Lido Circuit Breaker Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

Dacian

SBSecurity (Blckhv, Slavcheww)

May 1, 2026

Contents

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 in-scope code being audited.

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

`CircuitBreaker` is an emergency pause manager for the Lido protocol — a single contract that lets designated pausers pause critical contracts on behalf of the admin.

There are two roles: the **admin** (Lido Agent) and **pausers** (multisig committees such as a standard Gnosis multisig). The admin assigns one pauser to each pausable contract. In an emergency, the pauser triggers a pause on a contract address for a globally configured pause duration. After a successful pause the pauser assignment is cleared — the admin must explicitly re-assign before the contract can be paused again through `CircuitBreaker`.

Pausers must periodically send a heartbeat to prove liveness. A pauser whose heartbeat has expired cannot pause or refresh their heartbeat; only the admin can restore an expired pauser by re-registering them.

The protocol consists of two contracts:

- `CircuitBreaker` — the main contract with an immutable admin, configurable pause duration and heartbeat interval (within deployment-time min/max bounds), and functions for admin configuration (`setPauseDuration`, `setHeartbeatInterval`, `registerPauser`), pauser liveness (`heartbeat`), and emergency pause (`pause`). Uses a transient storage reentrancy guard (EIP-1153) on the `pause` function to prevent a malicious pausable from re-entering to trigger additional pauses. Must be granted `PAUSER_ROLE` on the target pausable contracts in order to call `pauseFor`
- `Registry` — a library managing an enumerable pausable-to-pauser registry using a swap-and-pop pattern for O(1) additions and removals. Maintains bidirectional consistency between a `pausables` array, `oneBasedIndex` mapping, `pauser` mapping, and per-pauser `pausableCount`. Idempotent unregistration is supported by design to prevent a pauser from griefing the admin by frontrunning an unregistration transaction with a `pause` call

The pausable contracts this system is expected to manage on Ethereum mainnet include WithdrawalQueue, ValidatorExitBusOracle, TriggerableWithdrawalsGateway, VaultHub, PredepositGuarantee, and CSFeeOracle — all of which implement `pauseFor(uint256)` with `onlyRole(PAUSE_ROLE)` and `isPaused()` via Lido's `PausableUntil` base contract.

Paused contracts can be unpaused by `RESUME_ROLE` which is expected to be given to Lido Agent on-demand and then revoked immediately after. In case of `CircuitBreaker`, it doesn't need the role because the pause is temporary and auto-resumes after the pause duration has expired.

5 Audit Scope

The audit scope was limited to:

```
src/CircuitBreaker.sol
src/Registry.sol
```

6 Executive Summary

Over the course of 2 days, the Cyfrin team conducted an audit on the Circuit Breaker code provided by Lido. The findings consist of 2 Informational & 3 Gas Optimizations.

6.1 Centralization Risks

The immutable `ADMIN` (Lido Agent) has significant centralized control:

- Sole authority over pauser assignment - only `ADMIN` can register, replace, or unregister pausers. If the DAO agent is compromised or governance is captured, an attacker could unregister all pausers (disabling emergency pause capability) or register malicious pausers
- Controls pause parameters — `ADMIN` sets `pauseDuration` and `heartbeatInterval` within deployment-time bounds. Misconfiguration (e.g., very low heartbeat interval causing all pausers to expire, or very low pause duration making pauses ineffective) could degrade emergency response capability
- Only entity that can revive expired pausers — if a pauser's heartbeat expires, only `ADMIN` can restore coverage by re-registering them. Combined with the single-pauser-per-pausable design, this creates a window of zero emergency pause coverage during liveness gaps (see I-2)
- No admin transfer mechanism — `ADMIN` is immutable. If the Lido Agent address needs to change, `CircuitBreaker` must be redeployed and all pausable contracts must re-grant `PAUSE_ROLE` to the new deployment

6.2 Protocol Invariants

The following invariants were verified using the Certora Prover (38 of 41 rules verified; 3 known limitations marked with * below):

