Lido on Kusama Liquid Staking Audit

Smart Contract Security Assessment

Mar 28, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit on Lido on Kusama liquid staking contracts by MixBytes at commit hash `07da59e89e023be7d1bc56dc98bd6a3e7a6d85eb`. The code can be found here.

SETTING & CAVEATS

The Lido on Kusama liquid staking project aims to provide liquid KSM staking capabilities on **Kusama**, Polkadot's canary network - more specifically the **Moonriver** parachain. As the project contracts are hosted on Moonriver and the staking takes place in the relay chain, the protocol makes use of **xcKSM** - an interoperable token that allows users to transfer KSM to other parachains.

At a high level, the protocol allows users to stake their **KSM** (xcKSM tokens to be exact), in exchange for an equal amount of **stKSM** (staked KSM), while the protocol will stake the deposited funds in the relay chain. As stKSM is a rebase token, the balance of a users position will be passively increasing over time (assuming no slashing events), to reflect the staking rewards.

While traditional staking requires users to lock their funds, stKSM is a regular ERC20 that can be transferred freely. As stKSM is backed 1-to-1 by KSM, users can exchange their staked KSM tokens for an equal amount of xcKSM. This process will take a few days to complete, due to the Kusama unstaking period (7-8 days). Faster redemption will be made possible through the use of exchanges, once the token gets enough traction/usage in the future so that liquidity pools are sufficiently funded.

At the core of the protocol lies the Lido contract. This is the place where users stake/unstake their xcKSM. The pooled funds get distributed to a set of **Ledgers**, each of which manages a portion of the total funds. The role of a Ledger is to control the state of the **stash account** - an account unique to each Ledger, that lives on the Kusama relay chain and holds all the funds that have been bonded for staking. More specifically, Ledgers nominate a set of validators on the relay chain, who are at heart of Kusamas consensus algorithm and help secure the relay chain. It is important that this selection consists of high quality validators to minimize the risk of slashing.

Due to the heterogeneous nature of this setup, where the logic is in the Moonriver parachain but the underlying funds we wish to control lie on the relay chain, a set of **Oracles** are responsible for providing updates regarding the state of the stash accounts of each Ledger, mostly staking balance, as this is what is of interest to the application. Each era, Oracle members push staking report variants and the variant that manages to reach the required quorum is chosen as the "winner" of the current voting round, and the report data is pushed to the Ledger.

As there are heavy requirements for cross-chain communication (both upward and downward), a **Controller** contract has been created as an abstraction layer between the high-level business logic in the Moonriver parachain and the low-level operations required in order to bond/unbond KSM on the relay chain.

It should be noted that there are non-trivial elements of centralization in the protocol. While the authorization mechanism that has been put in place is great for the overall security of the project, when the privileged address is an EOA like in the current deployment of the protocol (https://moonriver.moonscan.io/address/0x749C992900dcc143D85E140c3aBE271FC021D498) there are some risks involved, namely leaking/exposing a private key. More specifically, the aforementioned admin address holds almost all roles, and the list of privileged actions it can perform includes, but is not limited to:

- Upgrading the beacon proxy implementation.
- Adding/removing and pausing/unpausing ledgers.
- Pausing/unpausing the protocol.
- ...

The developers have confirmed that these are only for the beta testing phase and the relevant roles will be transferred to a DAO multisig.

It is highly recommended that these actions be protected by a multi-signature vault, in order to reduce the risk of exposure. Ideally, this functionality will be delegated entirely to a DAO which will govern the entirety of the project, once the community has grown to a point where it can manage the protocol.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect 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: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Compromised oracle quorum in a single era can cause unbounded loss of funds	**RESOLVED**[1]

Although the protocol is inherently based on partially-trusted Oracles, it should tolerate some limited forms of Oracle manipulation. Complete long-term control of the Oracle quorum is infeasible for an adversary; still, controlling the quorum for a *single era* should not be ruled out. The protocol should ensure that the effect of such an attack is limited, and does not result in a complete loss of funds.

However, we have identified a situation in which a *single* malicious Oracle report (in a single era) can result in a complete loss of funds. The idea is simple: the malicious Oracle will register a huge reward for a specific Ledger, causing `fundRaisedBalance` to increase by an *arbitrary* amount. This will inflate the value of `getPooledKSMByShares`, and the adversary will be able to redeem his shares for an unbounded amount of KSM.

Reporting an arbitrarily large reward is normally not accepted by the Ledger. However, when a Ledger is new, and the malicious Oracle pushes the first report of the newly created Ledger, the corresponding checks fail and an unbounded reward can indeed be registered.

