Lido

Smart Contract Security Assessment

August 3, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of the Lido protocol on the Avalanche blockchain.

This is a liquid staking protocol, similar to the Lido protocol on other chains, which allows users to stake AVAX while enjoying the liquidity of their staked tokens. This audit report covers the contracts listed below from the repository AvaLido/contracts at commit hash 8de3d5f334c15c8ef949560af02bc8c18769e132. Two senior auditors along with one junior auditor worked over the codebase for over 8 days.

The audited contract list is the following:

```
src/
├─ AvaLido.sol
├─ MpcManager.sol
├─ Oracle.sol
├─ OracleManager.sol
├─ Roles.sol
├─ stAVAX.sol
├─ Treasury.sol
├─ Types.sol
├─ ValidatorSelector.sol
│
├─ deploy/
│  └─ Deploy.t.sol
│
└─ interfaces/
   ├─ IMpcManager.sol
   ├─ IOracle.sol
   ├─ ITreasury.sol
   └─ IValidatorSelector.sol
```

The codebase appears to be well-documented and of high-quality. It also contains an extensive test suite (which was not audited). Users of the protocol should also note that the protocol's functionality crucially relies on its interaction with off-chain code (oracles, MPC group members, etc) that was not covered in this audit.

One critical and one high severity issue were identified, which should be urgently addressed. These issues could be easily resolved by the protocol team.

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than 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, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. The scope of the audit includes smart contract code. Interactions with off-chain (front-end or back-end) code are not examined other than to consider entry points for the contracts, i.e., calls into a smart contract that may disrupt the contract's functioning.

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: -User or system funds can be lost when third-party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to 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:

ID	Description	STATUS
C1	Potential theft of funds due to the ability to manipulate the AVAX:stAVAX exchange rate	**RESOLVED**

The fact that the PayableAvaLido contract depends on `address(this).balance` to calculate the AVAX:stAVAX exchange rate makes the exchange rate susceptible to manipulation from an attacker that might be able to increase the AVAX balance of the AvaLido contract without minting new stAVAX shares/tokens.

More precisely, the protocol computes the AVAX:stAVAX exchange rate using the expression `avaxToStAVAX(protocolControlledAVAX(), amount)`, which depends on the functions below:

```
function avaxToStAVAX(uint256 totalControlled, uint256 avaxAmount)
        public view returns (uint256) {
    // The result is always 1:1 on the first deposit.
    if (totalSupply() == 0 || totalControlled == 0) {
        return avaxAmount;
    }
    return Math.mulDiv(avaxAmount, totalSupply(), totalControlled);
}

function protocolControlledAVAX() public view override returns (uint256) {
    return amountStakedAVAX + address(this).balance;
}
```

A manipulation attack of this nature would be most effective if the attacker managed to make the exchange rate 0 for a user's deposit, as this would mean that the user that deposited after the attacker would get 0 stAVAX for their deposit, essentially giving up their deposit to the people having deposited earlier or ideally just to the attacker. In our case, the attacker could succeed in that if they managed to make the expression `avaxAmount * totalSupply() / totalControlled`, where avaxAmount is the amount deposited by the unsuspected user, evaluate to 0. To do so, they would have to increase the `totalControlled` amount, without minting new stAVAX shares, i.e., without increasing `totalSupply()`, such that `avaxAmount * totalSupply()` is less than totalControlled. Assuming that `totalSupply()` is equal to `totalControlled` and the attacker observes (using the mempool) that a user is going to deposit `avaxAmount` of AVAX, the attacker would need to directly transfer to the contract any amount greater than `(avaxAmount-1) * totalControlled` to make the exchange rate for the user's deposit 0.

Of course, the feasibility and the success of such an attack depend on certain preconditions.