Registry Enumerable Set Integrity

1. For every valid index `i < pausables.length`: `oneBasedIndex[pausables[i]] == i + 1`
2. For every address `p` with `oneBasedIndex[p] != 0`: `pausables[oneBasedIndex[p] - 1] == p`
3. For address `p != address(0)`: `pauser[p] != address(0)` if and only if `oneBasedIndex[p] != 0` *
4. For every valid index `i < pausables.length`: `pauser[pausables[i]] != address(0)`
5. For all valid indices `i != j`: `pausables[i] != pausables[j]` (no duplicates)
6. For address `p` not in `pausables[]`: `oneBasedIndex[p] == 0` and `pauser[p] == address(0)` *
7. For any pauser `q`: `pausableCount[q]` equals the count of pausable addresses where `pauser[p] == q`
8. Sum of `pausableCount[q]` across all pausers equals `pausables.length` *
9. `pausableCount[address(0)] == 0`
10. `address(0)` is never in `pausables[]`

Access Control

11. `pauseDuration` changes only when `msg.sender == ADMIN`
12. `heartbeatInterval` changes only when `msg.sender == ADMIN`
13. Registry state changes only via `registerPauser` (ADMIN) or `pause` (authorized pauser)
14. `heartbeatExpiry[p]` changes only via `registerPauser` (ADMIN), `heartbeat` (self), or `pause` (self)

Bounds Enforcement

15. `MIN_PAUSE_DURATION <= pauseDuration <= MAX_PAUSE_DURATION` — always
16. `MIN_HEARTBEAT_INTERVAL <= heartbeatInterval <= MAX_HEARTBEAT_INTERVAL` — always

Heartbeat Liveness

17. `heartbeat()` reverts if `block.timestamp >= heartbeatExpiry[msg.sender]`
18. `pause()` reverts if `block.timestamp >= heartbeatExpiry[msg.sender]`
19. Only ADMIN can refresh an expired pauser's heartbeat (via `registerPauser`)
20. After heartbeat refresh: `heartbeatExpiry[pauser] == block.timestamp + heartbeatInterval`

Single-Use Pause

21. After successful `pause(_pausable)`: `pauser[_pausable] == address(0)` (consumed)
22. After successful `pause(_pausable)`: `pausableCount[msg.sender]` decreased by 1

Revert Conditions

23. `setPauseDuration`, `setHeartbeatInterval`, `registerPauser` revert if `msg.sender != ADMIN`
24. `heartbeat()` reverts if `pausableCount[msg.sender] == 0`
25. `pause(_pausable)` reverts if `msg.sender != pauser[_pausable]`
26. `registerPauser(_pausable, _)` reverts if `_pausable == address(0)`

Edge Cases

27. Idempotent unregistration: `registerPauser(p, 0)` when already unregistered is a net no-op
28. Replace-with-self: `registerPauser(p, q)` when `pauser[p] == q` preserves `pausableCount[q]`

Liveness

29. A registered, live pauser CAN successfully call `pause` (happy path reachable)
30. A registered, live pauser CAN successfully call `heartbeat`

* *Known limitation*: These 3 invariants are verified for all functions except `registerPauser`, where the Certora Prover cannot close the inductive proof due to circular dependencies between the enumerable set structural invariants. The properties are correct by manual analysis and verified for all other code paths including `pause` (which also modifies registry state). See `certora/README.md` for details.

6.3 Post-Audit Change Review

After the audit had concluded we were asked to review commit b4b2fbc which cleared the heartbeat when a pauser was left with no registered contracts, causing `CircuitBreaker::isPauserLive` to return `false` in this case. There were no issues found in the diff between commits.

6.4 Deployment Bytecode Verification

We were asked to verify that at commit b4b2fbc the locally compiled bytecode for `CircuitBreaker` matched the bytecode deployed to Ethereum mainnet at 0x6019cb557978296ba3c08a7b73225c0975dfb2f7. We used the following reproducible process to do this.