We present a step-by-step walkthrough of the issue:

As the Ledger is newly created, it has never been pushData'ed to before and will have the following state (not exhaustive - just the fields that are of interest right now):

- `Ledger.transferDownwardBalance = 0`
- `Ledger.transferUpwardBalance = 0`
- `Ledger.cachedTotalBalance = 0`

[1] Relative difference is now checked against `lido.totalSupply()`, instead of `cachedTotalBalance`. The parameter `MAX_ALLOWABLE_DIFFERENCE` has also been tuned to reflect this change.

At this point the single malicious report is pushed to the Ledger by `Ledger.pushData(report)`. During the execution of the function, the first "roadblock" is the following check:

```
function pushData(uint64 _eraId, Types.OracleData memory _report) {
    if (!_processRelayTransfers(_report)) {
        return;
    }
    // [...]
}
```

Examining the `_processRelayTransfers` function more closely, we see that it follows a skeleton similar to the following:

```
function _processRelayTransfers(Types.OracleData memory _report) {
    uint128 _transferDownwardBalance = transferDownwardBalance;
    // Dedaub: Check 1
    if (_transferDownwardBalance > 0) {
        // [...]
    }
    // [...]
    uint128 _transferUpwardBalance = transferUpwardBalance;
    // Dedaub: Check 2
    if (_transferUpwardBalance > 0) {
    // [...]
    }
    // Dedaub: Check 3
    if (_transferDownwardBalance == 0 && _transferUpwardBalance == 0) {
        // update ledger data from oracle report
        totalBalance = _report.stashBalance;
        lockedBalance = _report.totalBalance;
        return true;
    }
    return false;
}
```

Given the state of the Ledger (see bullet points before), it can be easily be seen that checks 1 and 2 will fail, thus the body of their respective if blocks will not be executed, and the control flow will then go to check 3, which will succeed, causing the function to return true.

Thus we successfully passed the first roadblock in pushData. Next up is the the check for non-zero cachedTotalBalance:

```
    if (cachedTotalBalance > 0) {
        uint128 relativeDifference =
         _report.stashBalance > cachedTotalBalance ?
         _report.stashBalance - cachedTotalBalance :
         cachedTotalBalance - _report.stashBalance;
        // NOTE: 1 / 10000 - one base point
        relativeDifference =
          relativeDifference * 10000 / cachedTotalBalance;
        require(relativeDifference < LIDO.MAX_ALLOWABLE_DIFFERENCE(),
                "LEDGER: DIFFERENCE_EXCEEDS_BALANCE");
    }
```

As cachedTotalBalance is zero in our case, this block will not be executed. Note that `relativeDifference < LIDO.MAX_ALLOWABLE_DIFFERENCE()` is an important check to prevent arbitrarily large rewards! The fact that we can skip this check is crucial to the severity of this issue.

At this point we should introduce what the malicious report will look like:

```
Report {
    stakeStatus: Idle,
    stashBalance: X, // X is arbitrarily large
    totalBalance: X,
    activeBalance: 0,
    unlocking: [{era: type(uint).max, balance: X}]
}
```

First, note is that this passes the `_report.isConsistent()` check done in `OracleMaster`.

Continuing with the attack exploitation walkthrough, it is easy to see that the following block will execute:

```
    if (_cachedTotalBalance < _report.stashBalance) {
      // if cached balance > real => we have reward
```

```
        uint128 reward = _report.stashBalance - _cachedTotalBalance;
        LIDO.distributeRewards(reward, _report.stashBalance);

        emit Rewards(reward, _report.stashBalance);
    }
```

As a consequence LIDO.distributeRewards will cause fundRaisedBalance to increase by an unbounded amount.

At this point, the attack is almost done, the only thing left is successfully completing the pushData call. We present a list of variables that are important to this goal:

```
1. uint unlockingBalance, uint128 withdrawableBalance =
       report.getTotaUnlocking() = (X, 0);

2. uint nonWithdrawableBalance = unlockingBalance - withdrawableBalance = X;

3. uint deficit = report.stashBalance - ledgerStake = X - 0 = X;
4. uint relayFreeBalance = report.getFreeBalance()
                         = report.stashBalance - report.totalBalance
                         = 0;
```

It can be seen that the following conditions need to be skipped in order to exit the function successfully:

1. if (_ledgerStake < MIN_NOMINATOR_BALANCE && status != Types.LedgerStatus.Idle)
    - Skipped as status = report.status = Idle
2. if (relayFreeBalance > 0)
    - Skipped as relayFreeBalance = 0
3. if (deficit > 0 && withdrawableBalance > 0)
    - Skipped as withdrawableBalance = 0
4. if (nonWithdrawableBalance < deficit)
    - Skipped as nonWithdrawableBalance = deficit
5. if (relayFreeBalance > 0)
    - Skipped as relayFreeBalance = 0

At this point, the malicious report has successfully gone through and manipulated the `fundRaisedBalance` field in Lido.

