SOLIDIFIED

Audit Report for SingularityNET. February 27, 2019.

Summary

Audit Report prepared by Solidified for SingularityNET covering their smart contracts which implement the SingularityNET platform.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the below contracts. The debrief took place on February 28, 2019 and the final results are presented here.

Audited Files

The following files were covered during the audit:

- ServiceRequest.sol

Notes:
The audit was performed on commit `67a7323f66848bffb0495910a966e69a02c42086`
The audit was based on the solidity compiler `0.4.24+commit.e67f0147`

Intended Behavior

The specification of the contracts is found in the corresponding README file

Issues Found

Critical

No critical issues were found.

Major

1. Foundation can grief users

After a user has created a request, the only way for them to retrieve their funds is via `requestClaimBack()`. The only way for `requestClaimBack()` to be called is if the `Request.status` is `Approved`, `Closed` or `Rejected`. A request can transition to these states only through the `closeRequest()`, `rejectRequest()` or `approveRequest()` functions, which are all only callable by foundation members. As a result, if a foundation member never calls any of these functions, a user's funds can be frozen in the contract.

Recommendation
Allow a user to call `closeRequest()` on requests they have opened which are still in the Opened state. This will allow them to subsequently call `requestClaimback()`.

Minor

2. The function `createRequest()` does not provide any checks to enforce that the request creator passes a valid `value`, `expiration` and a `documentURI` with their request.

There is no check that the stake the user is providing to create a request is more than the minimum stake. The same applies for the `createOrUpdateSolutionProposal()` function.

Recommendation
Add appropriate validation code to enforce that:
* `value > 0 && value >= minStake`
* `expiration > block.number`
* A valid IPFS `documentURI` exists

3. Owner cannot add funds to request before it's approved

On first look, the code indicates that a user is expected to be able to put more funds in their request before it is approved. For that to happen, the check of `block.number < req.endEvaluation` must pass. This is not the case, as `approveRequest()` is the only place where `req.endEvaluation` is set. As a result, `block.number` will always be greater than `req.endEvaluation` and the call will revert.

Recommendation
The feature that an owner is able to top up their request before approval is not documented in the specification. Consider removing the `(req.status == RequestStatus.Open && req.requester == msg.sender)` part of the require statement in line 206.

4. Consider counting previous stake towards `minStake` when adding funds to a request

If a user has already staked in a request and desires to increase it, she currently needs to increment at least `minStake`, even though that mark was already achieved.

Recommendation
Line 201:
`require(balances[msg.sender] >= amount && amount > 0 && balances[msg.sender].(amount) >= minStake)`

Notes

5. Tests do not pass

Upon cloning, calling `npm install` and then `./node_modules/.bin/truffle test`, the tests do not pass. This is a critical problem for the reliability of the codebase and verifying that the code has intended behavior. What is being tested is not clear to a reader of the tests, neither are any metrics about the code coverage provided.

Recommendation
Convert the tests to unit tests and make them independent of each other. Make use of beforeEach blocks to deploy fresh versions of the contract on each test so that you can test the behavior of each function in specific scenarios. An exceptional example of high quality testing is MolochDAO.

In your case, the issue is in the `migrations/3_ServiceRequest.js` script and the amounts used in the tests. The minimum stake variable minStake is set to a larger number than any of the Amt1...Amt7 values, resulting in all calls to `addFundsToRequest()` to fail.

6. Rename variables for clarity

A variable with name `status` typically implies an `enum` of several values, and having its value as a `bool` can be potentially confusing.

Recommendation
Consider renaming `Member.status` to `Member.active`. Also consider renaming `Member.role` to `Member.admin` and converting it to a boolean variable.

7. Remove redundant operations

Variables do not need to be initialized to 0. Based on the fact that `Request.endEvaluation` and `Request.endSubmission` are always required to be less than `Request.expiration` in `approveRequest()`, the checks double checks in require statements such as in Line 209 and 306 are redundant.

Recommendation
Refactor the code and remove such operations. Remove `(amount > 0)` since it's already covered in `(amount >= minStake)`, unless `minStake` is set 0. Remove `require(req.totalFund > 0)` in `requestClaimBack()` since it's redundant in order to save on gas costs. Using smaller `uints` can save storage gas inside structs. Ex: line 56 use uint8. Remove unnecessary struct operations: Line 148 & 149, 128 & 132, 308 & 315

8. Upgrade tooling

Current truffle version is v5.0.5. It provides better debugging, improved migrations with better async/await support, as well as shows the error strings from reverts. Avoid using outdated tooling.

Recommendation
Modify the repository to use the latest version truffle.

9. Consider using latest version of Solidity and lock contract compiler versions

The contracts use solidity version 0.4.24. It is suggested to use the latest version (`0.5.4`) and fix all compiler errors or warnings that arise. Also consider locking the version of the compiler in the pragma statement on the top of each file.

10. Add error strings to require statements

Since version `0.4.22` of solidity, `require` statements can include an error string. Consider adding appropriate error messages to all require statements.

11. Consider using `external`

Consider using `external` for function visibility if the method will only be accessed from outside. This can help save some gas especially in the case of functions where multiple arrays are passed as arguments.

12. Consider following the Solidity style guide

Formatting of the code should be adjusted for maximum readability by making sure you follow the solidity style guide rules. Consider using a linter such as ethlint. Also consider prefixing

internal functions with an underscore, "_". Also make sure to correct any typos in variable naming, such as "`fundation`" to `foundation`.

13. Consider using Solidity's `modifier` pattern instead of in-lined checks

It is best practice to use Solidity's `modifier` pattern instead of using inline-checks such as repetitive `require` statements or regular guard functions.

Recommendation
There are multiple snippets where you are performing access control and state assertions about a request which could be extracted to a modifier or a helper function.

14. Unnecessary wrapping of functions that return true or revert, with `require`

Public functions such as `depositAndCreateRequest()` adopt the pattern of returning true on success and reverting on failure. If that's the case, there's no need to wrap function calls in `require` since the result will never be false.

Recommendation
Remove the wrapping `require` statements when calling such functions.

Closing Summary

One major and three minor deficiencies were found and should be addressed before deployment of the smart contracts. Several informational notes around best practices and optimizations were also raised, and although they carry no security risk to the contracts we also encourage their amendment.

Beyond the issues mentioned, the contract was also checked for overflow/underflow issues, DoS, and re-entrancy vulnerabilities. None were discovered.

The automated scanning tools Securify, Mythril and Slither did not produce any true-positive results with respect to known vulnerabilities.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of SingularityNET or its products. This audit does not provide a security or correctness guarantee of the audited smart contracts. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.