P2P Economy Ltd

Lido Finance Security Assessment

Version: 2.1

December, 2020

Contents

Introduction 2
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary 3

Detailed Findings 5

Summary of Findings 6
- Node Operator Rewards Unevenly Leaked . . . 7
- Possible Market Manipulation . . . 9
- Outdated `dc4bc` Dependencies and Forks . . . 11
- Inconsistent QR and Decoding in dc4bc . . . 13
- DKG Susceptible to Rogue Key Attack . . . 15
- Stopped Validators Do Not Contribute Towards the Operator's Staking Limit . . . 16
- Insufficient Protection for Uninitialized Contracts . . . 18
- Oracle Reporting Delays Can Unfairly Impact Less Sophisticated Users . . . 20
- Biased Node Operator Assignment . . . 22
- Unnecessary Truncation When Calculating Fees . . . 24
- Accidental Submissions Can Lock ETH . . . 27
- Suboptimal Scrypt Parameterization for DC4BC Encryption Key . . . 28
- Storage Layout Makes Upgrades Difficult . . . 29
- Insecure `BytesLib` Dependency . . . 31
- Submit Always Returns Zero . . . 32
- Potential Gas Saving Optimisation in `Memutils.memcpy` . . . 33
- Miscellaneous Observations on Lido-DAO Codebase . . . 34
- Miscellaneous Observations on DC4BC Codebase . . . 35

A Vulnerability Severity Classification 36

Introduction

P2P Economy Limited (P2P.org) is a staking provider offering users a way of earning rewards by indirectly participating in the consensus logic of various cryptocurrency networks, such as Tezos, Cosmos and Kava, without having to worry about the overhead and costs associated with maintaining a dedicated staking infrastructure.

With the recent Ethereum 2.0 launch which introduces staking via the Casper FFG consensus mechanism, P2P.org built **Lido**, a set of protocols and tools, governed by a Decentralised Autonomous Organisation (De-Pool DAO), to provide a *liquid* staking service. Users will be able to deposit their Ether into the Lido system and obtain a tradeable, derivative token (stETH) in return.

Sigma Prime was approached by P2P.org to perform a security assessment of Lido.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the assessed system. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Lido platform. A summary followed by a detailed review of the discovered vulnerabilities is then given, which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities found within the code.

Overview

Lido is an eth2 staking platform allowing users to deposit **ETH** into a smart contract, and receive **stETH** in return. This staking service is governed by a Decentralized Autonomous Organizations, powered by Aragon, which is responsible for selecting node operators to be validators on the eth2 network, on behalf of Lido users. The Lido DAO also accumulates service fees to be spent on future upgrades.

The **dc4bc** (**D**istributed **C**ustody **for** the **B**eacon **C**hain) project allows trusted network participants (i.e. initial Lida DAO members) to perform a distributed key generation (DKG) ceremony using threshold signatures.

Security Assessment Summary

This review initially targeted the following commits:

- `lidofinance/lido-dao` : f51a8bc (Release v0.1.0-rc.1)
- `lidofinance/dc4bc` : 584b855
- `dc4bc` 's dependency `corestario/kyber` : 03e2e68 (Release v1.6.0)

A subsequent review round targeted:

- `lidofinance/lido-dao` : d4171a1 (Release v0.2.0)

A final review targeted:

- `lidofinance/lido-dao` : ad4b2f6 (Release v0.2.1-rc.0)

Retesting activities targeted:

- `lidofinance/lido-dao` : ad4b2f6 (Release v0.2.1-rc.1)
- `lidofinance/dc4bc` : 1d4f80d (Release 0.1.4)

Note: some issues raised in this report might only affect specific versions (i.e. these issues may have been fixed in subsequent releases). Each issue is labelled with the corresponding version it was identified in. Furthermore, Aragon dependencies were excluded from the scope of this review.

This assessment focused on identifying any vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focuses on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

Fuzzing activities leveraging go-fuzz have been performed by the testing team in order to identify panics within the code in scope. `go-fuzz` is a coverage-guided tool which explores different code paths by mutating input to reach as many code paths possible. The aim is to find memory leaks, overflows, index out of bounds or any other panics.

Specifically, the testing team produced the following fuzzing targets, shared with the development team:

- `encodedecodechunk`
- `encodedecodeqr`
- `encodedecodeqrhex`
- `encodedecodeqrhexzxing`
- `encodeqr`
- `fsminstance`
- `pngqr`

These fuzzing targets have all been shared with the development as a byproduct of this security review. Execution and instrumentation can be done using a `Makefile`, by simply running `make run-fuzz-${TARGET-NAME}` (targets list and detailed instructions are available inside the `Makefile`).

The testing team identified a total of eighteen (18) issues during this assessment, of which:

- Five (5) are classified as medium risk,
- Eight (8) are classified as low risk,
- Five (5) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Lido Finance platform. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the code base, including comments not directly related to the security posture of Lido-Dao or `dc4bc`, are also described in this section and are labelled as *"informational"*.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team;
- ***Resolved:*** the issue was acknowledged by the project team and the affected code as been updated, or relevant controls implemented, to mitigate the related risk;
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
LID-01	Node Operator Rewards Unevenly Leaked	Medium	Resolved
LID-02	Possible Market Manipulation	Medium	Closed
LID-03	Outdated `dc4bc` Dependencies and Forks	Medium	Resolved
LID-04	Inconsistent QR and Decoding in dc4bc	Medium	Resolved
LID-05	DKG Susceptible to Rogue Key Attack	Medium	Resolved
LID-06	Stopped Validators Do Not Contribute Towards the Operator's Staking Limit	Low	Open
LID-07	Insufficient Protection for Uninitialized Contracts	Low	Resolved
LID-08	Oracle Reporting Delays Can Unfairly Impact Less Sophisticated Users	Low	Closed
LID-09	Biased Node Operator Assignment	Low	Closed
LID-10	Unnecessary Truncation When Calculating Fees	Low	Resolved
LID-11	Accidental Submissions Can Lock ETH	Low	Resolved
LID-12	Suboptimal Scrypt Parameterization for DC4BC Encryption Key	Low	Resolved
LID-13	Storage Layout Makes Upgrades Difficult	Low	Closed
LID-14	Insecure `BytesLib` Dependency	Informational	Closed
LID-15	Submit Always Returns Zero	Informational	Resolved
LID-16	Potential Gas Saving Optimisation in `Memutils.memcpy`	Informational	Resolved
LID-17	Miscellaneous Observations on Lido-DAO Codebase	Informational	Closed
LID-18	Miscellaneous Observations on DC4BC Codebase	Informational	Closed

LID-01	Node Operator Rewards Unevenly Leaked		
Asset	`v0.2.1-rc.0: contracts/v0.4.24/nos/NodeOperatorRegistry.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

As updated validator balances are reported to Lido, a portion of the staking rewards are distributed as a fee to the staking node operators. This is done by transferring the total reward allocated to operators to the `NodeOperatorRegistry`, which then distributes it amongst relevant node operators.

This distribution occurs in `NodeOperatorRegistry.distributeRewards()`, where the reward value is divided amongst all active node operators, proportional to the number of active validators they control. However, this integer division invariably results in some small remainder that stays with the `NodeOperatorRegistry`.

Because the current `distributeRewards()` implementation only ever dispenses the current reward amount, as opposed to its total balance, this residual amount will accumulate. The leaked value has an upper bound of k stETH wei per reward period (day), where k is the number of active node operators. In the worst case, where daily rewards are quite low and there are a large number of staking providers, this can represent a not-insubstantial proportion of the total reward.

While this has no direct security implications, it represents lost value and may affect the accuracy of analysis of the economic incentive scheme.

While this leaked stETH can be easily recovered to the vault, or redistributed in subsequent updates to Node-OperatorRegistry or Lido, it is more complicated to fairly distribute amongst node operators based on historical validator activity.

We also note that this truncation does not evenly affect node operators. Because each will likely have a different number of active validators, the leaked remainder will also be different (anywhere in the range of $[0, 1)$ StETH wei). Provided the regular per-operator reward is well over 1 StETH wei, this is likely acceptable.

Recommendations

Consider distributing the `NodeOperatorRegistry`'s entire StETH balance, rather than just the recently minted `mintedFee`[1]. This could be achieved by passing the result of `token.balanceOf(address(operatorsRegistry))` to `operatorsRegistry.distributeRewards()` or otherwise modifying `NodeOperatorRegistry.distributeRewards()` to divide its entire balance.

If the extra `balanceOf()` call uses excessive gas, it may be sufficient to do this less often (e.g. once per week).

Also ensure `NodeOperatorRegistry.allowRecoverability()` is modified to disable the stored StETH from getting "intercepted" to the Vault.

[1] `Lido.sol:625`

Alternative Considerations

For entirely fair distribution (should even truncation be important), one could calculate a per-validator reward with a single division `uint256 perValReward = _totalReward.div(effectiveStakeTotal)`. Then, all node operators will experience a larger but even truncation to their rewards (which would be included in subsequent distributions).

You could also consider a round-robin system, where a single node operator receives all the remainder, but this would be less fair for little benefit.

Resolution

This issue has been tracked in Issue #237 and resolved in PR #236. The operator reward distribution is now reported by `NodeOperatorRegistry` to `Lido`, which directly distributes the rewards to recipients so no residual `stETH` can accumulate. Operator rewards are first calculated "per validator" before being distributed, such that each operator experiences an equal per-validator truncation, with the residual `stETH` sent to the treasury.

LID-02	Possible Market Manipulation		
Asset	Malicious Node Operators and Lido DAO Members		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Note: *This review did not focus on any analysis of economic incentives or their viability. However, this potential issue was recognised.*

While long term economic incentives appear to encourage "good behaviour" from staking node operators and the DAO governance system, the testing team has identified some potential scenarios where a malicious node operator or a voting majority of the DAO could harm the platform for short-term gain.

As part of the current trust model, malicious node operators cannot steal funds, but can cause their provided stake to be slashed. Similarly, the DAO has the ability to vote to upgrade code or increase fees such that all subsequent rewards are provided to DAO token holders (in the treasury).[2] Although the DAO governers and node operators cannot directly benefit from this (node operators would quickly get deactivated, and future ETH submissions could be reduced), these actions can cause the market value of stETH to drop and increase distrust in the platform.

As stETH is an ERC20-like token, it can potentially be integrated into a wide range of DeFi applications and platforms. Several markets and derivative products could allow users to "short" stETH, profiting off the fall in stETH valuation. Because this is an indirect method of profit, it can be difficult to distinguish malicious from negligent behaviour.

Similarly, shortly before Eth2 withdrawal is introduced as part of Phase 1.5, dumping the stETH price could allow for cheap purchase of stETH to be burnt and exchanged for ETH.

Declining future prospects (e.g. unrelated indications that participation in Lido will drop or stagnate) may also encourage this short-term behaviour.

Recommendations

Be careful to advise DAO members to carefully consider potential consequences when adding new node operators, adjusting staking limits, or making DAO tokens available for purchase.