This issue is quite complex and we don't have a definite solution:

- One easy check that would make the attack much harder is to disallow reward distribution when `cachedTotalBalance == 0`. Conceptually, one cannot register rewards without staking any funds.
- It would be also preferable to introduce checks that do not depend at all on values controlled by the Oracle. For instance, the code protects against an arbitrarily large reward by checking that `(_report.stashBalance - cachedTotalBalance) / cachedTotalBalance` Is small. However, all these quantities are controlled by the Oracle report. A compromised Oracle could maybe increase `cachedTotalBalance`, and set `stashBalance` even higher. If `cachedTotalBalance` is replaced by the amount of staked funds that the Lido sent to the Ledger (without any dependence on the Oracle) the check would be stronger.

H2	Failure to achieve quorum if the stash account balance is modified during an era.	**DISMISSED**[2]

An oracle report is pushed to the Ledger only if a certain quorum of oracles produce *identical* reports for a given era. Although the exact code of the off-chain oracles is not known to the auditors, a natural implementation would just report the state of the stash accounts in the current block, after a new era has started. Note that rewards are computed once every era, so balances are not expected to change within an era, so oracles can produce identical reports without coordination.
However, the technique of directly transferring funds to a stash account, already discussed in M2, can be used to achieve a different attack vector. A malicious user could constantly transfer arbitrarily small amounts to a stash account, multiple times per era (even in every single block), to cause the balance to vary (even slightly). Given that oracles don't coordinate with each other, it is likely that they will send their reports in different blocks, and since the balance constantly changes, they will report different

[2] Resolved by an off-chain mechanism; each oracle reports on state at the last block of the previous era, not the current one.

values. Since we need a quorum of identical reports, the protocol will fail to make progress. This attack has lower chances to succeed than M2, but on the other hand requires a negligible amount of funds to be performed.

To avoid this issue, the oracles could be instructed to report the state of a specific block, e.g. the first of each era. We generally recommend carefully documenting the exact behavior that is expected from the oracles, since it is important that they produce identical results.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Oracle vote result can be wrong if a quorum softening happens while voting is in progress.	**ACKNOWLEDGED**

The Oracle logic allows for quorum softening, that is, lowering the amount of required votes a report variant needs to win a voting round. This is performed in Oracle::softenQuorum:

```
function softenQuorum(uint8 _quorum, uint64 _eraId) external onlyOracleMaster {
    // Dedaub: Summary: Check if the quorum softening will result in a new
    //         winner. Note that the winner needs to be unique - if there
    //         are two variants that both share the first place, it is deemed
    //         that no consensus has been achieved yet.
    (bool isQuorum, uint256 reportIndex) = _getQuorumReport(_quorum);
    if (isQuorum) {
        Types.OracleData memory report = _getStakeReport(reportIndex);
        _push(
          _eraId, report
        );
    }
}
```

When the quorum is softened, a check is performed to see if consensus has been achieved, that is, there is a unique variant with the most votes. In the case where two variants share the first place, no consensus is achieved, and the voting process will go on. However, the Oracle::reportRelay function declares as winner *the first variant that reaches quorum* (or surpasses it in some cases, due to a softening). In case of softening, this might not be the variant with the most votes.

With this short context in place, consider the following scenario:

Assume we have the following state in Oracle, represented as a list of (report variant, #votes) pairs:

[(v0, 1), (v1, 3), (v2, 5), (v5, 6), (v6, 6)], required quorum = 7

Assume that a quorum softening takes place, which lowers the required quorum from 7 to 4. The state will now be:

[(v0, 1), (v1, 3), (v2, 5), (v5, 6), (v6, 6)], required quorum = 4

Note that no consensus has been reached yet, as both v5 and v6 share the "most voted slot" with 6 votes each.

Now assume that a new vote arrives, which is on variant v1. As this vote will cause the vote count for variant v1 to be greater than or equal to the quorum (=4), it will be declared the "winner" of this voting round, and the data of this report will be pushed to the ledger. This result however is *wrong*, since v5 and v6 in the above example have a greater number of votes (note also that that the Oracle members that have already cast their votes are "frozen" - they can't vote again, until the voting state is cleared by calling clearReporting on the Oracle).

A possible solution is in `Oracle::reportRelay` to check, not only that the current variant has the quorum, but also that no other variant has greater or equal votes (this can only happen after softening). This will ensure that the winner is always the variant with the most votes.