1) prepare the repo:

```
gh repo clone lidofinance/circuit-breaker
cd circuit-breaker
git checkout b4b2fbc921b3191560a3fc62d502d4bb98ad99e1
forge install
forge build
```

2) 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)
```

3) verify `CircuitBreaker` deployed bytecode using:

- `python3 extract_deployed_bytecode.py ETH_RPC_URL 0x6019cb557978296ba3c08a7b73225c0975dfb2f7 CircuitBreaker.mainnet.txt` to extract mainnet deployed bytecode
- `python3 compare_bytecodes.py out/CircuitBreaker.sol/CircuitBreaker.json CircuitBreaker.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 342-343 (length 2): local 0000 vs deployed 4f1a (MAX_PAUSE_DURATION high bytes of 0x4f1a00 =
  ↪  5184000 = 60 days)
  Range 364-383 (length 20): local 0000000000000000000000000000000000000000 vs deployed
  ↪  3e40d73eb977dc6a537af587d48316fee66e9c8c (ADMIN = Lido DAO Agent)
  Range 573-575 (length 3): local 000000 vs deployed 069780 (MIN_PAUSE_DURATION = 432000 = 5 days)
```

```
Range 611-614 (length 4): local 00000000 vs deployed 05a39a80 (MAX_HEARTBEAT_INTERVAL = 94608000 =
↪  1095 days)
Range 670-671 (length 2): local 0000 vs deployed 278d (MIN_HEARTBEAT_INTERVAL high bytes of 0x278d00 =
↪  2592000 = 30 days)
Range 832-851 (length 20): local 0000000000000000000000000000000000000000 vs deployed
↪  3e40d73eb977dc6a537af587d48316fee66e9c8c (ADMIN = Lido DAO Agent)
Range 1365-1384 (length 20): local 0000000000000000000000000000000000000000 vs deployed
↪  3e40d73eb977dc6a537af587d48316fee66e9c8c (ADMIN = Lido DAO Agent)
Range 2272-2291 (length 20): local 0000000000000000000000000000000000000000 vs deployed
↪  3e40d73eb977dc6a537af587d48316fee66e9c8c (ADMIN = Lido DAO Agent)
Range 3530-3531 (length 2): local 0000 vs deployed 278d (MIN_HEARTBEAT_INTERVAL high bytes of 0x278d00
↪  = 2592000 = 30 days)
Range 3619-3622 (length 4): local 00000000 vs deployed 05a39a80 (MAX_HEARTBEAT_INTERVAL = 94608000 =
↪  1095 days)
Range 3775-3777 (length 3): local 000000 vs deployed 069780 (MIN_PAUSE_DURATION = 432000 = 5 days)
Range 3865-3866 (length 2): local 0000 vs deployed 4f1a (MAX_PAUSE_DURATION high bytes of 0x4f1a00 =
↪  5184000 = 60 days)
```

Each immutable was independently confirmed by reading the on-chain getter via `cast call 0x6019cb557978296ba3c08a7b73225c0975dfb2f7 "<getter>"`:

- `ADMIN` = `0x3e40D73EB977Dc6a537aF587D48316feE66E9C8c` (Lido DAO Agent)
- `MIN_PAUSE_DURATION` = 432000
- `MAX_PAUSE_DURATION` = 5184000
- `MIN_HEARTBEAT_INTERVAL` = 2592000
- `MAX_HEARTBEAT_INTERVAL` = 94608000

Summary

Project Name	Circuit Breaker
Repository	circuit-breaker
Commit	a9e4d1e30faf...
Fix Commit	b4b2fbc921b3...
Audit Timeline	Apr 6th - Apr 7th, 2026
Methods	Manual Review, Formal Verification

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	0
Informational	2
Gas Optimizations	3
Total Issues	5

Summary of Findings

[I-1] Missing named mapping value parameters in `Registry.Storage`	Resolved
[I-2] Single pauser per pausable combined with heartbeat expiry lockout leaves zero emergency coverage during liveness gaps	Acknowledged
[G-1] Emit before state change to eliminate temporary variable	Resolved
[G-2] Solidity optimizer is not enabled	Resolved
[G-3] Redundant storage read of `registry.pauser[_pausable]` in `CircuitBreaker::pause`	Acknowledged

7 Findings

7.1 Informational

7.1.1 Missing named mapping value parameters in `Registry.Storage`

Description: Three mappings in `Registry.Storage` declare named keys but omit names for their value types, reducing readability and ABI clarity:

```
// Registry.sol lines 18-21
mapping(address pausable => address) pauser;
mapping(address pausable => uint256) oneBasedIndex;
mapping(address pauser => uint256) pausableCount;
```

Recommended Mitigation:

```
-mapping(address pausable => address) pauser;
-mapping(address pausable => uint256) oneBasedIndex;
-mapping(address pauser => uint256) pausableCount;
+mapping(address pausable => address pauser) pauser;
+mapping(address pausable => uint256 oneBasedIndex) oneBasedIndex;
+mapping(address pauser => uint256 pausableCount) pausableCount;
```

Lido: Fixed in commit cabcfec.

