GAMultisig smart contract
A Sophia contract on the Aeternity chain

Audit Report

Test Object:
GAMultisig smart contract code

Date: **April 19, 2022**

*This report presents the results of an audit performed for the **GAMultisig smart contract**. The Sophia code for this contract has been carefully reviewed, tested and analyzed for security vulnerabilities.*

The review of the code was performed in three stages from mid January 2022 until March 24, 2022. In the first two stages Quviq suggested improvements and slight modifications to make the resulting contract more efficient, more secure and easier to use.

The goal of this code review has been to make sure that the multi-signature schema implemented is first and foremost secure, but also that it is efficient and easy to use. Quviq did not find any critical issues in the multi-signature schema, but the design of the contract as a Generalized Account authorization contract were missing some checks that could have lead to tokens being extracted by a malicious miner. In the final contract all issues discussed in this report have been addressed.

We see that it was highly efficient to do the review in multiple stages - giving both the reviewers and the developers time to discuss, evaluate and adapt. We have been able to address several issues in an efficient manner using this approach.

Introduction

The reviewed contract is an authorization-contract for a Generalized account. The authorization is a multi-signature schema; upon creation the contract sets a (fixed) signing threshold (number of signatures required to authorize a transaction) and a (fixed) set of accounts that can co-sign. Any signer can propose a transaction for signing, and once enough signers have confirmed the transaction (hash) the transaction can be wrapped in a GAMetaTx and authorized by the contract.

Test object

The final GAMultisig software we have reviewed is retrieved from the master branch of the following github repositories with indicated SHA commit hash around March 18, 2022.

`https://github.com/aeternity/ga-multisig-contract/894588ac0b7a6130ecc687330a82d4cd87198a5b`

Note that the review is limited to the contract code. The surrounding code and supplied tests have not been reviewed.

The initial review focused on getting an understanding of the design, and scope of the implemented multi-signature schema. The initial review was performed on the source code **`contracts/SimpleGAMultisig.aes`** of the commit hash **`8ecaa029887c6679408f2a6ece25d07c640e1960`**.

During the initial review, we could see that the contract was unnecessarily complex, basically requiring two cryptographic signatures where one gives exactly the same level of confidence. Hence, the developers made a couple of simplifications to the implementation before the main audit. The main audit was performed for commit hash **`b79fc0eab9d487ac3b7bb4720e192d8df405387d`**. And finally, the follow up inspection was made for commit hash **`894588ac0b7a6130ecc687330a82d4cd87198a5b`**.

We only focused on functional correctness and potential security issues related to the contract. Possible user interfaces interacting with the contract are not part of this review.

Deployment assumptions

We have assumed that the contract will be compiled with the Sophia compiler [3] version 6 for FATE VM version 2. We expected the contract to run on Aeternity protocol Iris.

Methodology

We reviewed using two complementary techniques. Model based testing and manual code inspection.

Model based testing builds upon automatic generation of a large set of tests. The generation approach helps finding unforeseen software errors. These delicate software errors are often caused by a sequence of events that bring the system in an unexpected state. A tester will normally try to write complex tests that bring the system in such states, steered by the (limited) imagination of what may happen. In model testing we let the computer explore possible paths that may happen and generate tests in order to try to provoke such paths.

We made a state based test model using QuickCheck [1, 2]. In short this will allow us to generate wellformed test cases that we can run against the reviewed contract. The model specifies what each smart contract entrypoint call does, and allows us to chain together sequences of calls with a known expected outcome.
This is a powerful test technique and it gives us good confidence that the reviewed contract behaves as expected in many different scenarios.

For smart contracts, we typically generate tests for sequences of possible contract entrypoints. In this sense, the **`init`** entrypoint is also part of the test case and defines the initial configuration of the contract. For the GAMultisig, we tested any combination of M out of N accounts to sign, with N between 1 and 8 accounts. After initialization, the tests generated have an arbitrary number of `propose`, `confirm` and `revoke`. Thus even sequences in which the same user confirms and revokes several times. Anywhere in such sequences a meta transaction can be created, which of course should only be authorized if sufficiently many signers have confirmed. A successful meta transaction is then followed by new random sequences of propose, confirm and revoke.

Thousands of tests are generated and ran against this contract, covering clearly all the contract code, but more importantly all kind of weird combinations of endpoint calls.

We also performed a manual code inspection with several pairs of eyes. Here we reasoned about the contract in two steps, firstly on a protocol level, looking for potential problems with the overall usage and flow of calls and loopholes in general. In the second step we looked at each entrypoint in detail, checking for problematic inputs and unwanted behavior. We also looked for inefficient and/or unclear code, that makes it harder to understand, and possibly more expensive to run.

Testing and manual code inspection cannot guarantee to find all possible problems and security vulnerabilities, as mentioned in the Disclaimer, but by following a systematic approach we have made our review as thorough as possible

in the given amount of time.

Severity classification

We use the following classification scale for the issues we encounter.

- P1: Highest priority; loss of funds, deadlock of contract, hijack control, minting of tokens.
- P2: Severe disruption; No funds technically lost, DDOS, unusable protocol for extended period.
- P3: Inconvenient, or unexpected behavior of the protocol; loss of fund due to user error, easy to make mistakes, excessive gas usage.
- P4: Code organization, performance, clarity, gas consumption.
- P5: Cosmetic / Documentation.

Disclaimer

In this report Quviq presents an informational exercise documenting the due diligence involved in the secure development of the Sophia smart contract, and make no material claims or guarantees concerning the contract's operation post-deployment. Under no circumstances Quviq can be liable for any direct or indirect consequences arising out of the deployment or use of this smart contract.
This report makes no claims that its analysis is fully comprehensive.There always is a possibility of human error in the review process or a contextual change in the area in which the contract operates. We recommend always seeking multiple opinions and audits.