Another possible mitigation could be to reset the reporting each time a quorum softening takes place, similar to how it's done when an oracle member is being removed.

M2	DoS can be performed by increasing the ledger's stash balance	**ACKNOWLEDGED**

During an oracle update, `Ledger::pushData` checks that the Ledger's stash balance does not increase by more than `MAX_ALLOWABLE_DIFFERENCE` (currently set to 30%). This is an important check, as already discussed in H1, to ensure that compromised oracles cannot result in a total loss of funds. However, this check makes the assumption that all changes in the Ledger's balance (apart from transfers initiated by the protocol) are the result of staking rewards or slashing.

This is true under honest operation, however a malicious user could *directly transfer* funds to the Ledger's stash account, in order to increase its balance. If the adversary transfers to a stash account 30% of its balance, then *honest* Oracles will report this increase, causing `Ledger::pushData` to fail. This would completely stall all operations of the ledger, since `Ledger::pushData` also controls all transfers to/from the Ledger's stash account.

Of course, such an attack is costly, since the adversary needs to spend 30% of the Ledger's stash balance. However, when a new Ledger is launched, its balance is small until new funds are transferred to it, so such a DoS might be worth its cost.

We recommend modifying the oracle reporting procedure in order to detect direct transfers from unknown accounts, and allow updates to be performed even if `MAX_ALLOWABLE_DIFFERENCE` is exceeded.

LOW SEVERITY:

ID	Description	STATUS
L1	Setter functions can be frontrun	**ACKNOWLEDGED**

Throughout the codebase there are some one-time setter functions that are frontrunnable. More specifically:

- `OracleMaster::setLido`
- `Controller::setLido`
- `Withdrawal::setStKSM`

are all one-time runnable setters that can be front run by a malicious third party.

While this issue is easily detectable (execution reverts if already set) and can be addressed by just re-deploying the contract, it is highly recommended that these functions be guarded to allow only privileged users to call them.

L2	Missing argument validation	ACKNOWLEDGED

In Lido.sol, some arguments of the initialize function are not being validated, while in the corresponding setter functions sanity checks are performed. More specifically, non-zero checks are missing for the arguments _developers, _treasury and _maxAllowableDifference:

```
contract Lido is stKSM, Initializable {
    function initialize(
       // [...]
       address _developers,
       address _treasury,
       uint128 _maxAllowableDifference
       // [...]
     ) {
        // Dedaub: These checks are missing
        require(_developers != address(0));
        require(_treasury != address(0));
        require(_maxAllowableDifference != 0);
        // [...]
    }
    function setDevelopers(address _developers) external auth(ROLE_DEVELOPERS) {
       // Dedaub: _developers argument is validated here
       require(_developers != address(0), "LIDO: INCORRECT_DEVELOPERS_ADDRESS");
       developers = _developers;
    }

    function setTreasury(address _treasury) external auth(ROLE_TREASURY) {
        // Dedaub: _treasury argument is validated here
        require(_treasury != address(0), "LIDO: INCORRECT_TREASURY_ADDRESS");
        treasury = _treasury;
    }

    function setMaxAllowableDifference(uint128 _maxAllowableDifference) external
       auth(ROLE_BEACON_MANAGER)
    {
        // Dedaub: _maxAllowableDifference argument is validated here
        require(_maxAllowableDifference               >            0,            "LIDO:
INCORRECT_MAX_ALLOWABLE_DIFFERENCE");
        MAX_ALLOWABLE_DIFFERENCE = _maxAllowableDifference;
    }

}
```

It is recommended that these checks be added to reflect the validation logic that is present in the setters.

L3	Possible function signature clash in LedgerProxy	**ACKNOWLEDGED**

In LedgerProxy.sol, the function beaconAddress is declared external. While this on it's own is no reason for concern, when put in the context of a Proxy contract (which it is indeed part of), things get a bit more complicated.

As a proxy contract, it's core functionality is to forward all calls to the underlying implementation. However, if the proxy contract has some externally accessible functions of its own, then these get precedence over the ones in the proxy implementation. This, in conjunction with the fact that public function calls are dispatched based on their 4-byte hash signature, means that there is a really small but real chance of a function signature collision happening between beaconAddress and any of the external/public functions of the underlying implementation, and which can cause undefined and undesirable behavior.

While this event is quite rare, it is recommended that this external function be removed.

