hashlock.

Security Audit

Lay3r Labs (Blockchain Infrastructure)

September 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

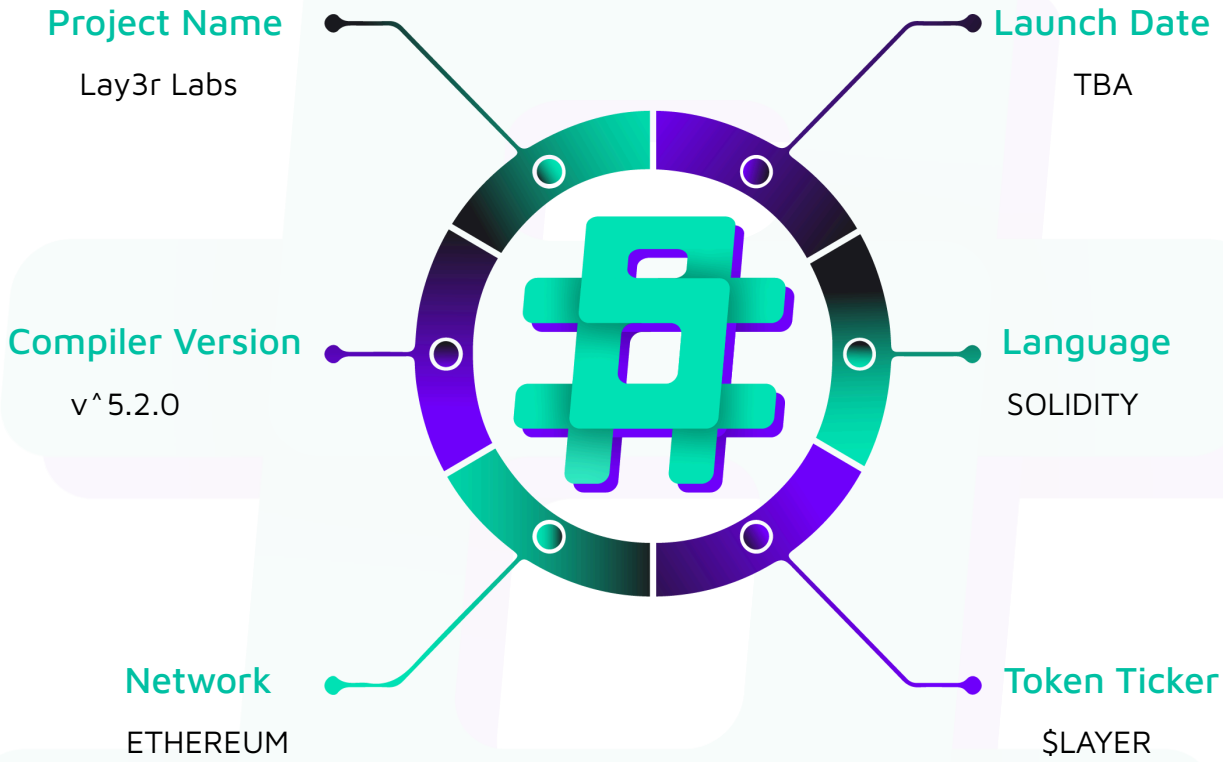
The Lay3r Labs team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