In particular, try to evaluate and balance potential gains of a malicious entity against long-term financial incentives, reputational or legal costs, and other collateral.

The potential gains from such behaviour can vary depending on the amount of stETH liquidity available in DeFi protocols.

Also consider that the effect of a malicious node operator on the market may be exaggerated past their staking limit. For example, given appropriate publicity, a malicious operator slashing 10 validators may cause similar fear to one controlling 100.

[2] For this to occur, a voting majority of the DAO would need to agree on the changes, or delegate the `MANAGE_FEE` role.

Resolution

The Lido team acknowledge this issue. The system design is geared to alleviate this risk by encouraging careful selection of well-known, reputable node operators.

DAO members have incentive to encourage the long-term success of the platform so are expected to carefully vet any prospective node operators. Thanks to this non-anonymous design, the DAO can effectively enforce distribution of the staked ETH across diverse node operators, setting limits appropriate to the level of trust.

In contrast, a design that involves explicit collateral and anonymous stakers may have difficulty setting an appropriate collateral amount that protects against indirect abuses.

LID-03	Outdated `dc4bc` Dependencies and Forks		
Asset	`dc4bc/go.mod`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In its `go.mod` file, dc4bc specifies several outdated dependencies and the following replace directive:

```
replace golang.org/x/crypto => github.com/tendermint/crypto v0.0.0-20180820045704-3764759f34a5
```

This directive replaces all imports of the standard `golang.org/x/crypto` library with the unmaintained `tendermint/crypto` fork, including in all dependencies[3]

Fortunately, no relevant security vulnerabilities were found that affect code used by dc4bc.[4]

We note that the random seed functionality exposed by the fork is not needed by dc4bc or any dependencies.[5]

Refer to `./tools-output/dc4bc-deps.out` and `./tools-output/kyber-deps.out` provided to the development team in addition to this report.

While cryptography libraries are quite heavily scrutinized and the dc4bc use-case prioritizes backwards compatibility over availability/performance, it is of paramount importance that any security issues in dependencies are reviewed and acknowledged. The security risk identified here is less about the current dependency versions, and more the update and security alerting processes in place for dc4bc.

[3] `corestario/kyber` uses `x/crypto` in the following files:

sign/bdn/bdn.go: "golang.org/x/crypto/blake2s"

xof/keccak/keccak.go: "golang.org/x/crypto/sha3"

share/vss/pedersen/dh.go: "golang.org/x/crypto/hkdf"

share/vss/rabin/dh.go: "golang.org/x/crypto/hkdf"

xof/blake2xb/blake.go: "golang.org/x/crypto/blake2b"

xof/blake2xs/blake.go: "golang.org/x/crypto/blake2s"

pairing/bn256/suite_test.go: "golang.org/x/crypto/bn256"

encrypt/ecies/ecies.go: "golang.org/x/crypto/hkdf"

[4] Refer to `golang.org/x/crypto` security issues since 2018-08-20. We noted some panics in `x/crypto/ssh` but nothing used by dc4bc:
https://github.com/golang/go/issues?page=1&q=label%3ASecurity+is%3Aclosed+crypto
https://www.cvedetails.com/vulnerability-list.php?vendor_id=14185&product_id=0&version_id=0&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=0&cvssscoremax=0&year=0&cweid=0&order=1&trc=31&sha=28620af5fce730868aaad8eb6b0b82bc3b861475

[5] Refer to the following to note changes introduced by the fork, and updates since:
https://github.com/golang/crypto/compare/master...tendermint:master
https://github.com/tendermint/crypto/compare/3764759f34a542a3aef74d6b02e35be7ab893bba...golang:master

Recommendations

Remove the unneeded replace directive for `golang.org/x/crypto`.

Update relevant dependencies, and consider introducing monitoring to alert for vulnerabilities in associated dependencies.

Resolution

This issue was remediated in PR #74, which removed the "replace" directive.

LID-04	Inconsistent QR and Decoding in dc4bc		
Asset	`dc4bc/qr/qr.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

During fuzzing of the QR encoding and decoding functionality, the team identified several results where the encoded message did not match the decoded one.

Some discrepancies were identified in how the QR libraries handled null bytes but, even when hex encoding prior to passing to the QR, and using alternative libraries, the inconsistencies persisted.

Refer to fuzzing targets and crashes provided to the development team along with this report.

The security risk is less associated with a malicious exploit, and more distributing malformed signatures.

Recommendations

As the exact bugs associated with the fuzzing results could not be identified, the testing team recommends the following:

- Hex-encode the message prior to encoding in QR, to avoid text encoding inconsistencies and null characters while allowing reasonable compression.
- Display a secure hash of each QR payload (dechunked) on the hot and airgapped nodes - after the hot node
- Consider prompting to confirm that these hashes match before proceeding.
- Investigate fuzzing results and consider alternative QR encoding/decoding libraries if possible.

Resolution

This issue was resolved in PR #83 and PR #90.

In PR #83, the intermediate serialization step[6] was changed from using gob encoding to JSON. The Golang JSON marshal always outputs valid utf8,[7] so the encoded QR message will never contain null bytes or other values that can be inconsistently handled. The default JSON marshalling behaviour encodes `[]byte` as a base64 string,[8] so there is no risk of the `Chunk.Data` field being modified in order to coerce the result into valid UTF8.

Because JSON is human readable, this also has the benefit in allowing users to more easily verify that the decoded message is what they expect.

