Linea Yield Management Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Stalin

Assisting Auditors

ChainDefenders (1337web3, PeterSRWeb3)

February 12, 2026

Contents

1 About Cyfrin **3**

2 Disclaimer **3**

3 Risk Classification **3**

4 Protocol Summary **3**

5 Audit Scope **3**

6 Executive Summary **3**

7 Findings **15**

7.1 High Risk . . . 15

7.1.1 Execution of `lstLiabilities` repayment upon detection of positive yield and sufficient `stVault` liquidity disrupts the integrity of `userFunds` accounting leading to breaking core functionality to report positive yield and repayment of liabilities and obligations . . . 15

7.2 Medium Risk . . . 18

7.2.1 `YieldManager::withdrawLST` uses stale `lstLiabilityPrincipal` can cause temporary DoS when negative rebasing occurs . . . 18

7.2.2 Negative yield never accounted in `YieldManager::_getTotalSystemBalance` can result in temporary DoS . . . 18

7.2.3 `YieldManager::fundYieldProvider` and `LidoStVaultYieldProvider::fundYieldProvider` don't enforce `isStakingPaused` and `isOssificationInitiated` allowing unsafe staking . . . 19

7.2.4 Settlement of liabilities and obligations lacks optimization for priority repayment, leading to accumulation of unpaid negative yield in the system . . . 20

7.2.5 LST withdrawal and repayment inflates `userFundsInYieldProvidersTotal` causing subsequent DoS in core protocol functions . . . 21

7.2.6 `YieldManager::unpauseStaking` uses stale `lstLiabilityPrincipal` causing DoS when external actor repays LST liability . . . 22

7.2.7 External LST liability settlements are lost to the protocol when ossification and yield provider removal precedes yield reporting . . . 22

7.3 Low Risk . . . 25

7.3.1 Missing revert of LST withdrawal when `L1MessageService` balance is exactly equal to required value . . . 25

7.3.2 Incorrect yield accounting when `_payNodeOperatorFees` reverts in `LidoStVaultYieldProvider::reportYield` . . . 25

7.3.3 Risks in `YieldManager::replenishWithdrawalReserve` function . . . 26

7.3.4 New deposits into `YieldManager` don't pay down LST liability causing protocol to accrue greater interest liability . . . 26

7.3.5 Users can't withdraw funds when all permissionless functions for withdrawing are paused, contrary to the protocol specification . . . 27

7.3.6 Beacon chain deposits can still occur when withdrawal reserve is in deficit violating safety property . . . 27

7.4 Informational . . . 29

7.4.1 Remove todo comments . . . 29

7.4.2 Consistently use `ErrorUtils::revertIfZeroAddress` . . . 29

7.4.3 Redundant `FUNDER_ROLE` in `LineaRollupYieldExtension` . . . 29

7.4.4 Inconsistent storage location namespace root in `YieldManagerStorageLayout` . . . 30

7.4.5 Consider emitting event when synchronizing `lstLiabilityPrincipal` . . . 30

7.4.6 Refactor duplicated checks into modifiers . . . 30

7.4.7 Withdrawing LSTs can result in a system balance below the minimum reserve requirement . . . 31

7.4.8 User principal can be permissionlessly used to pay system obligations violating safety property 31

7.4.9 `YieldManager::fundYieldProvider` does not automatically unpause staking after funding . . . 32

7.5 Gas Optimization . . . 33

7.5.1 Fast fast by performing input-related checks first 33
7.5.2 Refactor `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` to eliminate `liabilityETH` 33

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Linea is a Type 2 zkEVM L2 rollup with full EVM equivalence which aims to provide the "Ethereum experience" at scale and has been previously audited by Cyfrin.

5 Audit Scope

This was an incremental audit focusing on the new "Yield Management" functionality and related changes to existing contracts, with the most complex part of the audit being integration with the new Lido V3 protocol. The audit scope was limited to:

```
contracts/contracts/messageService/l1/L1MessageService.sol
contracts/contracts/lib/YieldManagerPauseManager.sol
contracts/contracts/messageService/lib/MessageHashing.sol
contracts/contracts/yield/libs/ValidatorContainerProofVerifier.sol
contracts/contracts/yield/LidoStVaultYieldProvider.sol
contracts/contracts/yield/LidoStVaultYieldProviderFactory.sol
contracts/contracts/yield/YieldManager.sol
contracts/contracts/yield/YieldManagerStorageLayout.sol
contracts/contracts/yield/YieldProviderBase.sol
contracts/contracts/LineaRollup.sol
contracts/contracts/LineaRollupBase.sol
contracts/contracts/LineaRollupYieldExtension.sol
```

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the Linea Yield Management smart contracts provided by Linea. In this period, a total of 25 issues were found.

The findings consist of 1 High, 7 Medium and 6 Low severity issues with the remainder being informational and gas optimizations.

The most significant findings detected accounting corruptions with various impacts and were mostly related to LST withdrawals and liability settlement.

Important Protocol Invariants

The following important on-chain invariants were derived from the technical specifications document:

- users can always eventually withdraw their funds; no actor can permanently block or prevent users from withdrawing their assets
- permissionless withdrawal mechanisms become available during reserve deficits; reserve replenishment takes precedence over repaying system obligations
- All ETH transfer operations involving the `YieldManager` MUST ensure that the withdrawal reserve remains minimum threshold
- Whenever an ETH transfer operation involving the `YieldManager` restores the withdrawal reserve following a deficit, it should aim to replenish up to the target reserve rather than stopping at the minimum
- accumulated system obligations (lst liability, lido & node operator fees) must be settled using unreported yield before yield can be distributed to L2 recipients
- User principal (L1 deposits + L2 circulating ETH) MUST NOT be used for system obligations; all obligations MUST be settled exclusively from unreported yield which ensures that users funds are insulated from ongoing system obligations
- Synthetic `L1MessageService.MessageSent` events derived from native yield reporting MUST exclude all accumulated system obligations from the reported amount
- Beacon chain deposits MUST be paused when withdrawal reserve is in deficit, outstanding stETH liabilities exist or ossification has been initiated or completed
- If the system does not maintain positive net yield over time, the system will automatically pause new beacon chain deposits
- Withdrawals must never fail due to stale LST liability accounting
- When ossification has been initiated or completed, liabilities are incurred, or withdrawal deficits exist, new validator deposits must be paused to protect user funds

Flow Of Value

Value Input

ETH is the only form of value which can enter the protocol via:

1. `YieldManager::receiveFundsFromReserve` - `L1MessageService` (`LineaRollupYieldExtension`) ETH from withdrawal reserve to `YieldManager` for staking
2. `YieldManager::receive` - secondary catch-all for any ETH transfers to `YieldManager` address
   - Vault withdrawals: When `YieldManager::withdrawFromYieldProvider` calls `StakingVault::withdraw` (via `Dashboard`/`VaultHub`), ETH is sent via `.call{value}` to `YieldManager`
   - Direct donations: Anyone can send ETH directly to `YieldManager`
3. Yield Generation via LidoV3 / External Actor Windfall - new ETH doesn't appear in the yield provider's `totalValue` tracked by `VaultHub` until the Lido Oracle committee pushes a fresh report that includes higher combined vault and validator balances. This is subsequently detected by `YieldManager::reportYield` and reported to L2 via `LineaRollupYieldExtension::reportNativeYield`
4. External LST liability settlement

Value Output

ETH and stETH are the two forms of value which can exit the protocol via:

1. `YieldManager::transferFundsToReserve` - `YIELD_PROVIDER_UNSTAKER_ROLE` directly transfers ETH from `YieldManager` to `L1MessageService`
2. `YieldManager::addToWithdrawalReserve`, `safeAddToWithdrawalReserve` - `YIELD_PROVIDER_UNSTAKER_ROLE` withdraws from `YieldProvider` and sends to `L1MessageService`
3. `YieldManager::replenishWithdrawalReserve` - permissionless function to top up L1 ETH reserve when in deficit
4. `YieldManager::withdrawFromYieldProvider` - `YIELD_PROVIDER_UNSTAKER_ROLE` withdraws previously deposited ETH from `YieldProvider`, may route to reserve if target deficit exists
5. `YieldManager::fundYieldProvider` - `YIELD_PROVIDER_STAKING_ROLE` sends ETH from `YieldManager` to `YieldProvider`
6. `YieldManager::withdrawLST` - mints stETH to recipient (value exit in LST form, not ETH), creates LST liability (principal & interest) that must eventually be repaid
7. Beacon Chain Slashing Penalties - validators can be penalized losing ETH by the beacon chain slashing mechanisms which can create negative yield
8. Lido System Obligations - payment of Lido protocol fees & node operator fees can cause ETH to leave the system. Normally this is done using generated yield and only the "net" generated yield (remaining after obligations are paid) is reported to L2 for distribution, though it can also be permissionlessly triggered by an external actor

Important Protocol Accounting

Theoretical Concepts

One of the most complicated parts of the protocol is the internal accounting. Some useful theoretical definitions which helped find accounting-related issues:

```
1) Total Gross Assets =
     YieldManager.balance          // ETH in YieldManager
     + L1MessageService.balance    // ETH in L1MessageService
     + (Dashboard.totalValue())   // For each yield provider, factors in negative yield

2) Outstanding Fees & Liabilities =
     (for each provider:
         STETH.getPooledEthBySharesDashboard.liabilityShares())  // LST liability with rebasing
         + (, feesToSettle) = Dashboard.obligations()             // Unpaid Lido protocol fees
         + Dashboard.accruedFee()                                 // Unpaid node operator fees
      )

3) Total Net Assets = Total Gross Assets - Outstanding Fees & Liabilities
```