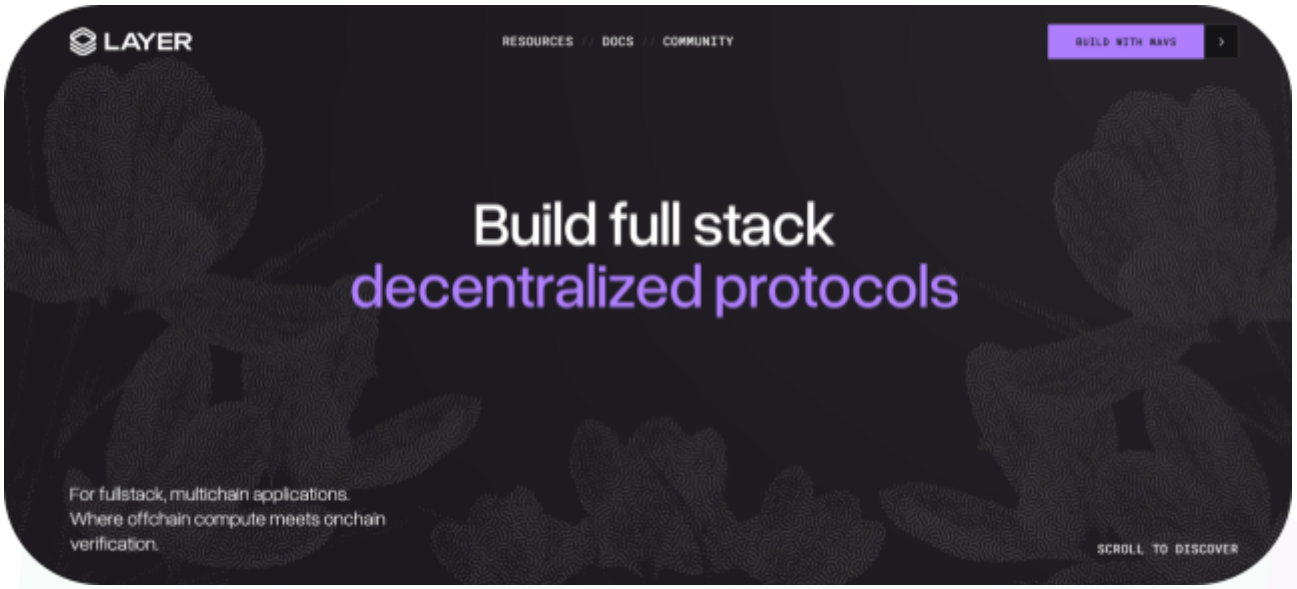
Lay3r Labs is a decentralized application development platform that empowers developers to build full-stack, multichain applications by integrating off-chain compute with on-chain verification. At its core is WAVS (WebAssembly Verifiable Services), a runtime environment that allows developers to create modular, composable services such as AI agents, custom oracles, and decentralized storage solutions. These services are designed to be deterministic, meaning their outputs are predictable and verifiable, which is crucial for applications requiring high trust and security. By leveraging WebAssembly and Rust, Layer.xyz provides a robust foundation for building decentralized protocols that can operate across multiple blockchain ecosystems.