Cyfrin: Verified.

7.1.2 Single pauser per pausable combined with heartbeat expiry lockout leaves zero emergency coverage during liveness gaps

Description: Each pausable contract can only have one configured pauser in `CircuitBreaker`. If that pauser's heartbeat expires, they are completely locked out — both `CircuitBreaker::heartbeat` and `CircuitBreaker::pause` require `isPauserLive` to return true (via `_updateHeartbeat` with `_requireActive = true`):

```
// CircuitBreaker.sol
246:     function heartbeat() external {
247:         require(registry.isRegistered(msg.sender), SenderNotPauser());
248:         _updateHeartbeat(msg.sender, true);  // requires live
249:     }

255:     function pause(address _pausable) external nonReentrant {
256:         require(msg.sender == registry.getPauser(_pausable), SenderNotPauser());
257:         _updateHeartbeat(msg.sender, true);  // requires live

276:     function _updateHeartbeat(address _pauser, bool _requireActive) internal {
277:         if (_requireActive) require(isPauserLive(_pauser), HeartbeatExpired());
```

The only way to restore an expired pauser is the admin calling `CircuitBreaker::registerPauser` to re-register them — this calls `_updateHeartbeat(_newPauser, false)` which bypasses the liveness check.

This creates a gap: between heartbeat expiry and admin re-registration, the pausable contract has zero emergency pause coverage through `CircuitBreaker`. If an emergency occurs during this window, the slow governance path is the only option — exactly the scenario `CircuitBreaker` was built to prevent.

The design rationale is sound ("A committee that cannot prove its liveness should not be trusted to respond in an emergency"), but the single-pauser-per-pausable constraint means there is no fallback. Consider whether the `Registry` contract used by `CircuitBreaker` should support multiple pausers for each pausable contract.

Lido: Acknowledged as:

- Single pauser keeps accountability clear

- Expiry is the enforcement mechanism; a committee that can't send one heartbeat transaction once per year (for example) shouldn't hold pause authority
- Heartbeat expiry is fully trackable; off-chain monitoring can alert pausers months in advance before expiry
- the tradeoff of having no fallback is accepted

7.2 Gas Optimization

7.2.1 Emit before state change to eliminate temporary variable

Description: `CircuitBreaker::_setPauseDuration` and `CircuitBreaker::_setHeartbeatInterval` each declare a `previous*` local variable solely to capture the old storage value for the event. Emitting the event before the state change lets the event read the current storage value directly, eliminating the temporary:

```
// CircuitBreaker.sol lines 289-292
uint256 previousPauseDuration = pauseDuration;
pauseDuration = _newPauseDuration;
emit PauseDurationUpdated(previousPauseDuration, _newPauseDuration);

// CircuitBreaker.sol lines 300-303
uint256 previousHeartbeatInterval = heartbeatInterval;
heartbeatInterval = _newHeartbeatInterval;
emit HeartbeatIntervalUpdated(previousHeartbeatInterval, _newHeartbeatInterval);
```

Recommended Mitigation:

```
 function _setPauseDuration(uint256 _newPauseDuration) internal {
     require(_newPauseDuration >= MIN_PAUSE_DURATION, PauseDurationBelowMin());
     require(_newPauseDuration <= MAX_PAUSE_DURATION, PauseDurationAboveMax());
-    uint256 previousPauseDuration = pauseDuration;
+    emit PauseDurationUpdated(pauseDuration, _newPauseDuration);
     pauseDuration = _newPauseDuration;
-    emit PauseDurationUpdated(previousPauseDuration, _newPauseDuration);
 }
```

Same pattern for `CircuitBreaker::_setHeartbeatInterval`.

Lido: Fixed in commit 11aea85.

Cyfrin: Verified.