First of all, the `totalControlled` value should be sufficiently small, as otherwise to be able to steal any significant `avaxAmount` the attacker would have to be in possession of huge amounts of AVAX (`(avaxAmount-1) * totalControlled`). Considering the potentially huge launch of the Lido protocol on Avalanche and the funds that might get attracted initially we believe that an attacker could attempt to be the first depositor (using front-running techniques) so as to set the `totalControlled` amount to a favorable value (e.g., 1 wei), which would allow them to steal later deposits of great value without needing exorbitant amounts of AVAX. One obstacle in the attacker's way appears to be the fact that the AvaLido contract defines a `minStakeAmount` of 0.1 ether (AVAX), which seems to be able to prevent such an attack. However, we believe that the attacker could circumvent that in the following way:

1. Deposit 0.1 AVAX.
2. Make a withdrawal request of 0.1 ether - 1 wei.
3. Forcefully send via contract self-destruct 0.1 ether - 1 wei to PrincipalTreasury.
4. Call `claimUnstakedPrincipals` to fill the withdrawal request.
5. Claim the filled request.

At the end of this process, which can be done atomically by including all interactions in a smart contract function, i.e., in one transaction, the attacker will have managed to set `totalControlled` to 1 wei.

The second precondition is that the attacker should be able to transfer `(avaxAmount-1) * totalControlled` AVAX to the contract without going through the deposit function and the minting of stAVAX. This can be easily achieved as the PayableAvaLido contract implements an unrestricted receive() function. It is important to note that even if the receive() function was guarded an attacker could forcefully send AVAX directly to the contract by self-destructing a contract with the needed amount of AVAX.

HIGH SEVERITY:

ID	Description	STATUS
H1	DoS due to unbounded loop	**RESOLVED**

`AvaLido::fillUnstakeRequests` loops over all pending `unstakeRequests`, trying to fill them with the given `amount`.

```
for (uint256 i = unfilledHead; i < unstakeRequests.length; i++) {
      if (remaining == 0) break;
      // ...
}
```

Note that the length of this loop is unbounded, and potentially controlled by an adversary. The `remaining == 0` conditional check does break the loop, but the length remains unbounded since a large number of small unstake requests could be filled by the given amount.

As a consequence, an adversary could perform a DoS by creating a large number of very small unstake requests, using multiple accounts since each account is limited to `maxUnstakeRequests` (10) requests. Each deposit is limited to at least `minStakeAmount (0.1 AVAX)`, but unstake requests have no such limit, thus a request of 1 wei is possible (the adversary could even deposit 0.1 AVAX but withraw everything but 10 wei before the attack, so in the end the attack is cheap to perform). The large number of pending unstake requests could cause the loop to run out of gas when trying to fill them, causing a DoS to many vital functions that call `fillUnstakeRequests` (`deposit, claimUnstakedPrincipals, claimRewards`).

To prevent such issues, we recommend all loops to be strictly bounded. This could be achieved by filling just a bounded number of unstake requests (the remaining ones can be filled by future transactions). Having a lower limit in the amount of an unstake request could be also beneficial.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Oracle::setNodeIDList may irreversibly delete essential data	**DISMISSED**
	`Oracle::setNodeIDList` deletes `reportsByEpochId[latestEpochId]`, i.e., the latest epoch data, as they might no longer be valid due to validators being removed from the list. The latest epoch data is supplied to the Oracle contract via the `OracleManager`, which calls the function `Oracle::receiveFinalizedReport` and marks that the report for that epoch has been finalized, meaning that it cannot be resubmitted. This information, which might irreversibly get deleted by the `Oracle::setNodeIDList`, is essential for the `ValidatorSelector` contract to proceed with the validator selection process. Thus, care should be taken to ensure that `Oracle::setNodeIDList` isn't called after `OracleManager::receiveMemberReport` and before `ValidatorSelector::getAvailableValidatorsWithCapacity`, as such a sequence of calls would leave the system in an invalid state.	
M2	Transaction may revert due to array out-of-bounds error in ValidatorSelector::getAvailableValidatorsWithCapacity	**RESOLVED**
	Function `ValidatorSelector::getAvailableValidatorsWithCapacity` retrieves the latest epoch validators from the Oracle in the `validators` array, computes how	