Accounting Implementation

As a result of our findings accounting-related implementation definitions were changed; this section reflects the most recent understanding:

1) `YieldProviderStorage`
   - `userFunds` - total ETH in a given yield provider owed to Linea bridge users
   - `lstLiabilityPrincipal` - ETH-denominated liability created when a user withdraws LST; must be reconciled against current ETH value of `VaultHub::liabilityShares` prior to use since external actors may permissionlessly settle liabilities
2) `YieldManagerStorage::userFundsInYieldProvidersTotal` - total ETH in all yield providers owed to Linea bridge users

Consider this scenario:

```
T=0
Action: 200 ETH deposited into a yield provider's StakingVault
```

```
State:
  StakingVault.balance = 200 ETH
  YieldProviderStorage::userFunds = 200 ETH
  YieldProviderStorage::lstLiabilityPrincipal = 0
  YieldManagerStorage::userFundsInYieldProvidersTotal = 200 ETH

At T=0:
* 200 ETH is in the vault
* 200 ETH is owed to users
* Vault Net Assets = 200 ETH

T=1
Action: LST withdrawal worth 100 ETH
State:
  StakingVault.balance = 200 ETH
  YieldProviderStorage::userFunds = 100 ETH
  YieldProviderStorage::lstLiabilityPrincipal = 100 ETH
  YieldManagerStorage::userFundsInYieldProvidersTotal = 100 ETH

AT T=1, 200 ETH is still in the vault, however liability has shifted:
* 100 ETH is owed to users
* 100 ETH is owed to Lido (can increase over time if not promptly repaid)
* Vault Net Assets = 200 - 100 = 100 ETH
```

Unknown Yield Recipients Risk

It is not clear which entities will receive the generated yield nor how the generated yield will be distributed amongst yield recipients, hence the following scenario could arise:

- users can bridge their ETH onto Linea L2
- their bridged ETH that previously sat in Linea's L1 contract can now be used to generate yield via Lido V3
- but users may not actually receive any of the generated yield, or they may not receive generated yield in proportion to the ETH they contributed

The effect is that users' bridged ETH can be exposed to greater risk but they may not receive any of the benefits. The new risks that users' ETH are exposed to if used for yield generation:

- smart contract risk from Linea's new yield manager contracts, Lido V3's contracts and the integration between them
- node operator validator slashing

Our understanding is that the yield distribution mechanism is a separate, subsequent development.

Post Audit Recommendations

The protocol is undergoing multiple simultaneous audits; depending on the total number of findings and whether significant refactoring is required, it may be prudent to conduct at least one final audit of the version including all mitigations if timeline allows.

LidoV3 is also very new which adds additional risk in regards to integrating with it; during our audit Lido was continually pushing fixes to their V3 codebase.

After deployment it would be prudent to run the protocol for some time with:

- only a small % of the bridged ETH used for yield generation
- extensive off-chain monitoring that continually tracks important protocol and vault accounting invariants, and alerts if it detects any accounting abnormalities

Ideally the protocol would run flawlessly with no accounting irregularities for some time before increasing the % of bridged ETH used for yield generation to more substantial amounts.

Self-Reported Findings Review

Linea self-reported the following findings and we reviewed the proposed mitigations:

- `_payMaximumPossibleLSTLiability` should execute before calling `dashboard.withdrawableValue` to avoid revert due to stale value - reviewed PR1853
- ossification fails to complete if the vault had no staged balance - reviewed PR1877
- `YieldManager::addYieldProvider` should initialize `userFunds` and increment `userFundsInYieldProvidersTotal` by the initial amount sent in `LidoStVaultYieldProvider::initializeVendorContracts` - reviewed PR1961

Other Audits Findings Review

Linea asked us to review the proposed mitigations for the following significant findings produced by other audits:

- Duplicate validator requests used to DoS permissionless unstaking - reviewed PR1879 and QA improvements in commits 3e86dad, 43df9a0, 2a5cd1c, 8b98bc0, 62818e8
- Zero amount provider withdrawal blocks partial reserve replenishment - reviewed PR1853
- Forwarded ETH inflates system balance and bypasses reserve deficit gate - reviewed PR1854
- Emergency removal does not decrement global user funds total - reviewed PR1876

Merged To Main

The new functionality containing mitigations from all audits was merged into the `main` branch at commit 9c16ce3.

Deployment Bytecode Verification

At the conclusion of the audit after all mitigations were successfully concluded, Cyfrin was asked to verify that at commit 25e323d the locally compiled bytecode matched the expected bytecode listed in that commit hash and deployed to mainnet. We used the following reproducible process to do this.

1) prepare the repo:

```
gh repo clone Consensys/linea-monorepo
cd linea-monorepo
git checkout 25e323d055dec40ef167a190c71c30aa9bf92c23
pnpm install --ignore-scripts // to get around an error related to better-sqlite3
cd contracts
npx hardhat test
```

2) diff the locally compiled bytecode against the expected:

```
linea-monorepo/contracts$ diff build/contracts/LineaRollup.sol/LineaRollup.json
↪  deployments/bytecode/2026-01-14/LineaRollup.json
linea-monorepo/contracts$ diff
↪  build/contracts/yield/LidoStVaultYieldProviderFactory.sol/LidoStVaultYieldProviderFactory.json
↪  deployments/bytecode/2026-01-14/LidoStVaultYieldProviderFactory.json
linea-monorepo/contracts$ diff
↪  build/contracts/yield/libs/ValidatorContainerProofVerifier.sol/ValidatorContainerProofVerifier.json
↪  deployments/bytecode/2026-01-14/ValidatorContainerProofVerifier.json
```

3) prepare the helper programs:

To extract deployment bytecode use this python program `extract_deployed_bytecode.py`:

```
import sys
import os
import requests

if len(sys.argv) != 4:
    print("Usage: python3 extract_deployed_bytecode.py <env_var_for_rpc_url> <address> <output_path>")
    sys.exit(1)

env_var = sys.argv[1]
```

```
address = sys.argv[2]
output_path = sys.argv[3]

rpc_url = os.environ.get(env_var)
if not rpc_url:
    print(f"Error: Environment variable '{env_var}' is not set.")
    sys.exit(1)

data = {
    "jsonrpc": "2.0",
    "method": "eth_getCode",
    "params": [address, "latest"],
    "id": 1
}

response = requests.post(rpc_url, json=data)
if response.status_code == 200:
    result = response.json()
    if "result" in result:
        bytecode = result["result"]
        with open(output_path, "w") as f:
            f.write(bytecode)
        print(f"Deployed bytecode saved to {output_path}")
    else:
        print("Error: No bytecode found in response")
else:
    print(f"Error: RPC request failed with status {response.status_code}")
```

To compare locally compiled bytecode against deployed bytecode ignoring differences in metadata & immutable positions, use this python program `compare_bytecodes.py`:

```
import sys
import json
import os

def strip_metadata(bytecode: bytes) -> bytes:
    if len(bytecode) < 2:
        return bytecode
    cbor_len = (bytecode[-2] << 8) | bytecode[-1]
    if cbor_len <= 0 or cbor_len + 2 > len(bytecode):
        return bytecode
    # Optional validation: Check if it looks like CBOR metadata (starts with 0xa1 or 0xa2)
    potential_cbor = bytecode[-cbor_len - 2 : -2]
    if potential_cbor[0] not in (0xa1, 0xa2):  # Common for Solidity metadata
        return bytecode
    return bytecode[:-cbor_len - 2]

if len(sys.argv) != 3:
    print("Usage: python3 compare_bytecodes.py <artifact_json_path> <deployed_bytecode_path>")
    sys.exit(1)

artifact_path = sys.argv[1]
deployed_path = sys.argv[2]

try:
    with open(artifact_path, "r") as f:
        data = json.load(f)

    # Get local deployed bytecode
    deployed_bytecode_section = data.get("deployedBytecode")
    if isinstance(deployed_bytecode_section, str):
        local_hex = deployed_bytecode_section
    elif isinstance(deployed_bytecode_section, dict):
```

```
    local_hex = deployed_bytecode_section.get("object")
else:
    print("Error: 'deployedBytecode' not found or invalid in artifact.")
    sys.exit(1)

if not local_hex:
    print("Error: No bytecode found in artifact.")
    sys.exit(1)

# Strip '0x' if present
if local_hex.startswith("0x"):
    local_hex = local_hex[2:]

local_bytes = bytes.fromhex(local_hex)

# Get deployed bytecode from file
with open(deployed_path, "r") as f:
    deployed_hex = f.read().strip()

if deployed_hex.startswith("0x"):
    deployed_hex = deployed_hex[2:]

deployed_bytes = bytes.fromhex(deployed_hex)

# Strip metadata from both
local_bytes = strip_metadata(local_bytes)
deployed_bytes = strip_metadata(deployed_bytes)

if len(local_bytes) != len(deployed_bytes):
    print(f"Error: Bytecode lengths differ after stripping metadata: local {len(local_bytes)} vs
    ↪  deployed {len(deployed_bytes)}")
    sys.exit(1)

# Compare, ignoring positions where local is 0x00 but deployed is not (assumed to be immutable
↪  placeholders)
mismatch = False
assumed_immutable_positions = []
for i in range(len(local_bytes)):
    if local_bytes[i] == 0 and deployed_bytes[i] != 0:
        assumed_immutable_positions.append(i)
        continue
    if local_bytes[i] != deployed_bytes[i]:
        print(f"Mismatch at position {i}: local 0x{local_bytes[i]:02x} vs deployed
        ↪  0x{deployed_bytes[i]:02x}")
        mismatch = True

if mismatch:
    print("Bytecodes do NOT match (differences outside assumed immutable positions).")
else:
    print("Bytecodes match, ignoring differences in assumed immutable positions.")

if assumed_immutable_positions:
    print("\nAssumed immutable positions where local=0x00 and deployed !=0x00:")
    # Group consecutive positions
    groups = []
    start = assumed_immutable_positions[0]
    prev = assumed_immutable_positions[0]
    for pos in assumed_immutable_positions[1:]:
        if pos == prev + 1:
            prev = pos
        else:
            groups.append((start, prev))
            start = pos
```

```
                prev = pos
        groups.append((start, prev))
        for s, e in groups:
            length = e - s + 1
            local_val = local_bytes[s:s+length].hex()
            deployed_val = deployed_bytes[s:s+length].hex()
            print(f"  Range {s}-{e} (length {length}): local {local_val} vs deployed {deployed_val}")

except FileNotFoundError as e:
    print(f"Error: File not found - {e}")
    sys.exit(1)
except json.JSONDecodeError:
    print(f"Error: Invalid JSON in artifact file '{artifact_path}'.")
    sys.exit(1)
except ValueError as e:
    print(f"Error: Invalid hex in bytecode - {e}")
    sys.exit(1)
except Exception as e:
    print(f"Unexpected error: {str(e)}")
    sys.exit(1)
```