[6]This is used to convert from Go structs to a serialized representation that can then be encoded into a QR
[7]Explained in https://golang.org/pkg/encoding/json/#InvalidUTF8Error
[8]https://golang.org/pkg/encoding/json/#Marshal

Introduced in PR #90, QR images are now read via a standalone browser-app `https://github.com/lidofinance/qr-scanner`, which uses instascan to scan and decode QR images concatenating the chunks into the original JSON message that is imported via the airgapped `"read_operation"` command.

LID-05	DKG Susceptible to Rogue Key Attack		
Asset	dc4bc DKG procedure		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The currently implemented Distributed Key Generation (DKG) protocol is susceptible to a rogue key attack such that, given a poorly chosen threshold parameter t, fewer than t malicious participants can collude to gain full knowledge of the shared secret.

In particular, when the DKG protocol involves n members, of which any $t+1$ can recover the distributed key or sign messages, $n-t+1$ malicious participants can perform the rogue key attack.

Refer to `https://blog.sigmaprime.io/dkg-rogue-key.html` for more information.

Recommendations

When choosing DKG parameters for generating and distributing the staking withdrawal key, carefully ensure that $\min(t+1, n-t+1)$ is above the security threshold for an acceptable number of malicious members.

If a t closer to n is desired, alternative mitigations could involve a "commit-reveal" step during DKG, where all participants must publish secure hashes of their polynomial before any polynomial is revealed.

Resolution

The Lido team have acknowledged this issue and will select an appropriate threshold value for their key generation ceremony, such that this attack is mitigated.

LID-06	Stopped Validators Do Not Contribute Towards the Operator's Staking Limit		
Asset	`v0.2.1-rc.0: contracts/v0.4.24/Lido.sol`		
Status	Open		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Lido-DAO Node Operators are registered with a "Staking Limit", which restricts the number of validators that the Node Operator is allowed to run on behalf of the DAO. While set in `NodeOperatorRegistry.sol`, this limit is enforced only in `Lido.sol`:

```
uint256 stake = entry.usedSigningKeys.sub(entry.stoppedValidators);
if (stake + 1 > entry.stakingLimit)
    continue;
```

`Lido._ETH2Deposit()`

As shown, stopped validators (identified via `NodeOperatorRegistry.reportStoppedValidators()`) do not contribute to an operator's `stakingLimit`. Indeed, a Node Operator who had previously reached the limit would be immediately eligible to receive an additional validator after one was reported stopped.

Based on the comment at `NodeOperatorRegistry.sol:51` and usage, the "stopped validator" status is used to indicate a slashed or exited validator, as opposed to one that is temporarily offline. Prior to the ability to withdraw from Eth2 (expected in Phase 1.5[9]), there is no sound reason for a staking service to perform a voluntary exit or get slashed.

While it may be possible to reduce the operator's staking limit at the same time as reporting the stopped validator, this may be difficult to do atomically or without the delays associated with an additional DAO vote.

Although node operators are expected to be heavily vetted and trustworthy, a malicious operator could (in certain circumstances), slash more than its `stakingLimit` of validators. Because the current validator allocation prioritises operators with fewer active validators (see TODO ref), new deposits are more likely allocated to the malicious operator.

The `Pausable` mechanism can provide some protection against this, but requires external intervention and should be considered a last resort. We would argue that, in these circumstances, a safer default behaviour would be for the stopped validators to contribute to the operator's staking limit.

Recommendations

Consider allowing stopped validators to count towards an operator's staking limit (though this will be worth reconsidering when withdrawals are possible). With this, the `stakingLimit` can be interpreted as the total number of deposits entrusted to the operator.

In the event of an unexpected withdrawal or slashing (indicative of staking operator neglect or malice), the staking operator can submit an explanation to the DAO and request an increased `stakingLimit`.

[9] https://ethereum.org/en/eth2/staking/

Resolution

The Lido team acknowledge this issue and plan to release a fix in a subsequent update (post-deployment).

LID-07	Insufficient Protection for Uninitialized Contracts		
Asset	`v0.2.1-rc.0`: `contracts/v0.4.24/Lido.sol` & `contracts/v0.4.24/stETH.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Several state changing functions defined in Lido.sol and StETH.sol are not protected by an `isInitialized` modifier. Prior to `v0.2.0-rc.0` this was not a problem, as they all performed external function calls that would revert when uninitialized. As of `v0.2.0-rc.0`, however, these functions no longer make external calls and can succeed without error.

Most relevant is `Lido._submit()`, which is the implementation for the external `Lido.submit()` and fallback functions.

This affects more than non-standard deployments (where it may be possible to interact with an uninitialized Aragon proxy). Indeed, it should be possible to submit funds without error to a base Lido contract, that was correctly *petrified*[10] during deployment. The *petrified* status is intended to disable any state changing functionality in the base logic contract (which should be used only as the source of logic per the DelegateProxy pattern). However, the `Petrifiable` implementation only disables functions that have an `isInitialized` modifier (including those with an `auth` or `authP` modifier).

Because `Lido.transferToVault()` only allows the recovery of unbuffered ETH, any ETH successfully submitted to the petrified base contract would be lost entirely.

The security risk is deemed low, as this does not appear to be exploitable by a malicious attacker to affect other users. The primary impact appears that a user can accidentally lock their own funds by submitting to a base Lido contract. At most, a malicious entity may trick a user into submitting ETH to the base contract, from which it could not be recovered.

