest adding an optional `offset` argument to `get_pending_action_full_info` to iterate over a fixed number of past actions.

Optionally, we can also add another view function that takes actions indices `action_id_start` and `action_id_start` to iterate within a specific range of action ids.

A.11. Solved Medium **Actions within a batch are not necessarily executed sequentially.** This is because the check that the called contract is intra-shard, which is present for a `SendTransferExecuteEgld` action, is absent for a `SendTransferExecuteEsdt` action. If the action calls a contract on a remote shard, the action would be executed after subsequent synchronous actions.

Solution: In `propose_batch`, we recommend adding the check that the called contract is on the same shard for a `SendTransferExecuteEsdt` action exactly as done for a `SendTransferExecuteEgld` action.

A.12. Solved Medium **The multisig can't work anymore if it has too many board members.** Indeed, performing actions would run out of gas when iterating over board members.

Solution: We suggest imposing a hard bound on `num_board_members` e.g. `MAX_BOARD_MEMBERS = 30`.

A.13. Solved Medium **An action which is part of a batch can be discarded although other actions remain in the batch, possibly making users sign unexpected batches.**

Example: Alice and Bob share a multisig, which manages SC A (holding Alice's funds) and SC B (holding Bob's funds). Alice wants to steal Bob's funds:

1. Alice creates a batch with 2 actions representing an exchange of funds: first a transfer from SC A to Bob, and second a transfer from SC B to Alice.
2. When Bob is about to sign, Alice front-runs his transaction: she unsigns only the 1st action (transfer from SC A to Bob) and discards the action.
3. Bob's signature arrives to the contract.
4. Alice executes the batch, which transfers Bob funds (in SC B) to Alice.

Solution: Once an action is discarded from a batch, we recommend considering the batch as aborted and not allowing performing it. For this we can associate a boolean `batch_aborted` to each batch which when `true`, makes signing and performing the batch fail.

A.14. Solved Medium **Upgrading the contract with a new `quorum` and `board` value bypasses the multisig consensus.** This is because `upgrade` changes the value of the quorum and whitelists new board members.

Solution: We suggest not overwriting the quorum and not changing the board members in `upgrade`.

A.15. Solved Medium **If there are too many signers for a proposal, it can't be performed.** This is because `get_action_valid_signer_count` iterates over all signers, including old board members who were removed.

Solution: We suggest adding a public endpoint `unsign_for_outdated_board_members` which takes a list of `action_id` and addresses, check that the addresses have no role, and then remove their signatures from `get_action_valid_signer_count(action_id)`.

A.16. Solved Minor **Missing checks on actions when creating a batch.** When creating a batch, the checks on actions are not consistent with the checks made when creating the action without a batch:

- For a `SendTransferExecuteEsdt` action, no check is made within a batch, although in `propose_transfer_execute_esdt` it is checked that the amount of tokens to send is non-zero.
- For a `SendTransferExecuteEgld` action, no check is made within a batch, although in `propose_transfer_execute_esdt` it is checked that either the EGLD amount to send is non-zero, or that the contract call is not empty.

Solution: In `propose_batch`, we suggest adding the missing checks which are made when proposing an action without a surrounding batch.

A.17. won't solve Minor **No time limit to perform actions: an outdated action can be performed 1 year after it is proposed in case it reaches quorum later.** Indeed, as long as the action is not discarded, which requires *all* signers to withdraw their signature, an action can be triggered.

However, certain actions may be undesired if they occur too long after they were proposed, e.g buying a token 1 year after the initial proposal would lead to a completely different swap output than expected.

Similarly, when a board member is added, he is able to vote on past proposals, which is not expected such proposals were created given the quorum and board members at a past time, and might not be relevant for the new board member.

Solution: We could add a time limit for performing the action, as an internal field of the action.

A.18. Solved Minor **Storage key prefix of the other.** The storage name *"user"* is a prefix of another storage name *"user_role"*, which is bad practice as in certain situations it can lead to storages overwriting each other (see MultiversX doc).

In commit 0fd235fe4c8491ab086c4fd0903788d169f02583, a new prefix was introduced: "quorum" is a prefix of "quorum_for_action".

A.19. Solved Minor **3 Mandos tests do not pass.**

Solution: We suggest correcting these 3 tests.

A.20. Solved Minor **Syntax inconsistency.** There is some syntax inconsistency in `propose_transfer_execute_esdt`. Unlike other endpoints to propose actions which first define an instance `call_data` of the struct `CallActionData` before passing it to the method `propose_action` (like `propose_transfer_execute` and `propose_async_call`)

```
None
let call_data = CallActionData { ... };
self.propose_action(Action::SendAsyncCall(call_data))
```

`propose_transfer_execute_esdt` rather directly build the action as the argument of the method `propose_action`:

```
None
self.propose_action(Action::SendTransferExecuteEsdt { ... })
```

This syntax difference is a bit misleading and having a consistent way to write all endpoints to propose actions would simplify the understanding of the code.

Solution: In propose_transfer_execute_esdt, we suggest first defining `call_data` and providing it as an argument to `propose_action`.

Disclaimer

The report makes no statements or warranties, either expressed or implied, regarding the security of the code, the information herein or its usage. It also cannot be considered as a sufficient assessment regarding the utility, safety and bugfree status of the code, or any other

statements. This report does not constitute legal or investment advice. It is for informational purposes only and is provided on an "as-is" basis. You acknowledge that any use of this report and the information contained herein is at your own risk. The authors of this report shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.