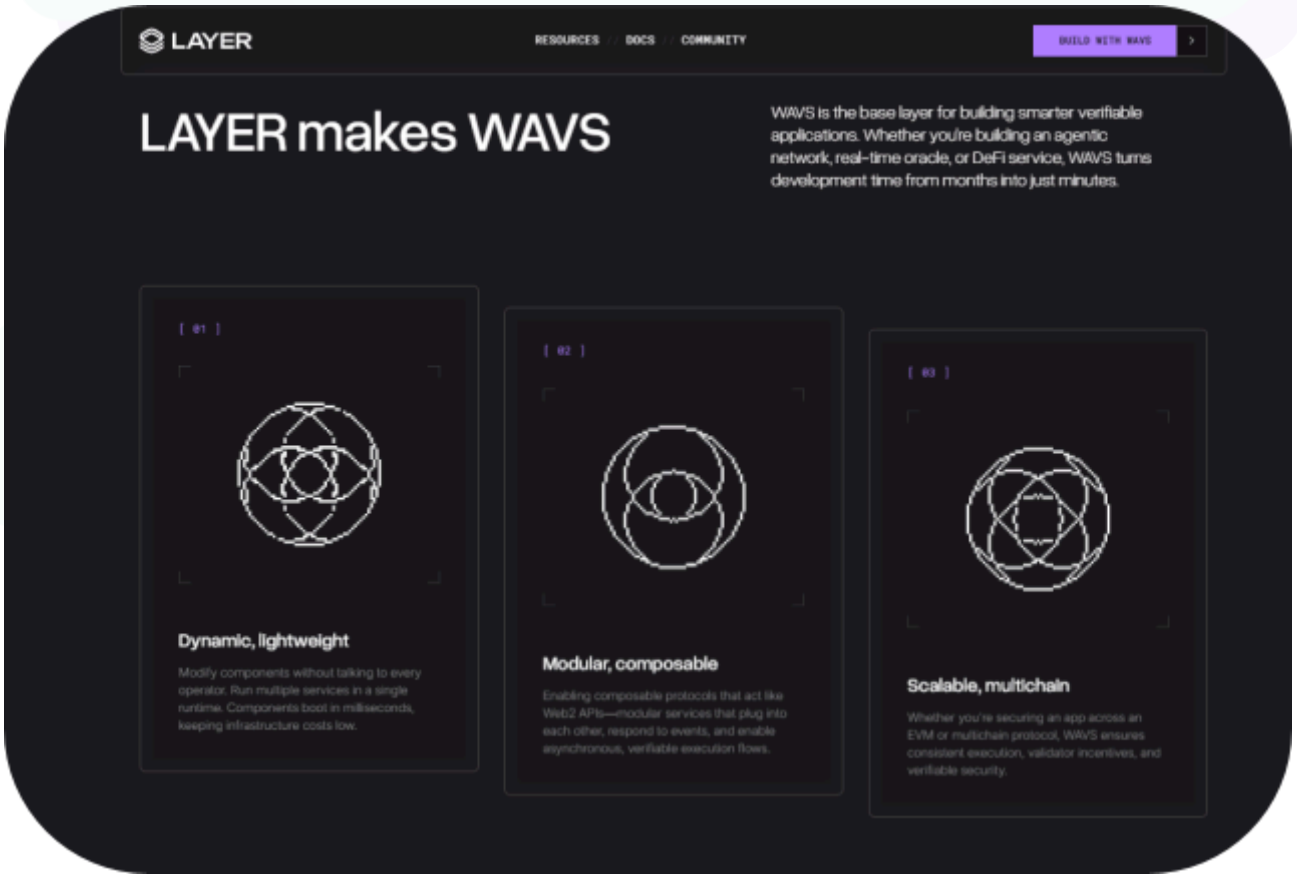
Project Name: Lay3r Labs
Project Type: Blockchain Infrastructure
Compiler Version: ^5.2.0
Website: https://www.layer.xyz/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the Lay3r Labs project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Lay3r Labs Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**September, 2025**
Contract 1	POAStakeRegistry.sol
Contract 2	POAStakeRegistryStorage.sol
Contract 3	IPOAStakeRegistry.sol
Audited GitHub Commit Hash	c10b0cd69252895798cea6c028e33d2879f07363
Fix Review GitHub Commit Hash	00f7dac58e444643f5e6e19387d472423099d226

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

1 High severity vulnerabilities

3 Medium severity vulnerabilities

4 Low severity vulnerabilities

2 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

POAStakeRegistry
• Allows operator to:
  ○ Update their own operator signing key
  ○ Query latest or historical operator signing key
  ○ Query operator registration status
  ○ Query operator weight (latest or at block)
  ○ Query total, threshold, and quorum weights (latest or at block)
• Allows admins to:
  ○ Register new operators with weight
  ○ Deregister operators
  ○ Update operator weights
  ○ Update stake threshold
  ○ Update quorum parameters
  ○ Set service URI
	Contract achieves this functionality.

POAStakeRegistryStorage
• Allows users to:
  ○ None
• Allows admins to:
  ○ None
	Contract achieves this functionality.

Code Quality

This audit scope involves the smart contracts of the Lay3r Labs project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring were recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Lay3r Labs project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] POAStakeRegistry#updateOperatorSigningKey - Signing Key Hijacking can lead to AVS DOS

Description

The contract does not enforce that operator signing keys must be unique. This allows any operator to claim a signing key already in use by another, leading to a "last writer wins" scenario where the hijacker seizes control of the key.