Recommendations

Consider adding `isInitialized` modifiers to any state changing functions that are not already protected by `auth()` `authP()`. In particular, the following functions:

- `Lido._submit()` (or the fallback and `Lido.submit()`)
- `StETH._transfer()`
- `stETH._approve()`

To protect the StETH functions, this would need to inherit from AragonApp (or `Initializable` at the least). For Lido to compile after StETH inherits from AragonApp, the order of inheritance would need to be changed

[10] See https://hack.aragon.org/docs/aragonos-ref#application-lifecycle-guarantees

from `contract Lido is ILido, IsContract, StETH, AragonApp`
to `contract Lido is ILido, IsContract, AragonApp, StETH`.[11]

Also consider adding the modifier for any externally accessible function that is not `view` or `pure` to protect against unintentional exposure in future updates. For example, although `Lido._depositBufferedEther()` should revert when not initialized due to an external call to `NodeOperatorRegistry.assignNextSigningKeys()`, a later update may unintentionally allow this to succeed, sending buffered ETH to the zero address.

Because these functions already use the `whenNotStopped` modifier, an alternative remediation could be to change the `Pausable` definition such that the uninitialized value counts as "Paused" (i.e. `isStopped() == true` by default). `Lido.initialize()` would then include a call to `_resume()`.

Resolution

This issue was remediated in PR #253, which set `Pausable` contracts to paused by default (`isStopped() == true`). `Lido` then resumes during initialization and existing authentication protections prevent the DAO from resuming the petrified base contract.

[11] With the existing inheritance order, compilation would fail with an "Linearization of inheritance graph impossible" error. See https://docs.soliditylang.org/en/v0.7.4/contracts.html#multiple-inheritance-and-linearization for explanation. Though this documents a newer solidity version, it is still applicable.

LID-08	Oracle Reporting Delays Can Unfairly Impact Less Sophisticated Users		
Asset	`v0.2.1-rc.0: contracts/v0.4.24/oracle/LidoOracle.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The LidoOracle contract is responsible for reporting the state of Eth2 to the Lido system, and directly controls the amount of stETH in circulation.

Because the LidoOracle can only report once per `frame` (day by default), there can be a sizeable delay before the Lido contracts see any dumps in balance due to slashing events.

As such, more sophisticated "whales" (with their own view of the Eth2 state) can sell their stETH before the balance changes are visible on-chain, thus transferring their penalties and much of the risk in owning stETH onto unsuspecting third parties.

This delay may make stETH valuation more difficult, exaggerating price fluctuations which could be taken advantage of. In a hypothetical scenario, stETH has some consistent value proportional to its associated Eth2 ETH. Should a balance drop occur, stETH is no longer one-to-one with its backed ETH for the duration of the oracle delay. Should the market be aware of this, in order to keep the value of the staked ETH consistent, the stETH price would artificially drop for this duration.

Because node operators are fairly trusted and any slashing events should be rare, this has been deemed a low security risk.

Recommendations

While the timeliness of oracle reporting can be improved, much of the potential for abuse is associated with uneven knowledge of the Eth2 state. It is not possible to entirely remove oracle reporting delays,[12] so education and communication can provide a sufficient, "due-diligence" mitigation.

Such a mitigation could include clear documentation explaining the risks associated, advising careful stETH purchases when balance decreases are pending, as well as a recommended communication platform that can notify users when such a balance decrease is expected. By providing this status notification, Lido can "even the playing field" and minimize market panic in the event of a slashing. This communication can also be helpful because the slashing penalties are not immediately applied to the balance.[13] Thus, additional balance decreases should be expected for the 36 days following a slashing event.

To reduce market fluctuations due to inconsistent balance information, consider implementing a "flux monitoring" oracle solution, where the oracles should report more frequently than once-per-day in the event of large balance fluctuations. This can have added gas savings benefits, where oracles may not need to report as often. [14]

[12] In an extreme case, it may be profitable to front-run trades while the oracle reporting transaction is in the mempool.
[13] See https://benjaminion.xyz/eth2-annotated-spec/phase0/beacon-chain/#epochs_per_slashings_vector.
[14] ChainLink provides some similar functionality with their "FluxMonitor" https://chainlinkgod.medium.com/scaling-chainlink-in-2020-371ce24b4f31

Resolution

The Lido team acknowledges the issue and plan to implement the following mitigations:

- A comprehensive dashboard to provide real-time status information to stakers.
- Educate Oracle operators to proactively notify the DAO should they detect an unusual situation.

Any changes to the reporting protocol can be implemented as upgrades to the `LidoOracle` contracts.

LID-09	Biased Node Operator Assignment		
Asset	`v0.2.1-rc.0: contracts/v0.4.24/nos/NodeOperatorRegistry.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The algorithm used to select signing keys from available node operators can sub-optimally prefer new and less trusted node operators.

Because Node Operators only receive staking rewards for their active signing keys, this also affects the distribution of rewards amongst operators.

The current algorithm attempts to distribute validators equally across all node operators, irrespective of their level of trust (where indicators of trust include an operator that has been staking for longer and/or have a higher staking limit).

Consider a scenario where 2 node operators exist: A and B. A is more highly trusted by the DAO, so has a `stakingLimit` of 500 while B's limit is 100. Each is currently running at full capacity (i.e. A is running 500 validators, and B 100). The DAO increases both operators' limit by 50 (A:550, B:150). In this scenario, A will not be assigned any more validators until B has again reached full utilization at 150 validators i.e. all of the next deposits will be assigned to B, even though A is more highly trusted by the DAO.

Recommendations

Consider modifying the signing key selection method (currently implemented in `NodeOperatorRegistry.assignNextSigningKeys()`) such that less trusted node operators (recently added and with a small staking limit) are not heavily preferenced over long trusted staking service providers, when both have available signing keys.