4) verify `LineaRollup` deployed bytecode using:

- `python3 extract_deployed_bytecode.py ETH_RPC_URL 0x04728BF704a716C26F9EF4085013b760AC885631 deployments/bytecode/2026-01-14/LineaRollup.mainnet.txt` to extract mainnet deployed bytecode
- `python3 compare_bytecodes.py build/contracts/LineaRollup.sol/LineaRollup.json deployments/bytecode/2026-01-14/LineaRollup.mainnet.txt` to verify both match - succeeds

5) verify `YieldManager` deployed bytecode using:

- `python3 extract_deployed_bytecode.py ETH_RPC_URL 0x751236A1aFC11B7F1A7630fe87b0Bd96AC5203C4 deployments/bytecode/2026-01-14/YieldManager.mainnet.txt` to extract mainnet deployed bytecode
- `python3 compare_bytecodes.py build/contracts/yield/YieldManager.sol/YieldManager.json deployments/bytecode/2026-01-14/YieldManager.mainnet.txt` to verify both match - succeeds with these differences where local=0x00 and deployed!=0x00 but is a known immutable value:

```
Assumed immutable positions where local=0x00 and deployed !=0x00:
  Range 2484-2503 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 7281-7300 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 7345-7364 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 10366-10385 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 14184-14203 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 19146-19165 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 20429-20448 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
```

6) verify `LidoStVaultYieldProvider` deployed bytecode using:

- `python3 extract_deployed_bytecode.py ETH_RPC_URL 0x486D8cADc10489B30b64c890aEc747F1220eEEC3 deployments/bytecode/2026-01-14/LidoStVaultYieldProvider.mainnet.txt` to extract mainnet deployed bytecode
- `python3 compare_bytecodes.py build/contracts/yield/LidoStVaultYieldProvider.sol/LidoStVaultYieldProvid deployments/bytecode/2026-01-14/LidoStVaultYieldProvider.mainnet.txt` - succeeds with these differences where local=0x00 and deployed!=0x00 but is a known immutable value:

```
Assumed immutable positions where local=0x00 and deployed !=0x00:
  Range 398-417 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  02ca7772ff14a9f6c1a08af385aa96bb1b34175a (Staking Vault Factory)
  Range 478-497 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 529-548 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 580-599 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  2309c45d44105928b483f608dd6140fb65f3ebde (ValidatorContainerProofVerifier)
  Range 747-766 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (Linea RollUp)
  Range 1143-1162 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  ae7ab96520de3a18e5e111b5eaab095312d7fe84 (Lido and stETH token)
  Range 1247-1266 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 1638-1657 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 1987-2006 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 2389-2408 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 2510-2529 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 2635-2654 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 2856-2875 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 3096-3115 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 3233-3252 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 3417-3436 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 4194-4213 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 4379-4398 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 4483-4502 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  02ca7772ff14a9f6c1a08af385aa96bb1b34175a (Staking Vault Factory)
  Range 4717-4736 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 4992-5011 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 5445-5464 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 5651-5670 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 6158-6177 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  ae7ab96520de3a18e5e111b5eaab095312d7fe84 (Lido and stETH token)
  Range 6483-6502 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 6747-6766 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 7011-7030 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 7563-7582 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  ae7ab96520de3a18e5e111b5eaab095312d7fe84 (Lido and stETH token)
  Range 8150-8169 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  2309c45d44105928b483f608dd6140fb65f3ebde (ValidatorContainerProofVerifier)
  Range 8350-8369 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  2309c45d44105928b483f608dd6140fb65f3ebde (ValidatorContainerProofVerifier)
```

```
  Range 8996-9015 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  ae7ab96520de3a18e5e111b5eaab095312d7fe84 (Lido and stETH token)
```

7) verify `LidoStVaultYieldProviderFactory` deployed bytecode using:

- `python3 extract_deployed_bytecode.py ETH_RPC_URL 0xE4FC9F1A8cB97fEAd3C2b37c11AD5B1C2eF73959 deployments/bytecode/2026-01-14/LidoStVaultYieldProviderFactory.mainnet.txt` to extract mainnet deployed bytecode
- `python3 compare_bytecodes.py build/contracts/yield/LidoStVaultYieldProviderFactory.sol/LidoStVaultYiel deployments/bytecode/2026-01-14/LidoStVaultYieldProviderFactory.mainnet.txt -` succeeds with these differences where local=0x00 and deployed!=0x00 but is a known immutable value:

```
Assumed immutable positions where local=0x00 and deployed !=0x00:
  Range 143-162 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  02ca7772ff14a9f6c1a08af385aa96bb1b34175a (Staking Vault Factory)
  Range 223-242 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 262-281 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 301-320 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  2309c45d44105928b483f608dd6140fb65f3ebde (ValidatorContainerProofVerifier)
  Range 340-359 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 387-406 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  ae7ab96520de3a18e5e111b5eaab095312d7fe84 (Lido and stETH token)
  Range 424-443 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  d19d4b5d358258f05d7b411e21a1460d11b0876f (LineaRollUp)
  Range 457-476 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  eb63cabdd78537b9b72a2afb573f7caa91bd8d94 (TransparentUpgradeableProxy)
  Range 490-509 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  1d201be093d847f6446530efb0e8fb426d176709 (Vault Hub)
  Range 523-542 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  02ca7772ff14a9f6c1a08af385aa96bb1b34175a (Staking Vault Factory)
  Range 556-575 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  ae7ab96520de3a18e5e111b5eaab095312d7fe84 (Lido and stETH token)
  Range 589-608 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  2309c45d44105928b483f608dd6140fb65f3ebde (ValidatorContainerProofVerifier)
```

8) verify `ValidatorContainerProofVerifier` deployed bytecode using:

- `python3 extract_deployed_bytecode.py ETH_RPC_URL 0x2309C45d44105928b483f608dD6140Fb65f3EBde deployments/bytecode/2026-01-14/ValidatorContainerProofVerifier.mainnet.txt` to extract mainnet deployed bytecode
- `python3 compare_bytecodes.py build/contracts/yield/libs/ValidatorContainerProofVerifier.sol/ValidatorC deployments/bytecode/2026-01-14/ValidatorContainerProofVerifier.mainnet.txt -` succeeds with these differences where local=0x00 and deployed!=0x00 but is a known immutable value:

```
Assumed immutable positions where local=0x00 and deployed !=0x00:
  Range 427-428 (length 2): local 0000 vs deployed 0b03 (GI_STATE_ROOT)
  Range 1544-1545 (length 2): local 0000 vs deployed 0b03 (GI_STATE_ROOT)
  Range 1724-1725 (length 2): local 0000 vs deployed 0b03 (GI_STATE_ROOT)
```

Summary

Project Name	Linea Yield Management
Repository	linea-monorepo
Commit	4360e31fa8c8...
Fix Commit	9c16ce37d2dc...
Audit Timeline	Nov 3rd - Nov 14th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	7
Low Risk	6
Informational	9
Gas Optimizations	2
Total Issues	25

Summary of Findings

[H-1] Execution of `lstLiabilities` repayment upon detection of positive yield and sufficient `stVault` liquidity disrupts the integrity of `userFunds` accounting leading to breaking core functionality to report positive yield and repayment of liabilities and obligations	Resolved
[M-1] `YieldManager::withdrawLST` uses stale `lstLiabilityPrincipal` can cause temporary DoS when negative rebasing occurs	Resolved
[M-2] Negative yield never accounted in `YieldManager::_getTotalSystemBalance` can result in temporary DoS	Resolved
[M-3] `YieldManager::fundYieldProvider` and `LidoStVaultYieldProvider::fundYieldProvider` don't enforce `isStakingPaused` and `isOssificationInitiated` allowing unsafe staking	Resolved
[M-4] Settlement of liabilities and obligations lacks optimization for priority repayment, leading to accumulation of unpaid negative yield in the system	Resolved
[M-5] LST withdrawal and repayment inflates `userFundsInYieldProvidersTotal` causing subsequent DoS in core protocol functions	Resolved
[M-6] `YieldManager::unpauseStaking` uses stale `lstLiabilityPrincipal` causing DoS when external actor repays LST liability	Resolved
[M-7] External LST liability settlements are lost to the protocol when ossification and yield provider removal precedes yield reporting	Resolved
[L-1] Missing revert of LST withdrawal when `L1MessageService` balance is exactly equal to required value	Resolved