many of those satisfy the filtering criteria and then creates an array of that size, `result`, and traverses again the `validators` array to populate it.

```
function getAvailableValidatorsWithCapacity(uint256 amount)
        public view returns (Validator[] memory) {
    Validator[] memory validators = oracle.getLatestValidators();

    uint256 count = 0;
    for (uint256 index = 0; index < validators.length; index++) {
        // ... (filtering checks on validators[index])
        count++;
    }

    Validator[] memory result = new Validator[](count);
    for (uint256 index = 0; index < validators.length; index++) {
        // ... (filtering checks on validators[index])
        // Dedaub: index can get bigger than result.length.
        // Dedaub: a count variable needs to be used as in the above loop.
        result[index] = validators[index];
    }
    return result;
}
```

However, there is a bug in the implementation that can cause an array out-of-bounds exception at line `result[index] = validators[index]`. Variable `index` is in the range `[0, validators.length-1]`, while `result.length` will be strictly less than `validators.length-1` if at least one validator has been filtered out of the initial `validators` array, thus `index` might be greater than `result.length-1`. Consider the scenario where `validators = [1, 2]` and `count` (or `result.length`) is 1 as the validator with id 1 has been filtered out. Then the second loop will traverse the whole `validators` array and will try to assign the validator with id 2 (array index 1) to `result[1]` causing an out-of-bounds exception, as `result` has a length of 1 (can only be assigned to index 0). Using a count variable, similarly to the first loop, would be enough to solve this issue.

M3	DoS due to the ability of a group to confirm any public key	**RESOLVED**

A DoS attack could be possible due to the ability of a group to perform confirmations for any given public key. More specifically, we think that a group with adversary members can front-run the `reportGeneratedKey()` using a public key which was requested by another group, via `requestKeygen()`. By doing so, this public key will be confirmed by and assigned to the adversary group.

```
// MpcManager.sol::reportGeneratedKey:214
if (_generatedKeyConfirmedByAll(groupId, generatedPublicKey)) {
    info.groupId = groupId;
    info.confirmed = true;
    ...
}
```

This will DoS the system for the benevolent group which will not be able to perform any further confirmations for this public key.

```
// MpcManager.sol::reportGeneratedKey:208
if (info.confirmed) revert AttemptToReconfirmKey();
```

The adversary group can then proceed with joining the staking request changing the threshold needed for starting the request (of course in the case where the adversary group has a smaller threshold than the original one).

```
// MpcManager.sol::joinRequest:238
uint256 threshold = _groupThreshold[info.groupId];
```

However, they don't have to join the request and can leave it pending. Since multiple public keys can be requested for the same group, they can proceed with different keys and different stake requests if they wish to interact with the contracts benevolently for their own benefit.

The `MpcManager.sol` contract has quite a bit of off-chain logic, but we believe that it is valid as an adversary model to assume that groups can not be entirely trusted and that they can act adversely against other benevolent groups. In the opposite scenario, considering all groups as trusted could lead to centralization issues while only the MPC manager can create the groups.		
M4	`MpcManager::reportUTXO()` can be called by a member of any group with any generated public key	**RESOLVED**
`MpcManager::reportUTXO()` does not contain any checks to ensure that the member which calls it is a member of the group that reported and confirmed the provided `genPubKey`. This means that a member of any group can call this function with any of the generated public keys even if the latter has been confirmed by and assigned to another group. By doing so, a group can run `reportUTXO()` changing the threshold needed for the report to be exported. It is not clear from the specification if allowing any member to call this function with any public key is the desired behaviour or if further checks should be applied.		
M5	A number of remaining TODO items suggest certain functionality is not implemented	**RESOLVED**
There are a number of TODO items that spread across the entire codebase and test suite. Most of these TODOs are trivial and the test suite appears to be well developed. However, there is a small number of TODOs that concern checks and invariants and also unimplemented functionality like supporting more types of validator requests. This could mean that further development is needed, which could render the current security assessment partially insufficient.		