Some alternative selection methods to consider include:

- Most fair would be to pick the next key from the node operator that has the lowest "validator utilization rate" (percent capacity) i.e. one with available signing keys and smallest value for `usedSigningKeys/stakingLimit`.

  This would allow popular staking providers to receive validators before less trusted ones reach their staking limit. This method equalizes each operator's capacity. However, calculations would likely require more gas.
- Round robin selection. Would likely involve storing the index of the last selected operator.

  This would reward "older" node operators more heavily and would equalize the rate at which each operator's capacity is reached.
- Random selection. A more even distribution on average, but has a chance of being biased and difficult to implement.

Resolution

The Lido team have acknowledged the bias and clarified that this is an intentional design decision. By prioritizing new node operators, the selection algorithm aims to decentralize the pooled funds as widely as possible across staking entities. This also has the effect of "onboarding" new node operators such that they can more quickly reach a profitable number of validators.

As this is a consciously introduced bias, the Lido team can appropriately educate the DAO such that new, less trusted, node operators will be configured with an initially lower `stakingLimit`.

LID-10	Unnecessary Truncation When Calculating Fees		
Asset	`v0.2.1-rc.0 : contracts/v0.4.24/Lido.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

As reward balances are updated, Lido mints additional token shares representing a portion of rewards taken as a fee. While the calculation (defined in `Lido.distributeRewards()` line [574]) correctly accounts for the devaluing effect of minting additional shares, it includes more truncating division[15] than necessary. In most cases, this results in slightly less StETH being distributed to fee recipients than intended by the Lido DAO.

In particular, line [256]

```
uint256 feeInEther = _totalRewards.mul(_getFee()).div(10000);
```

includes a division, and this result is used in the following calculation to determine `sharesToMint`.

This issue is not directly exploitable and probably not a large discrepancy, hence the low risk rating. However, it may be possible for the unexpected discrepancy to affect the validity of any economic incentive analysis.

Detailed Analysis

Although obvious that a single truncating division generally produces a smaller result than exact division, the impact is less clear when part of a larger calculation. As such, we ask "What is the effect of the truncated `feeInEther` on the value of `sharesToMint`?" Is it larger or smaller?

For this, we introduce the following notation:

p : the result of `_getFee()`, the staking reward fee rate (basis points).

r : `_totalRewards` the amount of new rewards generated by Lido staking (ETH).

f : `feeInEther` the concrete staking reward fee (ETH).

t : the result of `_getTotalPooledEther()` (ETH). This includes the most recently reported balance changes and `_totalRewards`.

s : `prevTotalShares` the total number of shares prior to this minting.

x : `sharesToMint` the amount of *new* shares to be minted to correspond to the desired fee.

When accounting for truncated division, we have:

$$f_{\mathbb{Q}} = \frac{rp}{10000} \qquad \text{(exact division)}$$

$$f_{\mathbb{Z}} = \lfloor f_{\mathbb{Q}} \rfloor = \left\lfloor \frac{rp}{10000} \right\rfloor \qquad \text{(integer division)}$$

[15] Division in Solidity is *integer* division, where the fractional part (remainder) of the result is discarded (or truncated).

The current implementation of the `sharesToMint` calculation is thus most correctly expressed as

$$x_{\mathbb{Z}} = \left\lfloor \frac{f_{\mathbb{Z}} s}{t - f_{\mathbb{Z}}} \right\rfloor$$

which is not the same as

$$x_{\mathbb{Q}} = \left\lfloor \frac{f_{\mathbb{Q}} s}{t - f_{\mathbb{Q}}} \right\rfloor$$

While we cannot directly compute $x_{\mathbb{Q}}$ in Solidity using $f_{\mathbb{Q}}$, it can be done by rearranging the calculation such that the extra division is not required, assuming no overflow

$$\begin{aligned} x_{\mathbb{Q}} &= \left\lfloor \frac{\frac{rp}{10000} s}{t - \frac{rp}{10000}} \right\rfloor \\ &= \left\lfloor \frac{rps}{10000t - rp} \right\rfloor \end{aligned}$$

With regards to impact, we now ask what is the relation between $x_{\mathbb{Z}}$ and $x_{\mathbb{Q}}$?

Proof ($x_{\mathbb{Z}} \leq x_{\mathbb{Q}}$) :

(Note all quantities are positive integers, so multiplication maintains the direction of any inequalities.) Since

$$f_{\mathbb{Q}} - 1 < f_{\mathbb{Z}} \leq f_{\mathbb{Q}},$$

then

$$\begin{aligned} f_{\mathbb{Z}} s &\leq f_{\mathbb{Q}} s && (\times\, s) \\ f_{\mathbb{Z}} s(t - f_{\mathbb{Z}}) &\leq f_{\mathbb{Q}} s(t - f_{\mathbb{Z}}) && (\times\, (t - f_{\mathbb{Z}}) \text{ given } t - f_{\mathbb{Z}} \geq 0) \end{aligned}$$

and

$$\begin{aligned} t - f_{\mathbb{Q}} &\leq t - f_{\mathbb{Z}} \\ f_{\mathbb{Z}} s(t - f_{\mathbb{Q}}) &\leq f_{\mathbb{Z}} s(t - f_{\mathbb{Z}}). && (\times\, f_{\mathbb{Z}} s) \end{aligned}$$

Thus

$$\begin{aligned} f_{\mathbb{Z}} s(t - f_{\mathbb{Q}}) &\leq f_{\mathbb{Z}} s(t - f_{\mathbb{Z}}) \leq f_{\mathbb{Q}} s(t - f_{\mathbb{Z}}) \\ f_{\mathbb{Z}} s(t - f_{\mathbb{Q}}) &\leq f_{\mathbb{Q}} s(t - f_{\mathbb{Z}}). \end{aligned}$$

Rearranging, we have

$$\frac{f_{\mathbb{Z}} s}{t - f_{\mathbb{Z}}} \leq \frac{f_{\mathbb{Q}} s}{t - f_{\mathbb{Z}}}$$

and therefore

$$x_{\mathbb{Z}} \leq x_{\mathbb{Q}} \qquad \blacksquare$$

The truncated division produces a smaller result so fewer fees are distributed.