[L-2] Incorrect yield accounting when `_payNodeOperatorFees` reverts in `LidoStVaultYieldProvider::reportYield`	Resolved
[L-3] Risks in `YieldManager::replenishWithdrawalReserve` function	Acknowledged
[L-4] New deposits into `YieldManager` don't pay down LST liability causing protocol to accrue greater interest liability	Acknowledged
[L-5] Users can't withdraw funds when all permissionless functions for withdrawing are paused, contrary to the protocol specification	Acknowledged
[L-6] Beacon chain deposits can still occur when withdrawal reserve is in deficit violating safety property	Acknowledged
[I-1] Remove todo comments	Resolved
[I-2] Consistently use `ErrorUtils::revertIfZeroAddress`	Resolved
[I-3] Redundant `FUNDER_ROLE` in `LineaRollupYieldExtension`	Resolved
[I-4] Inconsistent storage location namespace root in `YieldManagerStorageLayout`	Resolved
[I-5] Consider emitting event when synchronizing `lstLiabilityPrincipal`	Resolved
[I-6] Refactor duplicated checks into modifiers	Resolved
[I-7] Withdrawing LSTs can result in a system balance below the minimum reserve requirement	Acknowledged
[I-8] User principal can be permissionlessly used to pay system obligations violating safety property	Acknowledged
[I-9] `YieldManager::fundYieldProvider` does not automatically unpause staking after funding	Acknowledged
[G-1] Fast fast by performing input-related checks first	Resolved
[G-2] Refactor `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` to eliminate `liabilityETH`	Resolved

7 Findings

7.1 High Risk

7.1.1 Execution of `lstLiabilities` repayment upon detection of positive yield and sufficient `stVault` liquidity disrupts the integrity of `userFunds` accounting leading to breaking core functionality to report positive yield and repayment of liabilities and obligations

Description: The system attempts to repay all liabilities and obligations when positive yield is detected and there is liquidity on the `stVault`, but repayment of `lstLiabilities` in the execution flow of the `YieldManager::reportYield` doesn't modify the `userFunds` to accurately reflect the funds owed to Linea users.

The problem arises given that `userFunds` represents "funds owed to Linea users", but users can get their funds back in the form of `stETH` (lstToken) when the `LineaRollup` has not enough native balance. As such, repaying `lstLiabilities` but not modifying `userFunds` causes it to be desynchronised from the real funds owed to Linea users.

```
- At T0: stVault is funded with 200 ETH, and 100% of it is staked on the BeaconChain
Accounting state:
  dashboard.totalValue() => 200 ETH
  stVault.availableBalance() => 0 ETH (All staked)
  userFunds => 200 ETH
  lstLiabilityPrincipal => 0 ETH
  no obligations / no fees

    ↓

- At T1: - LST is minted to process a user withdrawal for 150ETH
    ↓
Someone calls: LineaRollupYieldExtension::claimMessageWithProofAndWithdrawLST
    ↓
calls: YieldManager::withdrawLST
    ↓
Delegatecalls: LidoStVaultYieldProvider::withdrawLST
    ↓
calls: Dashboard::mintStETH - mints stETH to the withdrawer worth 150 ETH
    ↓
- Increments liabilities owed by the Vault on the VaultHub, and increments `lstLiabilityPrincipal` on
↪  the YieldProvider.
    ↓
Accounting state post-processing withdrawal and minting stETH:
  dashboard.totalValue() => 200 ETH
  stVault.availableBalance() => 0 ETH (All staked)
  userFunds => 200 ETH
  lstLiabilityPrincipal => 150 ETH
  no obligations / no fees

    ↓

- At T2: - Given the high amount of lstLiabilities, a withdrawal from the BeaconChain is requested to
↪  pay off the liabilities.
    ↓
Someone calls: YieldManager.unstakePermissionless(yieldProvider, payload)
    ↓
Delegatecalls: LidoStVaultYieldProvider.unstakePermissionless()
    ↓
Validates: _validateUnstakePermissionlessRequest() - checks beacon state via Merkle proof
    ↓
Submits: _unstake() → Dashboard.requestWithdrawals() → EIP-7002 precompile
    ↓
Beacon chain: Queues validator exit
    ↓
 2-7 days waiting period
```

```
    ↓
Beacon chain: Automatically sends ETH to stVault
    ↓
Accounting state after the cooldown period to withdraw from the BeaconChain, and the stVault has
↪  received all the staked ETH (200 ETH)
  dashboard.totalValue() => 200 ETH
  stVault.availableBalance() => 200 ETH
  userFunds => 200 ETH
  lstLiabilityPrincipal => 150 ETH
  no obligations / no fees

    ↓

- At T3 - Yield is reported (Note: No yield/interests have been applied on the previous steps, but here
↪  we assume there is at least one wei of yield, so execution follows the path of positive yield.
    ↓
Call to: YieldManager::reportYield
    ↓
Delegatecalls: LidoStVaultYieldProvider::reportYield
    ↓
-> Positive Yield is detected, internal call to
↪  LidoStVaultYieldProvider._payMaximumPossibleLSTLiability()
  -> StakingVault has 200 ETH on its balance and has liabilities worth 150 ETH, therefore:
  ->    ↓
  Call to: Dashboard::rebalanceVaultWithShares - Will rebalance shares worth 150 ETH
    -> Call to VAULT_HUB.rebalance() to rebalance 150 ETH worth of shares
      -> checks, update accounting to reflect the repaid 150 ETH worth of shares, and external call to
      ↪  `stVault::withdraw`
      ->   ↓
        -> Call to: `stVault::withdraw` for 150 ETH
          -> 150 ETH are withdrawn from the Vault into the VaultHub
      -> execution continues on VaultHub.rebalance() where it left before the external call to withdraw
      ↪  from the stVault, and next uses the withdrawn ETH to deposit on Lido (CorePool/LidoV2) and
      ↪  rebalance accounting
  -> Execution comes back to LidoStVaultYieldProvider::reportYield where it left before calling
  ↪  Dashboard::rebalanceVaultWithShares, and continues, given that in the example there are no
  ↪  obligations/fees, execution inside YieldProvider ends and comes back to YieldManager::reportYield.
  ↪  - positive yield is 0 because the repaid lstLiabilities are bigger than the generated yield.
- On the YieldManager, given that no positive yield was generated, no changes to `userFunds` or
↪  `yieldReportedCumulative` or `userFundsInYieldProvidersTotal` occur.
    ↓
Accounting state after reporting yield - 150 ETH of lstLiabilitity were repaid
  dashboard.totalValue() => 50 ETH
  stVault.availableBalance() => 50 ETH
  userFunds => 200 ETH
  lstLiabilityPrincipal => 0 ETH
  no obligations / no fees

    ↓

- At T4 - A withdrawal of 50 ETH from the Vault is requested.
    ↓
Call to: YieldManager::withdrawFromYieldProvider
    ↓
Delegatecalls: LidoStVaultYieldProvider::withdrawFromYieldProvider
    ↓
Call: Dashboard.withdraw() - withdraws the requested 50 ETH from the Vault
    ↓
Execution comes back to YieldManager::withdrawFromYieldProvider
- reduces `userFunds` and `userFundsInYieldProvidersTotal` by the withdrawn 50 ETH
    ↓
Accounting state after processing the withdrawal
  dashboard.totalValue() => 0 ETH
```

```
  stVault.availableBalance() => 0 ETH
  userFunds => 150 ETH
  lstLiabilityPrincipal => 0 ETH
  no obligations / no fees

The end result is that `userFunds` is left as 150 ETH while the vault is empty. This would apparently
↪  signal a shortfall of 150 ETH to the Linea users, when in reality, the Linea users have already
↪  withdrawn 150 worth of ETH in the form of stETH (at T1), and the other 50 ETH were sent to the
↪  LineaMessage to process further withdrawals (At T4).
```

Impact: Integrity of `userFunds` accounting variables is disrupted with the primary impact being disruption to the yield reporting mechanism. This stems from non-detection of positive yield due to inflation of `userFunds` relative to the actual `totalValue` locked in the Vault. A secondary consequence is payment of system obligations from that positive yield no longer occurs as the protocol thinks there is no positive yield.

Recommended Mitigation: Refactor the system accounting layer to unify withdrawal handling, treating stETH-denominated withdrawals equivalently to native ETH withdrawals. Both `userFunds` and `userFundsInYieldProvidersTotal` must be proactively decremented upon lst issuance during withdrawal finalization.

Linea: Fixed as of commit 40b3328.

Cyfrin: Verified. Semantics for `userFunds` and `lstLiabilities` have been redefined. Code has been refactored to properly track `userFunds` as funds owed to Linea users.

- Now, each time a withdrawal request mints stETH to process the withdrawal, the ETH value of the minted stETH is discounted from the `userFunds` to correctly reflect those funds are no longer owed to Linea Users.
- Additionally, the YieldManager aims to repay as most `lstLiabilities` (principal and interest) as possible, no longer limited to only repay `lstPrincipal`.