LOW SEVERITY:

ID	Description	STATUS
L1	Partial claim of AVAX might result in rounding errors	**RESOLVED**

According to a note in the AvaLido::claim function, the protocol *allows partial claims of unstake requests so that users don't need to wait for the entire request to be filled to get some liquidity*. This is one of the reasons the exchange rate stAVAX:AVAX is set in function `requestWithdrawal` instead of in `claim`. The partial claim logic is implemented mainly in the following line:

```
uint256 amountOfStAVAXToBurn =
    Math.mulDiv(request.stAVAXLocked, amount, request.amountRequested);
```

The amount of stAVAX that are traded back, `request.stAVAXLocked`, is multiplied by the amount of AVAX claimed, `amount`, and the result is divided by the whole AVAX amount corresponding to the request, `request.amountRequested` to give us the corresponding amount of stAVAX that should be burned. This computation might suffer from rounding errors depending on the `amount` parameter, leading to a small amount of stAVAX not being burned. We believe that these amounts would be too small to really affect the exchange rate of stAVAX:AVAX, still it would make sense to verify this or get rid of the rounding error altogether.

ID	Description	STATUS
L2	Treasury::claim might fail due to uninitialized variable	**RESOLVED**

Function `Treasury::claim` could be called while the `avaLidoAddress` storage variable might not have been set via the `setAvaLidoAddress`, leading to the transaction reverting due to `msg.sender` not being equal to `address(0)`. This outcome can of course be considered desirable, but at the same time, the needed call to `setAvaLidoAddresss` adds unnecessary complexity. Currently, the `setAvaLidoAddress` function works practically as an initializer, as it cannot set the avaLidoAddress storage variable more than once. If that is the intent, avaLidoAddress could be set in the initialize function, which would reduce the chances of claim and successively of AvaLido::claimUnstakedPrincipals and AvaLido::claimRewards calls reverting.

L3	AvaLido::deposit check considers deposited amount twice	**RESOLVED**

The function AvaLido::deposit implements the following check:

```
if (protocolControlledAVAX() + amount > maxProtocolControlledAVAX)
    revert ProtocolStakedAmountTooLarge();
```

However, the check should be changed to:

```
if (protocolControlledAVAX() > maxProtocolControlledAVAX)
    revert ProtocolStakedAmountTooLarge();
```

as the function protocolControlledAVAX() uses address(this).balance, meaning that amount, which is equal to the msg.value, has already been taken into account once and if added to the value returned by protocolControlledAVAX(), it would be counted twice. Nevertheless, we expect that both conditions would never be satisfied as maxProtocolControlledAVAX is by default set to type(uint256).max. Still, we would advise addressing the issue just in case maxProtocolControlledAVAX is changed in the future.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

The protocol defines several admin/manager roles that serve to give access to specific functions of certain contracts only to the appropriate entities. The following roles are defined and used:

- `DEFAULT_ADMIN_ROLE`
- `ROLE_PAUSE_MANAGER`
- `ROLE_FEE_MANAGER`
- `ROLE_ORACLE_ADMIN`
- `ROLE_VALIDATOR_MANAGER`
- `ROLE_MPC_MANAGER`
- `ROLE_TREASURY_MANAGER`
- `ROLE_PROTOCOL_MANAGER`

For example, the entity that is assigned the `ROLE_MPC_MANAGER` is able to call functions MpcManager::createGroup and MpcManager::requestKeygen that are essential for the correct functioning of the MPC component. Multiple roles allow for the distribution of power so that if one entity gets hacked all other functions of the protocol remain unaffected. Of course, this assumes that the protocol team distributes the different roles to separate entities thoughtfully and does not completely alleviate centralization issues.