Recommendations

Consider avoiding the `div(10000)` in `Lido.sol:576` until later.

Instead, perform the following calculation, which is otherwise equivalent but contains only a single integer division.

$$\texttt{sharesToMint} = \frac{\texttt{feeBasis} \times \texttt{totalRewards} \times \texttt{prevTotalShares}}{(10000 \times \texttt{newTotalPooledEther}) - (\texttt{feeBasis} \times \texttt{totalRewards})}$$

Here, the variable naming is kept consistent with `Lido.distributeRewards()` and `feeBasis` is the result of `_getFee()`.

Resolution

This issue was tracked in Issue #241 and resolved in PR #236[16].

[16] See changes to `Lido.distributeRewards()`

LID-11	Accidental Submissions Can Lock ETH		
Asset	`v0.2.1-rc.0: contracts/v0.4.24/Lido.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `Lido` contract's current fallback function treats any unrecognised function call as a submission attempt.

This may result in users accidentally locking their funds if they mistook the Lido contract for something else, or accessed it through an inaccurate interface.

Recommendations

Consider adding a `require(msg.data.length == 0);` statement to the fallback function, to protect against accidental submissions by people calling non-existent functions.

Resolution

This was resolved in PR #238.

LID-12	Suboptimal Scrypt Parameterization for DC4BC Encryption Key		
Asset	`dc4bc/airgapped/encryption.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The current scrypt parameterization appears based on the 2017 recommendations for interactive logons.[17] While this is likely quite resistant to dictionary bruteforce attack, with a derivation time on the order of 100ms, a much higher parameterization is advisable given the use-case and highly valuable data.

The use-case is for long-term storage of highly valuable info (Eth2 withdrawal keys). As such, performance considerations associated with interactive logon are not a concern and a longer key derivation delay is acceptable.[18]

Recommendations

Consider performing a benchmarking test on representative hardware. Tune the scrypt N parameter such that key derivation takes an acceptable amount of time (e.g. 3 seconds).

Resolution

This issue was sufficiently remediated in PR #74 and PR #105.

In PR #74 the difficulty parameter N was increased from 2^{15} to 2^{16}.

PR #105 changed the long term storage medium from USB keys containing the encrypted data to a paper wallet (or similar) containing a BIP39 mnemonic. This mnemonic, used as a random seed, can be used to regenerate the relevant keys. As such, the database containing encrypted secrets is needed only for the duration of the ceremony and is recommended to be held ephemerally in memory (or the storage medium destroyed upon completion of the ceremony).

[17] Described in `https://godoc.org/golang.org/x/crypto/scrypt#Key`

[18] Refer to the following for useful info:
`https://github.com/golang/go/issues/22082`
`https://blog.filippo.io/the-scrypt-parameters`

LID-13	Storage Layout Makes Upgrades Difficult		
Asset	`v0.2.1-rc.0: contracts/v0.4.24/Lido.sol` & `contracts/v0.4.24/StETH.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The current layout of state variables defined in `Lido.sol` and `StETH.sol` greatly increases the complexity of any upgrades to the resulting contract. Because (as of `lido-dao v0.2.1-rc.0`) the `Lido` contract inherits from `StETH`, any changes to the type or number of state variables in `StETH` can break the layout of any conventionally defined state variables in `Lido.sol`.

This provides no security risk to the current Lido platform, but the increased upgrade complexity can greatly increase the risk of vulnerabilities unintentionally introduced by future updates.

Recommendations

For any state variables whose definition could change in future updates (particularly those in StETH.sol), heavily consider changing them to make use of the unstructured storage pattern.

In `v0.2.0`, the following state variables were of note:

- `Lido.sol:75 withdrawalCredentials`
- `StETH.sol:38 lido`
- `StETH.sol:44 _shares`
- `StETH.sol:45 _totalShares`
- `StETH.sol:47 _allowed`

In `v0.2.1-rc.0` following state variables are of note:

- `StETH.sol:62 shares`
- `StETH.sol:67 allowances`

Also be careful to ensure consistent storage layout across both contract definitions in future updates.

Resolution

This has been partially resolved in `v0.2.1-rc.0` where, although the `StETH` inheritance increased complexity, the conventionally defined `Lido.withdrawalCredentials` was moved to unstructured storage.

As `Lido` now defines no (non constant) conventionally stored state variables, changes to variables defined in `StETH` will only need to be internally consistent to preserve the storage layout. Similarly, `v0.2.1-rc.0` removed the `lido` and `_totalShares` state variables defined in `StETH.sol`.

The Lido team will need to ensure future upgrades retain a consistent storage layout. If an upgraded version of the `Lido` contract were to define conventional state variables, no new state variables could be defined in an upgraded version of `StETH` while retaining the same inheritance relationship.