L4	Incorrect length bound check in OracleMaster	**RESOLVED**

In OracleMaster::addOracleMaster, the following check on the length of the member array is performed before pushing a new member:

```
function addOracleMember(address _member) external
auth(ROLE_ORACLE_MEMBERS_MANAGER) {
        require(_member != address(0), "OM: BAD_ARGUMENT");
        require(_getMemberId(_member) == MEMBER_NOT_FOUND, "OM: MEMBER_EXISTS");

        // Dedaub: Should be simply < MAX_MEMBERS
        require(members.length < MAX_MEMBERS - 1, "OM: MEMBERS_TOO_MANY");

        members.push(_member);
        emit MemberAdded(_member);
}
```

However, this is incorrect as it allows up to MAX_MEMBERS-1 members, instead of the intended MAX_MEMBERS.

This can be easily fixed by changing the right-hand side of the comparison to simply MAX_MEMBERS, as the comment in the snippet suggests.

L5	Oracle::softenQuorum should exit early if result has already been pushed	**ACKNOWLEDGED**

In Oracle::softenQuorum, if quorum has already been reached before then softening, then this will be the case after the fact. Thus, the implementation should not push the report again, similar to how reportRelay returns early when result has already been pushed:

```
function softenQuorum(uint8 _quorum, uint64 _eraId) external onlyOracleMaster {
    (bool isQuorum, uint256 reportIndex) = _getQuorumReport(_quorum);

    // Dedaub: Second condition is missing
    if (isQuorum && !isPushed) {
       Types.OracleData memory report = _getStakeReport(reportIndex);
        _push(
            _eraId, report
        );
    }
}
```

It is recommended that this extra condition be added to be consistent with reportRelay which also pushes data to the ledger.

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	Functions can be declared view or pure	**OPEN**

There are some functions that can have their state mutability restricted further than it currently is. More specifically:

Can be declared as view:
- `Controller::getSenderIndex`
- `Controller::getSenderAccount`

Can be declared as pure:
- `wstKSM::decimals`
- `Controller::toLeBytes`

ID	Description	STATUS
A2	Gas optimization: Runtime verification can be performed at compile time	**OPEN**

In `LedgerProxy.sol`, the following check is done in the constructor:

```
contract LedgerProxy is Proxy {
    bytes32 internal constant _BEACON_SLOT =
0xa3f0ad74e5423aebfd80d3ef4346578335a9a72aeaee59ff6cb3582b35133d50;

    constructor(address beacon, bytes memory data) payable {
        // Dedaub: Can be statically computed
       assert(_BEACON_SLOT == bytes32(uint256(keccak256("eip1967.proxy.beacon")) -
1));
       _upgradeBeaconToAndCall(beacon, data, false);
    }
}
```

However this check is unnecessary, as the computation can be moved at the storage field declaration and it will be computed once, at compile time:

```
contract LedgerProxy is Proxy {
      // Dedaub: Computed once, at compile time.
      bytes32 internal constant _BEACON_SLOT =
bytes32(uint256(keccak256("eip1967.proxy.beacon")) - 1));

      constructor(address beacon, bytes memory data) payable {
          _upgradeBeaconToAndCall(beacon, data, false);
      }
}
```

A3	Gas optimization: Approval can be performed once	OPEN

In `wstKSM::submit`, there is the following check that is done on each call:

```
function submit(uint256 _vKSMAmount) external returns (uint256) {
        require(_vKSMAmount > 0, "WSTKSM: ZERO_VKSM");
        VKSM.transferFrom(msg.sender, address(this), _vKSMAmount);

        // Dedaub: This check is done on each call
        if (VKSM.allowance(address(this), address(LIDO)) < _vKSMAmount) {
             VKSM.approve(address(LIDO), type(uint256).max);
        }
        // [...]
}
```

The first time this check will be performed, the body of the if statement will also execute; granting “infinite” approval to the `Lido` contract. After this point, the check becomes a no-op – “infinite” approval is too large of a value for normal transactions to have a noticeable effect on it.

It is recommended that the approval be granted once in the constructor, and the check be removed to help save some gas from future invocations.

A4	Gas optimization: Duplicate operation	OPEN

In `WithdrawalQueue::init`, there is an operation is performed twice:

```
library WithdrawalQueue {
    function init(Queue storage queue, uint256 cap) internal {
        for (uint256 i = 0; i < cap; ++i) {
            queue.items.push(Batch(0, 0));
        }
        queue.ids = new uint256[](cap);
        queue.first = 0;
        queue.size = 0;

        // Duplicate: Duplicate operation
        queue.size = 0;
        queue.cap = cap;
    }
}
```

A5	Compiler known issues	INFO

The contracts are compiled with the Solidity compiler v0.8.12 which, at the time of writing, has no known bugs.

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 contracts. 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, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.