CertiK Auditing Report

for

Revision Date: Mar. 17, 2019

Disclaimer

This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Auditing Services Agreement between CertiK and MYKEY (“the Company”), or the scope of services / auditing, and terms and conditions provided to the Company in connection with the auditing (collectively, the “Agreement”). This Report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

Summary

This audit report summarises the smart contract auditing service requested by MYKEY. The goal of this security audit is to validate that the audited smart contracts are robust enough to avoid potential security loopholes.
The result of this report is only a reflection of the source code that was determined in this scope, and of the source code at the time of the audit.

General Understanding and Assumptions

1. The **main design goal** of the manager / logic dual-contract architecture is to **detach** action logic from action access / persistent data storage, so that the latter does not need upgrades and can remain stable while the client applications have zero dependency on the former, which thus can be easily upgraded by itself. Having all persistent data anchored in a stable, non-upgradable contract has great implications on data safety and security.

2. The manager and logic contracts are **mutually trusting**. Any isolations between them are a result of the main design goal, rather than for security, performance, or any other considerations. There is no need to protect the manager contract from a possible future adversary (upgraded) logic contract. Since the manager contract is the invariable part, future logic contracts are responsible for resolving any potential API or semantic conflicts with it.

3. The current architecture will not be used for sharded parallel execution when it becomes available in the future. Naive adaption to do so will likely result in either performance bottleneck or new concurrent security issues.

Design comments

4. The persistent data storage in the manager contract is highly specialized and their access functions are split between the manager and logic contracts. There are six different global persistent tables, each with its own set of modification functions defined

in the manager contract (and the read / lookup and other pre-processing function defined in the logic contract).

- *keydata_table: addkeydata(), mdfkeydata()*
- *backupdata_table: setbkpdata(), removebkp()*
- *defdata_table: setdefdata(), removedef()*
- *propdata_table: setproposal(), removeprop(), setapproval(),*
- *subacct_table: initsubacct(), rmsubacct(), tosubacct(), tomainacct(), mdfsubkey()*
- *subassetsum_table: tosubacct(), tomainacct()*

This **goes against** the above stated main design goal (1) of detaching action logic from action access and persistent storage in the following ways.

- The table entry types is built into the manager contract, which would make it hard to upgrade / expand these types without upgrading the manager contract synchronously.
- New tables might be needed, which cannot be added to the manager contract without also upgrading it.
- New semantics to some of these table modification functions might be needed in the future, but that will be impossible without modifying the manager contract itself.

Hence it is not clear to what extent has the current implementation realized the main design goal (1). To address this problem, CertiK recommends trying to use **generic untyped tables** instead of the specialized ones and move all specialized modification semantics from the manager contract into the logic contract in the following steps.

- First, define generic bytes table types and their generic modification functions such as *add()*, *modify()*, and *remove()*. Considering the specialized tables have different indexing scheme, it might be necessary to define multiple types of such generic untyped tables with different indexing scheme.

- Second, move the existing specialized table modification functions from the manager contract and merge them into their callers in the logic contract. Instead of calling the specialized table modification functions from the logic contract, pack the table data into bytes and instead call the generic table modification functions as defined in the manager contract.

- Third, change the logic contract existing table read / lookup functions so that they read out from the generic tables instead of the specialized ones, and do explicit unpacking of the generic bytes read out.

- Fourth, move the existing manager account creation functions into the logic contract.

Ideally there should only be a few generic table types and their generic modification functions defined into the manager contract. This will greatly reduce the complexity of the *manager* contract and stay truer to the main design goal (1).

If, for schedule or other considerations, MYKEY wants to continue releasing the first version as is, it is recommended to define the generic untyped tables and their generic modification functions in the manager contract, while leaving the actual usage of them as well as the read / lookup / pack / unpack functions for future work in the logic contract alone. This would allow a smooth non-disruptive migration from the special tables to generic tables in the future.

Correctness comments

5. *verify_sig()* can only be called once per signature as the *nonce* is bumped up in it. It is not clear the benefit of doing so, comparing to bumping up the *nonce* at the end of, e.g., *sendinternal()*, *sendexternal()*, etc. If the decision is intentional, consider renaming it to *verify_sig_destructive()* or something similar.

6. *setproposal()* invalidates all approvals. If the data has not actually changed, consider retaining the existing approvals.

7. We recommend adding explicit checking of permitted actions at the beginning of most functions, similar to what is done in *approveprop()*.

8. *In check_approval()*, the expression "*cnt * 6 / 10.0*" is prone to future **overflow** as cnt is only 8-bit. We recommend using safer expressions such as "*cnt * 0.6*".

9. *forward()* was previously written as synchronously done but has since been refactored to be asynchronous. So the caller of *sendaction(sendsubextnl|sendexternal)* will have no way to tell if the call to the external contract has actually happened or not, and, if yes,

was successful or not. This **seriously limits** the kind of useful dapps that utilize these functions.

10. In *updatelogic()*, if logic A and B are updated in the order of “A, B, A” while all of them are still pending, the second update of A will be **silently lost**, which is unexpected for callers, as normally one expects “update A, update B, update A” will eventually result in A being the final active logic. Throw exception here upon updating over pending updates of the same logic.

11. In *cancellogic()*, only the centralized logic update account can cancel pending logic contract updates. This differs from what CertiK has been previously told, i.e., "client has final control of logic contract update".

12. In *propose()*, it will need an *executeprop()* later to actually execute the proposal. This is **different from the design doc**, which says the proposal should take effect immediately.

13. In *bkppropose()*, it will need an *executeprop()* later to actually execute the proposal. This is **different from the design doc**, which says the proposal will have a 30-day delay before it gets executed (automatically).

Performance and style comments

14. Use constants and enums for key index, key size, key status, etc. so that their values are abstracted away from the code. Both the *manager* and *logic* contracts should refer to the same set of constants in order to avoid divergence.

15. Refactor *createacct()* and use *split_keys()* to split account name and backup name in addition to keys. This will help simplify the code by removing all the manual extraction bits, making the code clearer and more robust.

16. Do the *nonce++* operation via a light-weight dedicated *passaction* act. If the design comments of turning *manager* contract tables into generic ones is followed, consider providing a generic partial-update / nonce-bumping action.

17. Only modify the various table entries when the content is indeed different.

18. Rename *mdfsubkey()*, *addsubdapp()*, *rmsubdapp()*, and *rmsubacct()* in the manager contract in order to avoid confusion with related functions of the same names in the logic contract.

EOS Smart-Contract Typical Issues Audited

19. Issue: Action and code mismatches in *apply()* function.
    Result: No problem was found. The function itself is fine. However, since only *action == N(transfer)* is checked before the *on_transfer()* call, more analysis is needed for that function.

20. Issue: Fake EOS attack on on_transfer() function.
    Result: No problem was found. The function checks *code == N(eosio.token)*.

21. Issue: Fake EOS transfer attack on on_transfer() function.
    Result: No problem was found. The function checks *to != ACCOUNTMGR*.

22. Issue: Unlimited account creation by unrestricted accounts causing resource depletion.
    Result: No problem was found. *createacct()* has *require_auth(_self)*, hence only the contract owner can do it.

23. Issue: Internal functions being called without going through entry functions.
    Result: No problem was found. The required authentication of *LOGIC_CONTRACT*, *LOGIC_UPDATE_CONTRACT*, and *_self* ensures the proper invocation of the internal functions.

24. Issue: Arithmetic overflows.
    Result: No problem was found. *extended_assert* and *uint64* are properly used to avoid potential overflows.