Lido on Kusama/Polkadot Liquid Staking Delta Audit

Smart Contract Security Assessment

April 14, 2023

MixBytes()

ABSTRACT

Dedaub was commissioned to perform a security audit on Lido on Kusama/Polkadot liquid staking contracts by MixBytes.

This report focuses exclusively on the recent changes (on multiple contracts) in the protocol. The scope of the audit included the most recent updates and fixes to the contracts in the repository Lido KSM/DOT liquid staking, from commit `9ffd2e9acfeb2a967173eae8f436400b0f398638` up to commit `0a144834c4317d41e9853b166977f0b550ce3455` (PR #96).

As part of the audit, we also reviewed the fixes for the issues included in the report. The fixes were delivered at commit `1aec1b5809e0721c5f69b7733077920a2150601c` and we found that they had been implemented correctly.

The audit focussed solely on the delta between these versions, the auditors did not re-audit the whole protocol. The changes implement a forced unbond procedure and are accompanied by corresponding tests. Overall the auditors found that the changes properly implement the intended procedure and do not introduce vulnerabilities.

SETTING & CAVEATS

Our earlier audits describe the setting and caveats for Lido on Kusama/Polkadot protocol. As a general warning, we note that an audit of small changes in a large protocol is necessarily out-of-context. We made a best-effort attempt to understand the changed lines of code and assess whether these changes are reasonable and do not introduce vulnerabilities. The audit, however, was restricted to the modified lines, and their interaction with the rest of the protocol is not always easy to assess.

The audit's main target is security threats, i.e., what the community understanding would likely call “hacking“, rather than the regular use of the protocol. Functional correctness (i.e., issues in “regular use“) is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	`LidoUnbond::_processEnabled` might misbehave if `bufferedRedeems != fundRaisedBalance`	**RESOLVED**

The intended procedure of forced unbond requires setting `bufferedRedeems == fundRaisedBalance` via a call to `setBufferedRedeems`. As a consequence, `LidoUnbond::_processEnabled` expects these two amounts to be equal during forced unbond.

```
function _processEnabled(int256 _stake) internal {
    ...
    // Dedaub: This code will break if bufferedRedeems is not exactly
    //         equal to fundRaisedBalance
    if (isUnbondForced && isRedeemDisabled &&
        bufferedRedeems == fundRaisedBalance)
    {
        targetStake = 0;
    } else {
        targetStake = getTotalPooledKSM() / ledgersLength;
    }
}
```

However , the contract does not guarantee that these amounts will be exactly equal. For instance, `setBufferedRedeems` only contains an inequality check:

```
function setBufferedRedeems(
    uint256 _bufferedRedeems
) external redeemDisabled auth(ROLE_BEACON_MANAGER) {
    // Dedaub: Equality not guaranteed
    require(_bufferedRedeems <= fundRaisedBalance, "LIDO: VALUE_TOO_BIG");
    bufferedRedeems = _bufferedRedeems;
}
```

It is also hard to verify that no other function modifying these amounts can be called after calling setBufferedRedeems.

If, for any reason, the amounts are not exactly equal during forced unbond, the "else" branch in _processEnabled will be executed, causing targetState to be wrongly computed and likely leaving the contract in a problematic state. To make the contract more robust we recommend properly handling the case when the two amounts are different, possibly by reverting, instead of executing the wrong branch. For instance:

```
function _processEnabled(int256 _stake) internal {
    ...
    // Dedaub: Modified code
    if (isUnbondForced && isRedeemDisabled) {
        require(bufferedRedeems == fundRaisedBalance);
        targetStake = 0;
    } else {
        targetStake = getTotalPooledKSM() / ledgersLength;
    }
}
```

Another approach could be to actually set _bufferedRedeems = fundRaisedBalance within this function.

L2	Set bufferedRedeems = fundRaisedBalance and isUnbondForced in a single transaction	**RESOLVED**

Forced unbond is initiated by setting bufferedRedeems = fundRaisedBalance and isUnbondForced = true, via separate calls to setIsUnbondForced and setBufferedRedeems. If, however, **only one** of the two changes is performed, the

contract will likely misbehave. As a consequence, it would be safer to perform both updates in a single transaction.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing them.

ID	Description	STATUS
A1	Update and check all contracts before starting the forced unbond procedure	**INFO**
	The documented procedure for enabling forced unbond states to first update the Ledger contract, then to chill all Ledgers, and afterwards to upgrade the Lido contract. Although this order can work, we find it safer to first finish all upgrades of all contracts, check that the upgraded contracts work by simulating calls to the corresponding methods, and only then perform any state updating calls.	
A2	Incorrect function name in `ILidoUnbond`	**RESOLVED**
	`ILidoUnbond` contains a function `setIsRedeemEnabled`, while the method in `LidoUnbond` is called `setIsRedeemDisabled`.	
A3	Compiler known issues	**INFO**
	The code is compiled with Solidity 0.8.0 or higher. For deployment, we recommend no floating pragmas, i.e., a specific version, to be confident about the baseline guarantees offered by the compiler. Version 0.8.0, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.	

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.