Repayment of `lstLiabilities` doesn't require to discount the repaid ETH from `userFunds` as that value is correctly discounted at the moment when the lst was originally minted to process a user withdrawal request.

7.2 Medium Risk

7.2.1 `YieldManager::withdrawLST` uses stale `lstLiabilityPrincipal` can cause temporary DoS when negative rebasing occurs

Description: Lido's `stETH` is a rebasing token which can experience both positive and negative rebasing (eg due to slashing).

`LidoStVaultYieldProvider::withdrawLST`:

- takes `_amount` as input which is token amount
- calls `Dashboard::mintStETH` from LidoV3 using token `_amount`
- this in turn calls `STETH::getSharesByPooledEth` to get the shares and the shares are what is minted
- then `$$.lstLiabilityPrincipal += _amount;` records to storage the input token `_amount` as a liability, but this amount can change (both increase & decrease) due to subsequent `stETH` rebasing

Before using `$$.lstLiabilityPrincipal` other functions from `LineaStVaultYieldProvider` synchronize it by calling `_syncExternalLiabilitySettlement`.

However synchronization doesn't occur in the "entry" function `YieldManager::withdrawLST`:

```
YieldProviderStorage storage $$ = _getYieldProviderStorage(_yieldProvider);
// @audit `$$.lstLiabilityPrincipal` used without being sync'd
if ($$.lstLiabilityPrincipal + _amount > $$.userFunds) {
  revert LSTWithdrawalExceedsYieldProviderFunds();
}
```

At this point `$$.lstLiabilityPrincipal` may no longer be accurate depending on whether positive or negative rebasing occurred.

Impact: In the positive rebasing scenario, Lido's system wouldn't allow overminting `stETH` - it's capped by the deposited ETH on the vault and the collateralization parameter. Hence there is no impact in this case. But in the negative rebasing case a temporary DoS can occur:

```
State: lstLiabilityPrincipal = 100 ETH (stale)
Reality: Dashboard.liabilityShares worth 90 ETH (-10% slashing)
Request: withdrawLST(50 ETH) with userFunds = 140 ETH

Stale Check: 100 + 50 = 150 > 140  REVERTS
Real Check: 90 + 50 = 140  140  Should pass

Result: LST withdrawal blocked during reserve deficit
```

The temporary DoS would resolve itself the next time an operation occurred that triggered the sync.

Recommended Mitigation: Sync `$$.lstLiabilityPrincipal` prior to usage in `YieldManager::withdrawLST` similar to what happens everywhere else it is used.

Linea: Fixed in commits 2b99f9bf && 5cbe6b5, which belongs to PR 1703.

Cyfrin: Verified. The `userFunds` semantics change PR, removes the need to sync `$$.lstLiabilityPrincipal` in `YieldManager::withdrawLST`. It is no longer used for a check.

7.2.2 Negative yield never accounted in `YieldManager::_getTotalSystemBalance` can result in temporary DoS

Description: `YieldManager::reportYield` receives `outstandingNegativeYield` from providers but only emits an event without decrementing `$$.userFunds`. While this is intentional (`$$.userFunds` is a liability ledger, not an asset tracker), the negative yield is not accounted for anywhere causing `_getTotalSystemBalance` to return inflated values:

```
function _getTotalSystemBalance() internal view returns (...) {
    // @audit not adjusted for negative yield
    totalSystemBalance = L1_MESSAGE_SERVICE.balance +
                         address(this).balance +
                         $.userFundsInYieldProvidersTotal;
}
```

This inflated balance is then used to calculate reserve thresholds, producing incorrect operational decisions.

Impact: After slashing events, inflated `_getTotalSystemBalance` causes incorrect reserve threshold calculations, leading to:

Temporary DoS (incorrectly blocks operations):

- `receiveFundsFromReserve` - can't receive funds when reserve is actually sufficient
- `fundYieldProvider` - can't stake funds
- `unpauseStaking` - can't resume staking

Incorrect Execution (incorrectly allows operations):

- `unstakePermissionless` - allows unstaking when reserve is actually adequate
- `replenishWithdrawalReserve` - attempts replenishment when unnecessary

Other Effects:

- `getTargetReserveDeficit` returns inflated deficit, affecting downstream calculations
- `withdrawLST` check could be more conservative (though Lido prevents actual over-withdrawal)

Example: After 100 ETH slashing with 10% reserve requirement:

- Calculated minimum: 100 ETH (based on inflated 1000 ETH)
- Actual minimum: 90 ETH (based on real 900 ETH)
- If reserve has 95 ETH: operations blocked until yield recovers

All impacts are temporary until positive yield normalizes accounting.

Recommended Mitigation: Store the individual `_yieldProvider` `outstandingNegativeYield` in `YieldProvider-Storage` then use that in:

- `LSTWithdrawalExceedsYieldProviderFunds` check in `YieldManager::withdrawLST`
- `YieldManager::withdrawableValue` to return a more accurate figure

What is more difficult is that `YieldManager::_getTotalSystemBalance` should ideally account for the total outstanding negative yield of all yield providers in order to return an accurate value. However this can't be efficiently done on-chain and is likely safe to not implement.

Linea: Fixed in commit 4fd227

Cyfrin: Verified. Outstanding Negative Yield is stored for each Yield Provider and used to tighten the minting of `stETH` to process user withdrawal requests. Negative Yield is not accounted on the total system balance to prevent DoS because of using outdated data as is possible to repay liabilities and obligations asynchronously.

7.2.3 `YieldManager::fundYieldProvider` and `LidoStVaultYieldProvider::fundYieldProvider` don't enforce `isStakingPaused` and `isOssificationInitiated` allowing unsafe staking

Description: `YieldManager::fundYieldProvider` and `LidoStVaultYieldProvider::fundYieldProvider` don't check the `isStakingPaused` and `isOssificationInitiated` flags, allowing staking of new funds when this should be blocked. The protocol specification states: *"When ossification has been initiated or completed, liabilities are incurred, or withdrawal deficits exist, new validator deposits must be paused to protect user funds."*

Yield Providers can be paused by either:

- explicit call to `YieldManager::pauseStaking`
- when LST liabilities are incurred via `YieldManager::withdrawLST` or ossification is initiated via `YieldManager::initiateOssification`, both functions call `_pauseStakingIfNotAlready` to set `isStakingPaused = true` for the given `_yieldProvider`

`LidoStVaultYieldProvider::fundYieldProvider` reverts if the `isOssified` flag has been set to `true`, but neither it nor `YieldManager::fundYieldProvider` ever revert if the `isStakingPaused` flag for that yield provider has been set to `true`.

Impact: Unsafe staking operations; new funds can be staked even if staking has explicitly been paused for a given yield provider.

Recommended Mitigation: Add `isStakingPaused` and `isOssificationInitiated` checks to `LidoStVaultYieldProvider::fundYieldProvider` which already has the ossification check.

Linea: Fixed in commit becfd756.

Cyfrin: Verified. `LidoStVaultYieldProvider::fundYieldProvider` now reverts if staking has been paused or if ossification has been initiated or completed.

7.2.4 Settlement of liabilities and obligations lacks optimization for priority repayment, leading to accumulation of unpaid negative yield in the system

Description: Interest accrued on `lstLiabilities` and operational costs (obligations) is designated for priority settlement using the yield generated from staked ETH. Only the residual yield—after full settlement of all debts (encompassing both obligations and liabilities)—is to be reported as positive yield to Linea (L2).

The payment of liabilities and obligations is architected to execute concurrently with yield reporting, triggered exclusively upon detection of a positive yield delta.

The incorrect misprioritization in settlement is derived from a dependency on sufficient vault balance: transactions are gated by the `stVault's` available liquidity. If the vault lacks funds, liability settlement is deferred, resulting in unpaid negative yield accrual.

Consider a scenario wherein 100% of `ETH` within an `stVault` is allocated to staking:

```
- At T0:
dashboard.totalValue = 100 ETH
userFunds = 100 ETH
stVault.availableBalance = 0 ETH
no obligations nor liabilities

- At T1 - 1 ETH worth of yield has been generated, and 0.1 ETH of obligations has been incurred
dashboard.totalValue = 101 ETH
userFunds = 100 ETH
stVault.availableBalance = 0 ETH
obligations (fees/liabilities) = 0.1 ETH
- Here, 1 ETH worth of positive yield is detected, but no payment is possible because the stVault has no
↪  balance (therefore, no withdrawal is possible)
- So, that 1 ETH is reported as a positive yield to the L2, leaving 0.1 ETH as obligations.

- At T2 - A partial withdrawal for 1 ETH from the BeaconChain is completed, and another 0.1 ETH on
↪  obligations and yield is accounted
dashboard.totalValue = 101.1 ETH
userFunds = 101 ETH
stVault.availableBalance = 1 ETH
obligations (fees/liabilities) = 0.2 ETH

- At T3 - Yield is reported, 1 more ETH has been earned as yield and obligations
dashboard.totalValue = 102.1 ETH
userFunds = 101 ETH
stVault.availableBalance = 1 ETH
obligations (fees/liabilities) = 1.1 ETH
```

```
This time 1 ETH of obligations is paid off, but 0.1 ETH is still pending.

The issue is that at T1, 1ETH was reported as yielding a positive return, even though there were
↪  actually outstanding obligations that needed to be paid. Therefore, payment of obligations was not
↪  prioritized correctly; instead, yield distribution took precedence.
```

Impact: Negative yield accrues within the system, resulting in the following effects:

1. **Diminished Effective Yield on Staked ETH**: The real yield derived from staked ETH is reduced when stETH rebases at a percentage exceeding the accrued staking interest. This discrepancy arises because the delayed settlement of liabilities allows interest on the `lstLiabilities` to compound over time, thereby amplifying the shortfall.
2. **Exacerbation of Other Issues**: The accumulation of negative yield proceeds without appropriate tracking mechanisms, thereby intensifying the deficiencies outlined in a related issue.