Although more cumbersome to code with, the unstructured storage pattern allows upgrades to the token and management logic to remain encapsulated even with `Lido` inheriting from `StETH`. The testing team also acknowledge that it is more complicated to implement the unstructured storage pattern for more complicated reference types, like the `shares` and `allowances` mappings, and this complexity can introduce its own security risk.

LID-14	Insecure `BytesLib` Dependency
Asset	`v0.2.1-rc.0`: `contracts/v0.4.24/Lido.sol` & `contracts/v0.4.24/nos/NodeOperatorRegistry.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `solidity-bytes-utils` `v0.0.6` dependency has a critical vulnerability in the `BytesLib.slice()` method.

See `https://github.com/GNSPS/solidity-bytes-utils#important-fixes-changelog` for more info.

The current codebase does not expose this vulnerability, as user-supplied inputs are never passed to `BytesLib.slice()`.

Recommendations

Ensure any future updates to Lido avoid introducing user-supplied input to the arguments for `BytesLib.slice()`.

Consider introducing documentation, automation or procedure to check that future updates are consistent with this recommendation.

When possible (i.e. when AragonOS can migrate to a newer version of Solidity), update to a newer version of the `solidity-bytes-utils` dependency.

Resolution

The Lido team have acknowledged this in Issue #247 and will be careful to continue to avoid passing user supplied input.

LID-15	Submit Always Returns Zero
Asset	`v0.1.0-rc.1` : `contracts/v0.4.24/DePool.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`DePool._submit()` [19] never assigns to the `StETH` return variable, so the value returned is always 0.

This has no direct security impact on the Lido platform, but may cause problems for contracts that use Lido and expect a non-zero return value upon successful call to `submit()` .

Recommendations

Modify `Lido._submit()` such that it returns the amount of stETH minted by the submission, as documented.

Resolution

This has been resolved in PR #193 such that `Lido.submit()` now returns the number of shares generated.

[19] Defined at `v0.1.0-rc.1` `DePool.sol:408-431`

LID-16	Potential Gas Saving Optimisation in `Memutils.memcpy`
Asset	`v0.2.1-rc.0: contracts/v0.4.24/lib/Memutils.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `Memutils.memcpy()` function uses a gas-expensive `EXP` operation, which can be easily replaced by using the much cheaper `SHL`.

The current memcpy implementation has the following line [36]:

```
let mask := sub(exp(256, sub(32, _len)), 1) // 2 ** (8 * (32 - _len)) - 1
```

Which can be replaced as follows:

```
let mask := sub(shl(1, mul(8, sub(32, _len))), 1) // 2 ** (8 * (32 - _len)) - 1
```

`MUL` and `SHL` cost 5 and 3 gas respectively, where `EXP` is at least 10 gas.[20]

While `SHL` was introduced more recently, in the Constantinople hard fork, the current contracts are compiled for the Constantinople EVM. Because these operations are written in inline assembly, the Solidity compiler will not perform any optimizations.

Recommendations

Consider modifying `Memutils.sol:36`, as described above, including relevant tests to confirm equivalent functionality.

Resolution

This issue was tracked in Issue #242 and resolved in PR #239.

[20] See: https://eips.ethereum.org/EIPS/eip-145, https://eips.ethereum.org/EIPS/eip-160

LID-17	Miscellaneous Observations on Lido-DAO Codebase
Asset	`lidofinance/lido-dao`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team on the codebase associated with the Lido-DAO contracts, that do not have direct security implications:

1. The `stETH.burn()` [21] and `StETH.burnShares()` methods are unused. Although protected by `BURN_ROLE`, there is no reason for this method to exist until stETH withdrawal is possible.
2. We recommend avoiding use of the pre and post-increment operators (`++`) within other expressions, such that the pre vs post-increment result in different outcomes. Although they make the code concise, this can be more difficult to understand or easily missed by less familiar readers. e.g. `Lido.sol:518,776` `LidoOracle.sol:311`
3. Note that the `Lido.stop()` function is shadowed in inline assembly by the `stop()` instruction. This would, however, only be an issue if some inline assembly tried to call the function.
4. Gas savings in `NodeOperatorRegistry.assignNextSigningKeys()`: The loop at line [336] iterates through the entire cache when it could break when `numLoadedKeys == numAssignedKeys`. This is likely a very minimal improvement though, when taking into all the looping earlier in the function.
5. In the `getBit()` and `setBit()` functions defined in `BitOps.sol`, consider asserting (using a `require` statement) that the parameter `_bitIndex < 256`, or pass it as a `uint8`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments have been understood and acknowledged by the Lido team, with fixes implemented where deemed useful/relevant.

In particular, items 4 and 5 were fixed in PR #239.

The Lido team also made a note regarding item 1, clarifying that `StETH.burn()` will be used prior to withdrawal as part of the insurance fund implementation.

[21] removed since `v0.1.0-rc.1`

LID-18	Miscellaneous Observations on DC4BC Codebase
Asset	`dc4bc`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team on the dc4bc codebase that do not have direct security implications:

- In `airgapped/encryption.go`, key derivation is performed in every call to encrypt and decrypt, rather than just after reading the password from stdin (in `SetEncryptionKey` roughly every 10min).

  This offers no meaningful security improvements compared to saving the derived key in memory. By deriving the encryption key only once every 10 min, a much stronger parameterization is possible.
- `client/http_server.go:263` sets content type to `"image/jpeg"` but should be `"image/png"` [22]

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments have been understood and acknowledged by the Lido team, with some relevant fixes implemented in PR #74.

[22] See `https://godoc.org/github.com/skip2/go-qrcode#Encode`

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

σ'