The contract `MpcManager.sol` appears to build on/depend on a lot of off-chain logic that could make it suffer from centralization issues as well. A possible attack scenario is described in issue M3 above that raises the question of credibility for the MPC groups even though they can only be created by the MPC manager.

OTHER/ ADVISORY ISSUES:

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

ID	Description	STATUS
A1	The array of public keys provided to MpcManager::createGroup needs to be sorted	**RESOLVED**

The array of public keys provided to MpcManager::createGroup by the MPC manager needs to be sorted otherwise the `groupId` produced by the keccak256 of the array might be different for the same sets of public keys. As sorting is tricky to perform on-chain and has not been implemented in this instance, the contract's API or documentation should make it clear that the array provided needs to be already sorted.

ID	Description	STATUS
A2	Check in AvaLido::fillUnstakeRequests is always true	**RESOLVED**

The following check in `AvaLido::fillUnstakeRequests` is expected to be always true, since the `isFilled` check right before guarantees that the request is not filled.

```
if (isFilled(unstakeRequests[i])) {
      // This shouldn't happen, but revert if it does for clearer testing
      revert("Invalid state - filled request in queue");
}

// Dedaub: the following is expected to be always true
if (unstakeRequests[i].amountFilled < unstakeRequests[i].amountRequested)
{
    ...
}
```

A3	Functions responsible for setting/updating numeric protocol parameters could define bounds on these values	**INFO**

Functions like `AvaLido::setStakePeriod` and `AvaLido::setMinStakeAmount` could set lower and/or upper bounds for the accepted values. Such a change might require more initial thought but could protect against accidental mistakes when setting these parameters.

A4	AvaLido::claim might revert with ClaimTooLarge error	**INFO**

The function AvaLido::claim checks that the amount requested, `amount`, is not greater than `request.amountFilled - request.amountClaimed`. The user experience could be improved if in such cases instead of reverting the claimed `amount` was set to `request.amountFilled - request.amountClaimed`, i.e., the maximum amount that can be claimed at the moment. Such a change would require the `claim` function to return the claimed amount.

A5	Unused storage variables	**RESOLVED**

There are a few storage variables that are not used:

- `ValidatorSelector::minimumRequiredStakeTimeRemaining`
- `AvaLido::mpcManagerAddress`

A6	Unused `UnstakeRequest` struct field	**RESOLVED**

Field `requestedAt` of struct `UnstakeRequest` is not used.

A7	Function can be made external	**RESOLVED**

`OracleManager::getWhitelistedOracles` can be defined as external instead of public, as it is not called from any code inside the `OracleManager` contract.

A8	Gas optimization	**RESOLVED**

In function AvaLido::claimUnstakedPrincipals there is a conditional check that if true leads to the transaction reverting with InvalidStakeAmount().

```
function claimUnstakedPrincipals() external {
    uint256 val = address(pricipalTreasury).balance;
    if (val == 0) return;
    pricipalTreasury.claim(val);
    // Dedaub: the next line can be moved before the claim
    if (amountStakedAVAX == 0 || amountStakedAVAX < val)
        revert InvalidStakeAmount();

    // … (rest of the function's logic)
}
```

This check could be moved before the principalTreasury.claim(val) as it is not affected by the call. This would lead to gas savings in cases where the transaction reverts, as the unnecessary call to treasury would be skipped.

A9	AvaLido::stakePeriod contradicts with ValidatorSelector::minimumRequiredStakeTimeRemaining	**RESOLVED**

Even though ValidatorSelector::minimumRequiredStakeTimeRemaining is not used, it is defined as 15 days, while AvaLido::stakePeriod is defined as 14 days.

A10	Incorrectly spelt function name	**RESOLVED**

Function name hasAcceptibleUptime of the Types.sol contract should be corrected to hasAcceptableUptime.

A11	Compiler bugs	**INFO**

The code is compiled with Solidity 0.8.10, which, at the time of writing, has some known bugs, which we do not believe to 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.