Recommended Mitigation: The idea would be to account for the obligations and liabilities at the moment of determining if there is a positive yield, as opposed to optimistically assuming the vault has enough balance to pay them. In this way, the "positive yield" would be reserved as payment of liabilities/obligations for when the vault has balance (i.e. a withdrawal of the generated yield from the beacon chain is completed).

Bottom line is, do not report positive yield to the L2 when there are pending payments, the yield should be reserved to give priority to repayment of debts, and only until all debt is settled, only then the generated yield should be reported to the L2.

In code, the fix would looks something along the lines of:

- Where `ALL_OBLIGATIONS` includes liabilities, obligations and node fee

```
  function reportYield(
    address _yieldProvider
  ) external onlyDelegateCall returns (uint256 newReportedYield, uint256 outstandingNegativeYield) {
    ...
    uint256 lastUserFunds = $$.userFunds;
    uint256 totalVaultFunds = _getDashboard($$).totalValue();
    // Gross positive yield
-    if (totalVaultFunds > lastUserFunds) {
+    if (totalVaultFunds > lastUserFunds + ALL_OBLIGATIONS ) {
      ...
      // Gross negative yield
    } else {
      newReportedYield = 0;
      outstandingNegativeYield = lastUserFunds - totalVaultFunds;
    }
  }
```

Linea: Fixed in commit 0e46ee.

Cyfrin: Verified. Positive yield is now reported only when the total value held by the underlying `stVault` exceeds all liabilities, obligations, and fees. When no positive yield is generated, `outstandingNegativeYield` is accumulated on the system. Payment of liabilities, obligations and fees is attempted each time `reportYield` is executed.

7.2.5 LST withdrawal and repayment inflates `userFundsInYieldProvidersTotal` causing subsequent DoS in core protocol functions

Description: `YieldManager::userFundsInYieldProvidersTotal` stores the total amount of user funds deposited into yield providers, which is a liability that eventually needs to be repaid to users when they withdraw. When:

- LSTs are withdrawn via `YieldProvider::withdrawLST`, the liability that needs to be repaid to users decreases however this is never accounted for by for example decrementing `userFundsInYieldProvidersTotal` and `userFunds` (see related issue)
- Yield is reported via `YieldManager::reportYield` which ends up calling `LidoStVaultYieldProvider::_payMaximumPossibleLSTLiability` to pay LST liabilities, `userFundsInYieldProvidersTotal` is never decremented even though ETH in the `stVault` is used to pay down LST liability

Impact: `YieldManager::userFundsInYieldProvidersTotal` will become inflated over time as LSTs are withdrawn and LST liabilities are repaid. Since:

- `YieldManager::_getTotalSystemBalance` uses `userFundsInYieldProvidersTotal` to calculate total system balance, hence it will return an inflated value
- `YieldManager::isWithdrawalReserveBelowMinimum` ends up calling `_getTotalSystemBalance` but since its return value will be inflated, the minimum reserve amount required will be calculated to be higher than it actually should be, as long as the % based calculation in `_getEffectiveMinimumWithdrawalReserve` returns a higher (inflated) value than the hard-minimum

Hence `isWithdrawalReserveBelowMinimum` can return `true` when it should actually return `false`, since it is based on an inflated `userFundsInYieldProvidersTotal`. This means that protocol functions which call `isWithdrawalReserveBelowMinimum` will revert when they shouldn't, causing a DoS.

Recommended Mitigation: Withdrawing LST and repaying LST liabilities must maintain the correct `userFundsInYieldProvidersTotal`. The simplest way to achieve this is to decrement `userFundsInYieldProvidersTotal` in `YieldManager::withdrawLST`.

Linea: Fixed as of commit 40b3328.

Cyfrin: Verified. `YieldManager::withdrawLST` now deducts from `userFundsInYieldProvidersTotal` to correctly reflect those funds are no longer owed to Linea users, while also incrementing `lstLiabilityPrincipal` to indicate a liability has been created to Lido.

7.2.6 `YieldManager::unpauseStaking` uses stale `lstLiabilityPrincipal` causing DoS when external actor repays LST liability

Description: `YieldManager::unpauseStaking` contains this check:

```
if ($$.lstLiabilityPrincipal > 0) {
  revert UnpauseStakingForbiddenWithCurrentLSTLiability();
}
```

However it doesn't first sync `$$.lstLiabilityPrincipal` using `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` like other places in the code, meaning it uses a stale value.

Impact: LST liabilities can be settled by external parties; when an external party settles a vault's LST liabilities, `YieldManager::unpauseStaking` will incorrectly revert resulting in a denial of service since it erroneously believes that an LST liability still exists.

Recommended Mitigation: `YieldManager::unpauseStaking` should sync `$$.lstLiabilityPrincipal` via `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` prior to using it.

Linea: Fixed in commit 0c17a98.

Cyfrin: Verified.

7.2.7 External LST liability settlements are lost to the protocol when ossification and yield provider removal precedes yield reporting

Description: When an external actor settles LST liabilities this creates a windfall (vault has more ETH than `userFunds` reflects) which the protocol's accounting does not immediately reflect. Next time `YieldManager::reportYield` is called the windfall from the external LST liability settlement is recognized by the protocol and reported as yield to be distributed on L2.

However this external windfall is lost to the protocol if `YieldManager::reportYield` is not called and:

1) ossification is initiated and completed
2) `YieldManager::withdrawFromYieldProvider` is called followed by `YieldManager::removeYieldProvider`:
   - `withdrawFromYieldProvider` correctly syncs `lstLiabilityPrincipal` to detect the external settlement but does NOT capture the windfall in `userFunds`
   - `YieldManager::removeYieldProvider` only checks that `userFunds == 0` before allowing removal and transferring vault ownership. However, it does NOT verify that the vault's actual balance matches `userFunds`

Impact: Permanent loss to the protocol of "windfall" protocol assets that could have been distributed to L2 users as:

- the protocol's internal accounting (`userFunds` and `userFundsInYieldProvidersTotal`) becomes permanently disconnected from physical reality (`vaultBalance`), with no mechanism to reconcile the discrepancy
- vault ownership is transferred to a new owner

Recommended Mitigation: 1) `YieldManager::removeYieldProvider` should additionally check the vault's actual value and revert if it is not zero:

```
    uint256 actualVaultValue;
    if ($$.isOssified) {
        actualVaultValue = IStakingVault($$.ossifiedEntrypoint).availableBalance();
    } else {
        actualVaultValue = IDashboard($$.primaryEntrypoint).totalValue();
    }

    if (actualVaultValue > 0) {
        revert VaultHasUnreportedValue(actualVaultValue);
    }
```

2) Currently `YieldManager::initiateOssification` calls `LidoStVaultYieldProvider::initiateOssification` which calls `_payMaximumPossibleLSTLiability`. When an external actor has settled the LST liability, `dashboard.liabilityShares()` will return 0 so the `if` branch will never be entered:

```
function _payMaximumPossibleLSTLiability(
    YieldProviderStorage storage $$
) internal returns (uint256 liabilityPaidETH) {
    if ($$.isOssified) return 0;
    IDashboard dashboard = IDashboard($$.primaryEntrypoint);
    address vault = $$.ossifiedEntrypoint;
    uint256 rebalanceShares = Math256.min(
        dashboard.liabilityShares(), // @audit returns 0 when LST liability externally settled
        STETH.getSharesByPooledEth(IStakingVault(vault).availableBalance())
    );
    if (rebalanceShares > 0) {
        // @audit code never executes when LST liability externally settled
        //
        // Cheaper lookup for before-after compare than availableBalance()
        uint256 vaultBalanceBeforeRebalance = vault.balance;
        dashboard.rebalanceVaultWithShares(rebalanceShares);
        // Apply consistent accounting treatment that LST interest paid first, then LST principal
        _syncExternalLiabilitySettlement($$, dashboard.liabilityShares(), $$.lstLiabilityPrincipal);
        liabilityPaidETH = vaultBalanceBeforeRebalance - vault.balance;
    }
}
```

One potential fix is to always sync LST liabilities in `LidoStVaultYieldProvider::_payMaximumPossibleLSTLiability`. The current strategy of not syncing if `dashboard.liabilityShares() == 0` appears incorrect as it doesn't handle the case when LST liabilities are externally settled.

Linea: Fixed in commit f45bdfc by having `_payMaximumPossibleLSTLiability` always call `_syncExternalLiabilitySettlement`.

Regarding the suggestion to have `YieldManager::removeYieldProvider` revert if the actual balance is not zero, this creates a potential DoS vector by sending ETH to the vault so did not implement this.

Additionally added commit 55fe25a to make `LidoStVaultYieldProvider:exitVendorContracts` revert if there is no vendor exit data.

Cyfrin: Verified.