7.2.2 Solidity optimizer is not enabled

Description: The Foundry configuration does not enable the Solidity optimizer:

```
foundry.toml
[profile.default]
src = "src"
out = "out"
libs = ["lib"]
solc = "0.8.34"
```

No `optimizer` or `optimizer_runs` keys are present. Foundry disables the optimizer by default when these are absent. Even at the default 200 runs, the optimizer eliminates dead code, simplifies constant expressions, and reduces both deployment and runtime gas.

Recommended Mitigation: Add optimizer settings to `foundry.toml`:

```
 [profile.default]
 src = "src"
 out = "out"
 libs = ["lib"]
 solc = "0.8.34"
+optimizer = true
+optimizer_runs = 10000
```

`optimizer_runs` controls the trade-off between deployment cost and runtime call cost. Lower values (e.g., 1) produce smaller bytecode that is cheaper to deploy but slightly more expensive per call. Higher values (e.g., 10,000+) produce larger bytecode optimized for cheaper calls at the cost of more expensive deployment.

Factory-deployed contracts benefit from lower values because deployment cost is paid every time the factory creates a new instance — with hundreds of deployments the cumulative savings outweigh the marginal per-call increase.

Core protocol contracts deployed once but called millions of times benefit from higher values since deployment is a one-time expense. `CircuitBreaker` falls into the latter category — it is deployed once and called repeatedly — so a higher `optimizer_runs` value (e.g., 10,000) is appropriate. The default 200 is a reasonable starting point if unsure.

Lido: Fixed in commit 30b01f1.

Cyfrin: Verified.

7.2.3 Redundant storage read of `registry.pauser[_pausable]` in `CircuitBreaker::pause`

Description: In `CircuitBreaker::pause`, the storage slot `registry.pauser[_pausable]` is read twice via separate library calls:

```
CircuitBreaker.sol
255:     function pause(address _pausable) external nonReentrant {
256:         require(msg.sender == registry.getPauser(_pausable), SenderNotPauser());  // SLOAD 1
257:         _updateHeartbeat(msg.sender, true);
258:
259:         uint256 duration = pauseDuration;
260:         IPausable target = IPausable(_pausable);
261:
262:         registry.setPauser(_pausable, address(0));  // SLOAD 2 (re-reads pauser[_pausable] at
↪  Registry.sol:85)
```

`Registry::getPauser` reads `_self.pauser[_pausable]` (Registry.sol:44) for the authorization check. Then `Registry::setPauser` reads the same slot again (Registry.sol:85: `address previousPauser = _self.pauser[_pausable]`) to determine the previous pauser for count management. The value has not changed between reads, so the second SLOAD (100 gas warm) is redundant.

Recommended Mitigation: `Registry::setPauser` already reads `previousPauser` — return it via a named return value and use it in `CircuitBreaker::pause` to combine the authorization check with the unregistration:

```
 // Registry.sol
-function setPauser(Storage storage _self, address _pausable, address _newPauser) internal {
+function setPauser(Storage storage _self, address _pausable, address _newPauser) internal returns
↪  (address previousPauser) {
     require(_pausable != address(0), PausableZero());

-    address previousPauser = _self.pauser[_pausable];
+    previousPauser = _self.pauser[_pausable];

     // ... existing logic unchanged ...
 }
```

```
 // CircuitBreaker.sol
 function pause(address _pausable) external nonReentrant {
-    require(msg.sender == registry.getPauser(_pausable), SenderNotPauser());
+    require(msg.sender == registry.setPauser(_pausable, address(0)), SenderNotPauser());
     _updateHeartbeat(msg.sender, true);

     uint256 duration = pauseDuration;
     IPausable target = IPausable(_pausable);

-    registry.setPauser(_pausable, address(0));
     target.pauseFor(duration);
     require(target.isPaused(), PauseFailed());
```

```
    emit PauseTriggered(_pausable, msg.sender, duration);
}
```

Lido: Acknowledged; 100 gas saved is negligible compared to the decreased readability for a function call that we hope will never be called. The proposed fix:

- hides state mutation inside an auth check
- returns the previous pauser from `setPauser`, which breaks the standard convention where a setter usually returns the new value (or a success boolean)