This report does not constitute legal or investment advice.

List of findings

F01 - Unnecessary signature in propose/confirm TX

--- Severity: P3

--- Round: Initial check

Any participant (lets denote the participant P) in the multi-signature schema can propose a TX for signature, or confirm a proposed TX. This is done by making a call directly to the GA authorization contract. In the initial design the caller was supposed to attach a signature of the proposed TX (hash). This is unnecessary, since by making the contract call P has proved (by signing the contract call TX) that P can sign anything. Therefore, requiring another signature adds no security - and it is just an extra unnecessary complication for the user. Also, checking a signature is an expensive operation that cost 20k gas.

--- Recommendation

Remove the unnecessary signature check and replace it with just the TX hash.

The propose/confirm should still be for a specific transaction, but the transaction hash is proof enough.

--- Status: **Fixed before the main audit.**

F02 - Add a simple nonce to distinguish transactions

--- Severity: P4

--- Round: Initial check

Technically, a transaction performed by a Generalized Account is authenticated by wrapping it in a GAMeta-transaction. The GAMeta-transaction is in principle just a description of a call to the GA authorization function. The transaction is valid if the call terminates and returns `true`. In the initial design the authorize-function takes no argument, which opens a possibility for an unfortunate situation where the GA wants to submit an identical transaction to one already existing on the chain. While in principle not a problem, there are many tools (including the TX-pool) that assume transactions to be unique - thus, this is a problem waiting to happen. It is easily solved by adding a

simple nonce as the argument of the authorization function, at the same time this will serve as a counter of how many transactions have been multi-signed.

--- Recommendation

Add a simple nonce (as an argument to `authorize` and in the state to ensure it

is the correct one).

--- Status: **Fixed before the main audit.**

F03 - Shortcomings of GAs - protect fee and gas price from manipulation

--- Severity: P1

--- Round: Initial check

A generic problem with GAMeta-transactions is that there is no protocol level mechanism that protects the fee and gas_price transaction parameters. This is a known problem, and one that is expected to be corrected in later protocol versions, but until then it unfortunately has to be guarded at the GA contract level. In a GAMeta-transaction the signature is replaced by a contract call, however, a normal transaction signature provides two things - (1) authorization that the account is allowed to do the transaction, and (2) that the transaction has not been changed. In the GAMeta-transaction the (2) is not taken care of at protocol level (though it should!) and we need to do this in the contract.

If this is not done properly a (malicious) miner could change the fee, and/or the gas price and charge the GA an arbitrary amount of tokens.

--- Recommendation

Add state variables (and setters) to allow for a limit on fee and gas price to be defined (and changed).

--- Status: **Fixed in the final version**.

F04 - Call.caller should not go into the list of signers

--- Severity: P3

--- Round: Main audit

Adding `Call.caller` to the list of signers is wrong. This is the GA-account, and it will not be able to participate in a meaningful way. If part of a N-of-N multi-signature schema this would mean a complete deadlock of the GA funds.

--- Recommendation

Don't add `Call.caller` implicitly, and even better make sure it is **not** part of the list of signers.

--- Status: **Fixed in the final version**.

F05 - Ensure the list of signers is free of duplicates

--- Severity: P3

--- Round: Main audit

The list of signers is passed as a list (which makes sense since Sets are somewhat hard to work with at the API level) - the problem is the contract check that `confirmations_required =< List.length(signers)`. If (by accident) the same account is present twice (or more) in `signers` tokens can be locked into the GA account.

--- Recommendation

Convert to a `set` before checking that we have enough signers to meet `confirmations_required`. The contract state keeps the signers as a set anyway so it is not an extra cost.

--- Status: **Fixed in the final version.**

F06 - Double negations

--- Severity: P4

--- Round: Main audit

The code contains the expression `!not_expired(...)` which is bad practice.

--- Recommendation

Change the function to `expired(...)` and use `!` when its negation is needed.

--- Status: **Fixed in the final version.**

F07 - Multiple `TxConsensusReached` events

--- Severity: P4

--- Round: Main audit

When doing N-of-M (when N < M) signing, the `TxConsensusReached` event can be emitted multiple times for a proposed Tx. This is possibly confusing.

--- Recommendation

Only emit the event when `confirmations_required` is reached, not for additional confirmations.

--- Status: **Fixed in the final version.**

F08 - Variable naming

--- Severity: P4

--- Round: Main audit

The variable `ga_tx_hash` (used in several places) is a misnomer, it contains the hash of the TX that we want to sign, and has nothing to do with GA.

--- Recommendation

Rename variable to `tx_hash`.

--- Status: **Fixed in the final version**.

F09 - Return the actual TTL in `get_consensus_info`

--- Severity: P4

--- Round: Main audit

The entrypoint just returns whether the proposed Tx is expired or not - it is probably useful to know when the Tx expire.

--- Recommendation

Add the actual TTL

--- Status: **Fixed in the final version**.

Recommendation

Since all earlier recommendations are followed and there are no new issues found in the final review, **we have no more recommendations**.

References

[1] Arts, T., Hughes, J.: How well are your requirements tested? In: Briand, L., Khurshid, S. (eds.) International Conference on Software Testing, Verification and Validation (ICST). IEEE (April 2016)

[2] Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing Telecoms Software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang. pp. 2–10. ERLANG '06, ACM, New York, NY, USA (2006)

[3] Ulf Norell, Hans Svensson, et. al., Sophia Compiler, https://github.com/aeternity/aesophia