7.3 Low Risk

7.3.1 Missing revert of LST withdrawal when `L1MessageService` balance is exactly equal to required value

Description: The `LineaRollupYieldExtension::claimMessageWithProofAndWithdrawLST` function is designed to withdraw LST tokens from a yield provider only when the `L1MessageService` balance is insufficient to fulfil message delivery. According to the function's documentation, it should "revert if the `L1MessageService` has sufficient balance to fulfill the message delivery." However, the balance check uses a strict less-than operator (<) instead of less-than-or-equal-to (<=):

```
if (_params.value < address(this).balance) {
  revert LSTWithdrawalRequiresDeficit();
}
```

This means when `_params.value` is exactly equal to `address(this).balance`, the condition evaluates to false and the function proceeds with LST withdrawal, even though the contract has sufficient balance to fulfil the claim without withdrawing from the yield provider.

Impact: When the contract balance exactly matches the claim value, the function will incorrectly proceed with withdrawing LST from the yield provider instead of reverting. This results in:

1. Unnecessary LST withdrawal when funds are already available
2. Gas waste for the caller
3. Violation of the stated invariant that LST withdrawal should only occur when there is a balance deficit
4. Potential operational inefficiencies in the yield management system

Recommended Mitigation: Change the comparison operator from < to <= to ensure the function reverts when the balance is sufficient (including when it's exactly equal):

```
if (_params.value <= address(this).balance) {
  revert LSTWithdrawalRequiresDeficit();
}
```

This ensures that LST withdrawal only occurs when there is an actual deficit (`_params.value > address(this).balance`).

Linea: Fixed in commit 9722a2a.

Cyfrin: Verified.

7.3.2 Incorrect yield accounting when `_payNodeOperatorFees` reverts in `LidoStVaultYieldProvider::reportYield`

Description: When `LidoStVaultYieldProvider::reportYield` invokes `_payNodeOperatorFees`, a revert in `_payNodeOperatorFees` causes the `nodeOperatorFees` value to effectively be treated as `0`.

Despite this, the parent function `YieldManager::reportYield` continues processing the yield without adjusting for the failed fee payment. As a result, the yield is misreported, and the `userFunds` value becomes **artificially inflated**, since the operator fees are never deducted.

Impact:

If `_payNodeOperatorFees` reverts, `$$.userFunds` will be overstated, leading to inaccurate accounting of user balances and potential overestimation of yield. Furthermore, this behavior does **not** trigger a revert in situations where insufficient funds exist to pay the operator, allowing inconsistent state updates to persist.

Recommended Mitigation: Persist the intended `nodeOperatorFees` value even if `_payNodeOperatorFees` fails, and ensure it is accounted for in subsequent operations (e.g., withdrawals or yield adjustments).

Linea: Fixed in commit 0e46ee which belongs to PR 1703.

Cyfrin: Verified. Positive yield is now reported only when the total value held by the underlying `stVault` exceeds all liabilities, obligations, and fees. Payment of liabilities, obligations and fees is attempted each time `reportYield` is executed.

7.3.3 Risks in `YieldManager::replenishWithdrawalReserve` function

Description: The function `replenishWithdrawalReserve` is designed to top up the withdrawal reserve to a target threshold using available liquidity. Key characteristics include:

- **Permissionless Access:** Any caller can invoke this function.
- **Yield Provider Selection:** The caller specifies which yield provider to use.
- **Reserve Logic:**
  1. Checks if the withdrawal reserve is below the effective minimum threshold.
  2. Attempts to fund the deficit using the YieldManager's own balance.
  3. If the balance is insufficient, withdraws from the specified yield provider.
  4. If the deficit still exists, staking is paused to prevent further withdrawals.

Potential Risk: Since the function allows arbitrary selection of `_yieldProvider`, a malicious or careless caller could select a yield provider with higher liabilities or lower liquidity. This can result in:

- Inefficient fund usage.
- Failed replenishments if the selected provider cannot fulfill the withdrawal.
- Unintended pauses of staking due to insufficient replenishment.

Impact:

1. **Liquidity Risk:** Using a suboptimal yield provider could deplete critical liquidity or fail to meet reserve thresholds.
2. **Operational Risk:** Unnecessary staking pauses.

Recommended Mitigation: To reduce risk, consider the following:

1. **Restrict Access:**
   - Make the function callable only by a trusted role (e.g., `ADMIN` or `TREASURY_MANAGER`).
2. **Optimize Provider Selection:**
   - Internally select the yield provider with the most favorable liquidity-to-liability ratio rather than relying on user input.
   - Alternatively, implement a scoring mechanism to choose the most optimal provider automatically.

Linea: Acknowledged; the business requirement is for this function to be permissionless so that if the authorized roles fail to act, anyone can step in to replenish the `L1MessageService` balance.

7.3.4 New deposits into `YieldManager` don't pay down LST liability causing protocol to accrue greater interest liability

Description: The technical specification says:

> `liabilityPrincipal` is effectively an advance to the user on the staked reserve funds. Hence it is paid down from:
>
> - New deposits into the YieldManager
> - Withdrawals from a Yield Provider
> - Earned yield

While the last 2 have been implemented, the first has not as can be seen by the implementation of `YieldManager::receiveFundsFromReserve` & `receive`:

```
function receiveFundsFromReserve() external payable {
  if (msg.sender != L1_MESSAGE_SERVICE) {
    revert SenderNotL1MessageService();
  }
  if (isWithdrawalReserveBelowMinimum()) {
    revert InsufficientWithdrawalReserve();
  }
  emit ReserveFundsReceived(msg.value);
}

receive() external payable {}
```

Impact: LST liability is not paid down using new deposits causing protocol to accrue greater interest liability.

Recommended Mitigation: The difficulty is that LST liability is per yield provider, while ETH deposits have no yield provider context. To implement this on-chain while preserving the existing interfaces would require a list or set to track yield providers with LST liability, then `receiveFundsFromReserve` could iterate through this list and pay off as much LST liability as possible.

Alternatively the specification could be amended to remove this requirement.

Linea: Acknowledged; LST liability is paid down when a yield provider which has a positive liability is funded, but not when the `YieldManager` itself is funded. While this is slightly less optimal it significantly simplifies the code; we will update the protocol specification to note this.

7.3.5 Users can't withdraw funds when all permissionless functions for withdrawing are paused, contrary to the protocol specification

Description: The protocol's technical specification states:

> users can always eventually withdraw their funds; **no actor can permanently block or prevent users from withdrawing their assets**

However all permissionless methods of withdrawing funds can be paused by privileged actors as `YieldManager::withdrawLST`, `replenishWithdrawalReserve`, `unstakePermissionless` all have modifier `whenTypeAndGeneralNotPaused(PauseType.NATIVE_YIELD_PERMISSIONLESS_ACTIONS)`.

Impact: Privileged actors can permanently block users from withdrawing their assets by pausing `PauseType.NATIVE_YIELD_PERMISSIONLESS_ACTIONS` then never unpausing, violating the protocol's specification which states: **no actor can permanently block or prevent users from withdrawing their assets**.

Recommended Mitigation: To comply with the protocol specification, remove the ability to pause permissionless actions. Otherwise rephrase the relevant section of the protocol specification.

Linea: Acknowledged; our rationale is that unbounded pauses would still exist for permissionless withdrawals on the `L1MessageService::claimMessageWithProof` function.

7.3.6 Beacon chain deposits can still occur when withdrawal reserve is in deficit violating safety property

Description: The protocol's technical specification states:

> Beacon Chain Deposit Restriction: Beacon chain deposits MUST be paused when:
>
> - Withdrawal reserve is in deficit, or
> - Outstanding stETH liabilities exist, or
> - Ossification has been initiated or completed

This is mostly implemented however an edge-case exists where:

- withdrawal reserve naturally falls into deficit through normal L2→L1 withdrawals

- no operations are triggered that would call `YieldManager::_pauseStakingIfNotAlready` for every unpaused yield operator
- ETH already exists in one or more Lido `stVault` from prior funding
- Lido's DEPOSITOR role makes a deposit

Impact: Beacon chain deposits can still occur when withdrawal reserve is in deficit, contrary to the protocol's safety criteria.

Recommended Mitigation: The only way to mitigate this is to:

- monitor the withdrawal reserve status off-chain
- if it falls into deficit, call `YieldManager::pauseStaking` for all yield providers
- once it is no longer in deficit, call `YieldManager::unpauseStaking` for all yield providers

This mitigation adds significant complexity and doesn't appear to provide that much value; perhaps change the protocol's safety criteria to allow for this scenario.

Linea: Acknowledged; we will have an off-chain Native Yield Automation Service to handle this.

7.4 Informational

7.4.1 Remove todo comments

Description: Remove "todo" comments:

```
LineaRollup.sol
38:  // TODO - Add access control to proxy admin only
```

Linea: Fixed in commit d8f57d5.

Cyfrin: Verified.

7.4.2 Consistently use `ErrorUtils::revertIfZeroAddress`

Description: Some parts of the code (eg `YieldProverBase::constructor`) use `ErrorUtils::revertIfZeroAddress` to verify that an input address is not the zero address. But other parts of the code don't do this, re-implementing the check. Use `ErrorUtils::revertIfZeroAddress` consistently in these other places:

- `LineaRollup::initialize, reinitializeLineaRollupV7` - yield manager checks
- `YieldManager::initialize` - `defaultAdmin` check

Linea: Fixed in commit b4b8ef5.

Cyfrin: Verified.

7.4.3 Redundant `FUNDER_ROLE` in `LineaRollupYieldExtension`

Description: The `LineaRollupYieldExtension` contract defines a `FUNDER_ROLE` constant with a comment indicating it is "The role required to call fund()". However, the `fund` function does not implement any access control and is callable by anyone:

```
function fund() external payable virtual {
  if (msg.value == 0) revert NoEthSent();
  emit FundingReceived(msg.value);
}
```

The function lacks the `onlyRole(FUNDER_ROLE)` modifier, making the defined role constant redundant and potentially misleading to developers, auditors, and users reviewing the code.

