On Oct 5, 2021, a potential exploit was reported via the Lido bug bounty program. The exploit could only be performed by a whitelisted node operator and allowed stealing a small share of user funds. The response timeline and details can be found in this Lido blog post.

The vulnerability is currently fully mitigated by pausing deposits from the protocol's buffer, no funds are at risk. However, a longer-term solution needs to be designed and implemented. Since we've been greenlit for the exploit disclosure by all affected parties, this post will detail the exploit mechanics and provide an initial list of possible mitigations for further discussion.

Nature of the exploit

The exploit is based on the fact that, as per the Ethereum consensus layer specification, the validator public key is associated with the withdrawal credentials (WC) on the first valid deposit that uses the public key. Subsequent deposits will use the WC from the first deposit even if another WC are specified.

While this design choice is not an issue for self-stakers, it opens an attack vector for delegated staking, including liquid staking protocols. These protocols, Lido among them, use protocol-controlled WC to ensure only the protocol can withdraw users' funds. In Lido's case, WC point to a smart contract managed by the DAO. The current Ethereum consensus layer design allows a node operator to associate the validator's public key with the node operator-controlled WC by front-running a deposit transaction sent by a protocol with another deposit transaction specifying the same public key, validator-controlled WC, and 1 ETH amount. The end state is a validator managing 1 ETH of node operator's funds and 32 ETH of users' funds, fully controlled and withdrawable by the node operator.

The exploit was initially reported on Ethereum Research forums a long time ago. The presence of this vulnerability in the Lido codebase is a long-term oversight.

Technical details

Lido distributes users' funds to node operators using a contract called NodeOperatorsRegistry

. This contract stores deposit data entries of all future validators that will be registered by the protocol. Each deposit data entry contains, among other fields, the validator public key and WC.

In other to participate in the protocol, node operators submit new deposit data entries containing their public keys to the registry. At that point, these freshly submitted deposit data entries cannot be used yet since they are not approved. Then, DAO members review these new deposit data entries, making sure they contain valid protocol-controlled WC, and vote for approval. Only after this vote passes do the new deposit data entries become enabled.

When users submit Ether to the protocol, it gets temporarily buffered in the Lido

smart contract. After a sufficient amount of Ether is accumulated, it can be deposited to the Ethereum DepositContract

by calling the permissionless Lido.depositBufferedEther()

function. This function selects a set of DAO-approved deposit data entries from the NodeOperatorsRegistry

contract and performs a deposit for each of them, attaching user funds from the buffer.

Due to the Ethereum consensus layer design described above, any node operator whose public key is used in this transaction can force the Ethereum consensus layer to attach an arbitrary WC to the public key instead of the ones specified in the DAO-approved deposit data entry. As a malicious node operator, do the following:

1. Monitor the mempool. As soon as there is a Lido.depositBufferedEther()

transaction in the mempool, go to the next step.

1. Generate new WC that you control.
2. From the deposit transaction, select those deposit data entries that use your public keys. For each of the selected entries, prepare a different deposit data entry that uses the same public key but the WC from step 2) and the amount equal to 1 ETH.
3. Prepare a transaction performing a set of deposits using the newly-crafted deposit data entries from the previous step. Send it in a way that ensures it's included in the chain before the transaction from step 1), e.g. by specifying a higher gas or using a direct channel to a miner.
4. Since the Lido.depositBufferedEther()

transaction is included in the chain after the pre-deposit transaction sent by the node operator, the consensus layer will process the legitimate deposits after the pre-deposits, leading to the legitimate deposits being associated with the node operator-controlled WC generated in 2). As a result, the node operator will host a set of validators associated with the

operator's WC, each managing operator's own 1 ETH and 32 ETH coming from the Lido protocol users.

Possible mitigations

a) Pre-depositing

In order for deposit data entries submitted by a node operator to be approved by the DAO, require the operator to pre-deposit 1 ETH with the correct protocol-controlled WC for each public key used in these deposit data entries.

Upon deposit data entry approval, 1 stETH is minted by the protocol to the node operator's reward address. Alternatively, a somewhat larger amount might be minted to compensate for the locking of the funds, e.g. 1.002 stETH.

Implementation complexity:

medium.

Operations complexity:

high.

Pros:

bulletproof.

Cons:

not trustless, capital inefficient, more complex protocol operations.

b) Risk cover

Lido or node operators keep a reserve of cover to mitigate a node operator going rogue (we don't think they will).

Implementation complexity:

low.

Operations complexity:

low.

Pros:

little to no changes to the protocol.

Cons:

risk of node operator compromise.

c) Using threshold signatures

Implement a threshold signature scheme similar to the one described in this Ethereum Research forum commentDeposit contract exploit - #5 by djrtwo - Casper - Ethereum Research.

Each deposit data entry is signed by a threshold signer committee that reconstructs and transfers a private key to the actual node operator (or an SSV committee) only after the deposit is made.

Implementation complexity:

high.

Operations complexity:

high.

Pros:

bulletproof.

Cons:

not trustless, fragile (requires private key transfer), more complex protocol operations.

d) Approving deposit contract Merkle root

Introduce a committee that follows the events on the execution layer and off-chain collects signatures verifying that no malicious pre-deposits were made yet by active node operators at deposit contract data root XXXXX.

Make the committee signatures and the deposit contract data root the parameters of the Lido.depositBufferedEther

function. Revert the transaction unless the current deposit data root is the same as the one passed to the function AND the committee has vetted that data root.

The committee can consist of our oracles OR our node operators OR completely unrelated entities.

Send deposit transactions with gas price sufficiently high for them to be included in one of the next several blocks to minimize the probability of the deposit contract data root being changed in the meantime. The current average time between deposit data root changes is 20+ blocks, the mean is 3 blocks, so in good weather, the solution will work even without paying an extremely high gas fee.

Implementation complexity:

medium.

Operations complexity:

medium.

Pros:

relatively easy to implement, doesn't add a lot of operational complexity.

Cons:

not trustless, might require paying a higher gas fee.

e) ZK proof of lack of malicious deposits

The same as previous mitigation, but instead of using the committee to vet the data root, use ZK proofs to prove to the Lido.depositBufferedEther

smart contract function that no malicious pre-deposits were made at deposit contract data root XXXXX (making the data root and the proof the arguments of the function).

Implementation complexity:

high.

Operations complexity:

low.

Pros:

trustless, adds no operational complexity.

Cons:

complex to implement; the data root might become stale by the time the proof is generated; might require paying a high gas fee.

f) EIP for the Ethereum consensus layer

Propose a change to the consensus layer specification, changing the handling of deposits made with the same public keys but different WC. For example, "if WC don't match and deposit value >= max effective balance, make a new validator in withdrawable status".

Implementation complexity:

very high.

Operations complexity:

low.

Pros:

proper long-term mitigation, trustless, adds no operational complexity.

Cons:

will take a lot of time and community effort to design and implement.

The proposed mitigations

At first sight, d) “Approving deposit contract Merkle root” seems to be the way to go mid-term and f) “EIP for the Ethereum consensus layer” long-term.

We’re preparing the detailed specification for d) and will post an update here as soon as it’s ready.

Please express your opinion about the mitigations described here and suggest new ones if you happen to come up with any.