DLSA Protocol Ethereum Contracts Security Audit

March 27th, 2021 (Updated March 29th, 2021)

@lucash-dev

https://hackerone.com/lucash-dev

https://github.com/lucash-dev

Disclaimer

Security audits are inherently limited -- there is no possibility of strict proofs that a given system doesn't contain further vulnerabilities. The best that can be offered is a good faith effort to find as many vulnerabilities as possible within the resource constraints of the project.

Given the above, this audit report is provided "AS IS", without any guarantee of correctness or completeness. The author takes no responsibility for any loss or damage caused by the use, misuse or failure to use the information contained in this document, as well as for any information that might be missing from it (e.g. missed vulnerabilities).

By using the information contained in this report, you agree to do so at your own risk, and not to hold the authors liable for any consequences of such use.

This audit does not mean endorsement of any product or service.

Summary

This document contains the results of the audit conducted on the DSLA Protocol Ethereum smart contracts.

This document is divided in the following sections:

- **Scope**: brief description of the scope of the audit.
- **Methodology**: a general description of the methodology used to find possible issues.
- **Exploitable Vulnerabilities**: a list of issues found for which attack vectors have been identified by the auditor, which can lead to a successful attack if the issue isn't fixed prior to launch.
- **Potential Vulnerabilities**: a list of actually found issues that might lead to vulnerabilities depending on factors external to the smart contracts being audited, depending on assumptions about contracts interacted with or other pieces of technology.

- **Improvement Suggestions**: a list of issues that, if corrected, might lead to increased security, though there is no evidence they currently present a security risk as they were found.

Scope

The scope of this audit are the contracts found at the GitHub repo `https://github.com/Stacktical/stacktical-dsla-contracts` as of commit `9948e1252eddb5ae08b72fc35b28a78fe79d6866`.

The audit is focused on the *implementation* and *logic* of the smart contracts code, and the following items are a non-exhaustive list of elements that are outside the scope of the audit:

- Vulnerabilities on other systems interacting with the smart contract (such as web apps).
- Vulnerabilities in the EVM implementation or the Ethereum platform in general.
- Weaknesses in the underlying cryptographic primitives (hash functions, digital signature functions).
- Economic soundness and financial feasibility of the protocol.
- Ownership structure of governance tokens or privileged cryptographic keys (if applicable).

Methodology

The audit was conducted manually by an experienced security researcher, by inspecting the code in two different ways:

- Systematic review of every function in the contracts, spotting missing best practices and logical flaws.
- Free exploration of interaction points and execution flows, searching for concrete attack vectors and invalid assumptions.

While the first approach mimics a more traditional code review, the second one tries to reproduce the kind of coverage you would obtain from a "bug bounty" program.

It's the author's belief that combining these approaches offer much higher likelihood of catching high-severity issues than using either individually.

Exploitable Vulnerabilities Found

1. Lack of validation of SLA address in `SLARegistry.requestSLI` lets attacker empty messenger's owner's LINK balance.

The root cause is that the method "requestSLI" doesn't validate that the "_sla" parameter points to a valid SLA contract created by the SLARegistry.

There is a number of validations in this method -- but all assume the "_sla" contract is valid.

The reason that is problematic is that the "requestSLI" method in the SLARegistry calls the "requestSLI" method in the Messenger (SLARegistry being trusted by the Messenger) -- and that method can create a Chainlink job -- using the Messanger's owner's LNK balance.

That means an attacker can create any number of fake SLA contracts, and then for each one, for each period, force the Messenger to create a bogus Chainlink job using the owner's tokens.

In normal use, while anyone could still use the Messanger's owner's balance to create Chainlink jobs -- those are limited by the DSLA tokens required to be staked for the SLA contract.

While this doesn't seem to allow the attacker to directly steal tokens, it is still possible to misuse user's funds.

Suggested Remedy: add a "require" statement to the "requestSLI" method that validates the "_sla" parameter corresponds to an SLA contract created by the SLARegistry. Example: "require(isRegisteredSLA(address(_sla)));"

Update from Development Team: this issue has been addressed by the development team and fixed in the master branch as of 2021-03-29.

Potential Vulnerabilities Found

2. Calls performed on external contracts without validation of address

There are multiple points in the code that allow for users to pass addresses of external contracts as an argument, and later execute calls to those contracts, while not performing any sort of validation on the addresses provided. This lack of validation can possibly lead to attacks, as the smart contract is made to call arbitrary code controllable by the attacker. While the only instance for which a concrete attack vector is the one listed in the previous section of this document, the author strongly suggests adding validations that prevent calling arbitrary contracts to the following methods:

- `SLARegistry.requestSLI` (parameter `_sla`) -- discussed in previous section.
- `SLARegistry.returnLockedValue` (parameter `_sla`)
- `SLARegistry.registerMessenger` (parameter `_messengerAddress`)

Suggested Remedy: all the above parameters should be validated as being the address of a valid contract.

Update from Development Team: this issue has been addressed by the development team and fixed (for the methods `SlARegistry.requestSLI` and

`SLARegistry.returnLockdedValue`) in the master branch as of 2021-03-29.

3. Reentrancy on Calls to Chainlink

While the good functioning of the DSLA Protocol requires Chainlink Oracles to function properly, it is important to isolate the trust placed on the Chainlink contracts, so as to try to limit potential losses in case those external dependencies are ever compromised.

In general, all methods that perform calls to external contracts should ideally be guarded against reentrancy, even if the third-party is assumed trusted. Though no concrete attack vector was identified, the author of this audit wasn't able to find any guards agains reentrancy in the contracts under scope.

In particular, both methods that create Chainlink jobs, `NetworkAnalysis.requestAnalytics` and `SEMMessenger.requestSLI` (called from `SLARegistry.requestSLI`, could become vulnerable to reentrancy attacks leading to stealing of LINK tokens from the messenger's owners -- if the Chainlink contract ever is compromised.

NOTE: this issue is listed as *potential* as the only concrete vector for this attack requires compromising a vetted external contract.

Suggested Remedy: in general *all* methods that call external contracts, *even token transfers* should be guarded against reentrancy, unless there is a *specific* reason it might need to call other methods in the same contract. In particular, the two methods listed above should be guarded against reentrancy to reduce potential for losses in case Chainlink is compromised.

Improvement Suggestions

This section contains suggestions for improvement of the contracts and issues that don't represent a significant security risk. We consider fixing those would benefit the legibility and maintainability of the contracts though.

4. Confusing documenting of Beta Tokens

Token implementations in the `contracts/tokens` directory seem to be intended as mock implementations for a public beta and/or testing -- as all three tokens allow any user to mint arbitrary amounts of tokens.

However the comments in each of the contracts doesn't clearly state that fact, which might lead to confusion and possibly mistakes when reusing the code for production.

Update from Development Team: this issue has been addressed by the development team and fixed in the master branch as of 2021-03-29.

5. Checks-Effects-Interactions pattern

To further secure functions that perform calls to external contracts, the Checks-Effects-

Interactions pattern should be followed. There are instances of functions that don't follow this patter. For example, the two methods mentioned in issue (3) perform updates after all calls to external contracts .

Most significantly, the line

```
periodAnalyticsRequested[_networkName][_periodType][_periodId] = true;
```

in `NetworkAnalytics.requestSLI` should be moved to right after the `require` statement that validates that this value was previously `false`.

Conclusion

One immediately exploitable vulnerability was found in the audited code, and some potential issues and possible improvements were discussed.

This document will be updated once a response from the development team is obtained.