Impact: While the `FUNDER_ROLE` definition does not create a security vulnerability on its own, it introduces misleading documentation that conflicts with the actual implementation. This could lead to:

1. Confusion for developers maintaining the codebase who might assume access control is enforced when it is not
2. Potential future implementation errors if developers attempt to grant/revoke the `FUNDER_ROLE` expecting it to control access to the `fund` function

According to another comment in the contract, the `fund` function is intentionally designed to "accept both permissionless donations and YieldManager withdrawals," which suggests the lack of access control is deliberate. However, this contradicts the presence of the `FUNDER_ROLE` constant and its associated documentation.

Recommended Mitigation: Remove the unused `FUNDER_ROLE` constant and its associated comment since the `fund` function is intentionally permissionless. This will eliminate the inconsistency between the documentation and implementation.

If access control on the `fund` function is desired, add the `onlyRole(FUNDER_ROLE)` modifier. However, this would conflict with the stated intent to accept permissionless donations.

Linea: Fixed in commit 9876bc5.

Cyfrin: Verified.

7.4.4 Inconsistent storage location namespace root in `YieldManagerStorageLayout`

Description: In `YieldManagerStorageLayout` `YieldManagerStorage` is annotated with `@custom:storage-location erc7201:linea.storage.YieldManager`, but the hardcoded namespace root used by the contract is documented and chosen for a different identifier, `"linea.storage.YieldManagerStorage"`. Per ERC-7201, the namespace id used in the annotation must be the exact same id used to derive the storage root; otherwise the annotation does not describe the actual storage schema in use.

Impact: Off-chain tooling that reads the annotation and computes `erc7201("linea.storage.YieldManager")` will inspect the wrong storage root, leading to incorrect decoding, analysis, or monitoring of state. Standards non-compliance can cause confusion during audits and upgrades, and increases the risk of accidental collisions or mismatches if another component follows the annotated id literally.

Recommended Mitigation: Choose one of the following and keep the annotation, the comment, and the constant perfectly in sync:

1. Option A (minimal change): Update the annotation to match the slot's documented/computed id.
   - Change to: `@custom:storage-location erc7201:linea.storage.YieldManagerStorage`
2. Option B (preserve current annotation): Recompute the constant and its doc comment using the ERC-7201 formula for `"linea.storage.YieldManager"`, and update `YieldManagerStorageLocation` to that value.
   - Formula: `keccak256(abi.encode(uint256(keccak256(bytes("linea.storage.YieldManager"))) - 1)) & ~bytes32(uint256(0xff))`
   - Keep the comment and the hex value aligned with this id.

Linea: Fixed in commit f3e11c0.

Cyfrin: Verified.

7.4.5 Consider emitting event when synchronizing `lstLiabilityPrincipal`

Description: `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` synchronizes `$$.lstLiabilityPrincipal` which may have changed due to positive or negative `stETH` rebasing.

However this function emits no events even though it changes `$$.lstLiabilityPrincipal`; consider emitting an event with at least the delta change for easier off-chain tracking and auditability.

Linea: Fixed in commit 9f32daf.

Cyfrin: Verified.

7.4.6 Refactor duplicated checks into modifiers

Description: Refactor duplicated checks into modifiers:

- `YieldManager.sol`:

```
// duplicated in `receiveFundsFromReserve` and `withdrawLST`
if (msg.sender != L1_MESSAGE_SERVICE) {
  revert SenderNotL1MessageService();
}

// duplicated in `receiveFundsFromReserve`, `fundYieldProvider`, `unpauseStaking`
if (isWithdrawalReserveBelowMinimum()) {
  revert InsufficientWithdrawalReserve();
}

// duplicated in `unstakePermissionless` and `replenishWithdrawalReserve`
if (!isWithdrawalReserveBelowMinimum()) {
  revert WithdrawalReserveNotInDeficit();
}
```

```
// duplicated in `initiateOssification` and `progressPendingOssification`
if ($$.isOssified) {
  revert AlreadyOssified();
}
```

Linea: Fixed in commit e832420 implemented the first 3 suggestions.

Cyfrin: Verified.

7.4.7 Withdrawing LSTs can result in a system balance below the minimum reserve requirement

Description: One core protocol property is: "All ETH transfer operations involving the YieldManager MUST ensure that the withdrawal reserve remains minimum threshold."

However neither `LineaRollupYieldExtension::claimMessageWithProofAndWithdrawLST` nor `YieldManager::withdrawLST` call `YieldManager::isWithdrawalReserveBelowMinimum` to enforce that the LST withdrawal doesn't result in the protocol being under its minimum reserve requirement.

And even if this call were present, `YieldManager::_getTotalSystemBalance`:

- doesn't factor outstanding LST liabilities when calculating the total system balance even though an LST withdrawal is a liability which will need to be repaid that withdraws funds out of the system
- is affected by the corruption of `userFundsInYieldProvidersTotal` when LST liabilities are repaid as highlighted in a separate issue

Impact: LST withdrawals can violate the minimum reserve requirement resulting in the protocol having less reserves than it requires.

Recommended Mitigation: * `YieldManager::withdrawLST` should call `isWithdrawalReserveBelowMinimum` after completing everything to verify that the LST withdrawal did not breach the minimum reserve requirement

- `YieldManager::_getTotalSystemBalance` should deduct outstanding LST liabilities when calculating the total system balance
- The corruption of `userFundsInYieldProvidersTotal` when LST liabilities are repaid should be fixed (separate issue).

Linea: Acknowledged; from a feature standpoint, `LineaRollupYieldExtension::claimMessageWithProofAndWithdrawLST` → `YieldManager::withdrawLST` serves as an emergency pathway that allows users to withdraw LST only when the reserve is fully depleted.

So this flow is an intentional exception to the tech spec property of "All ETH transfer operations involving the YieldManager MUST ensure that the withdrawal reserve remains minimum threshold."

These two points will be addressed by mitigations to other issues:

- `YieldManager::_getTotalSystemBalance` should deduct outstanding LST liabilities when calculating the total system balance
- The corruption of `userFundsInYieldProvidersTotal` when LST liabilities are repaid should be fixed (separate issue).

7.4.8 User principal can be permissionlessly used to pay system obligations violating safety property

Description: The protocol's technical specification states:

> User principal (L1 deposits + L2 circulating ETH) MUST NOT be used for system obligations; all obligations MUST be settled exclusively from unreported yield which ensures that users funds are insulated from ongoing system obligations

However permissionless actors can directly call LidoV3 functions such as `VaultHub::settleLidoFees` the protocol's `stVault` which will use deposited ETH to pay system obligations.

Impact: Apart from violating this safety property there doesn't appear to be further impact since the protocol:

- tracks both the total deposited ETH `userFunds` and the per-vault deposited ETH `userFundsInYieldProvidersTotal` which it owes to users
- treats ETH outflows due to external liability settlement as negative yield which will eventually be offset by staking yield

Linea: Acknowledged.

7.4.9 `YieldManager::fundYieldProvider` does not automatically unpause staking after funding

Description: In `YieldManager::fundYieldProvider`, when liquidity is transferred back to a yield provider to restore its balance or repay liabilities, the function does **not automatically unpause staking** — even if all safety conditions (e.g., sufficient withdrawal reserve, cleared LST liabilities, and completed ossification) are satisfied afterward.

As a result, a yield provider can remain **stuck in a paused state** despite being fully solvent and capable of resuming normal operation. This behavior introduces unnecessary downtime in staking operations and may reduce protocol efficiency or yield generation.

Recommended Mitigation: Enhance `fundYieldProvider()` (or the underlying `_fundYieldProvider` helper) to **attempt an automatic unpause** when the provider becomes eligible after funding.

Linea: Acknowledged; we prioritize Linea user liquidity over yield efficiency — quick to pause staking when reserves are low, cautious to resume. The Native Yield Automation Service will handle the unpause timing based on withdrawal reserve conditions.

7.5 Gas Optimization

7.5.1 Fast fast by performing input-related checks first

Description: If a function call were to revert due to invalid input, there's no point performing other unrelated work before that happens. Fast fast by performing input-related checks first:

- `LineaRollup::initialize, reinitializeLineaRollupV7` - perform the valid yield manager address check before everything else. Consider creating a modifier and adding the modifier to both functions to reduce code duplication

Linea: Fixed in commit b4b8ef5.

Cyfrin: Verified.

7.5.2 Refactor `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` to eliminate `liabilityETH`

Description: Local variable `liabilityETH` can be refactored away in `LidoStVaultYieldProvider::_syncExternalLiabilitySettlement` by using the named return variable:

```
function _syncExternalLiabilitySettlement(
  YieldProviderStorage storage $$,
  uint256 _liabilityShares,
  uint256 _lstLiabilityPrincipalCached
) internal returns (uint256 lstLiabilityPrincipalSynced) {
  lstLiabilityPrincipalSynced = STETH.getPooledEthBySharesRoundUp(_liabilityShares);
  // If true, this means an external actor settled liabilities.
  if (lstLiabilityPrincipalSynced < _lstLiabilityPrincipalCached) {
    $$.lstLiabilityPrincipal = lstLiabilityPrincipalSynced;
  } else {
    lstLiabilityPrincipalSynced = _lstLiabilityPrincipalCached;
  }
}
```

Linea: Fixed in commit 5b55950.

Cyfrin: Verified.