Vulnerability Details

The `_updateOperatorSigningKey` function is vulnerable to two distinct exploits. First, it does not check if a `newSigningKey` is already the active key for another operator. This allows a hijacker to claim an active key, causing the system to incorrectly attribute signatures to the hijacker instead of the original owner.

Second, the function's cleanup logic can be abused. A malicious operator can register an unused key and then immediately update to a different key within the same transaction. This action triggers `(_signingKeyOperatorHistory[oldSigningKey].push(...))`, which sets the operator for the first key to address(0). Consequently, any future attempts to validate a signature from that key will cause `the _getOperatorForSigningKey` function to revert.

```
/// @inheritdoc IPOAStakeRegistry

    function updateOperatorSigningKey(

        address newSigningKey

    ) external {

        if (!_operatorRegistered[msg.sender]) {

            revert OperatorNotRegistered();
```

```
        }

        _updateOperatorSigningKey(msg.sender, newSigningKey);

    }
```

Impact

This issue allows a malicious operator with low stake to perform a cheap and effective `DOS` attack on the `AVS`. By hijacking the active keys of high-stake operators, an attacker can neutralize their stake-weight. By using the register-and-delete exploit, an attacker can also preemptively disable keys they anticipate will be used by new operators.

Recommendation

Implement a uniqueness check in `the _updateOperatorSigningKey` function. Before a key is assigned, the contract must verify that the key is not currently active for another operator. If the key is already in use, the transaction should revert, ensuring that each active signing key is exclusively tied to a single operator.

```
error InvalidSigningKey();

error SigningKeyInUse(address key, address currentOwner);

function _updateOperatorSigningKey(address operator, address newSigningKey) internal {

    if (newSigningKey == address(0)) revert InvalidSigningKey();

                                            address           currentOwner           =
address(uint160(_signingKeyOperatorHistory[newSigningKey].latest()));

    if (currentOwner != address(0) && currentOwner != operator) {

        revert SigningKeyInUse(newSigningKey, currentOwner);

    }

    // proceed with current logic...}
```

Status

Resolved

Medium

[M-01] POAStakeRegistry#initialize - Implementation Contract Can Be Initialized (Missing `_disableInitializers()`)

Description

The `POAStakeRegistry` implementation (logic) contract can be initialized directly because the original code did not disable initializers at the implementation level. Any externally owned account can call initialize on the logic contract, set arbitrary parameters, and assume ownership. While this does not immediately seize control of proxied instances, it leaves the implementation mutable and creates confusion and upgrade-safety risks.

Vulnerability Details

In the original code, the implementation contract lacked a constructor that calls `_disableInitializers`. Consequently, the external initializer function is callable on the logic contract itself, not just on the proxy instance. This allows any caller to run initialization logic and become the owner of the implementation.

```
// Original (vulnerable): no constructor present to disable initializers
function initialize(
    address initialOwner,
    uint256 thresholdWeight,
    uint256 quorumNumerator,
    uint256 quorumDenominator
) external initializer {
    _updateStakeThreshold(thresholdWeight);
    _updateQuorum(quorumNumerator, quorumDenominator);
    __Ownable_init(initialOwner);
}
```

Because the implementation is left with an active initializer, an attacker can directly invoke it on the logic contract address, establishing ownership and setting arbitrary governance parameters on the implementation instance.

Proof of Concept

```
function testUninitializedImplementation() public {
    // --- Setup and Preconditions ---
    // Deploy ProxyAdmin as the upgrade admin
    vm.startPrank(owner);
    address proxyAdmin = UpgradeableProxyLib.deployProxyAdmin();

    // Deploy an empty proxy and implementation contract
    poaStakeRegistry = POAStakeRegistry(UpgradeableProxyLib.setUpEmptyProxy(owner));
    address poaStakeRegistryImpl = address(new POAStakeRegistry());

    // Prepare initialization calldata for the proxy upgrade
    bytes memory poaStakeRegistryUpgradeCall = abi.encodeCall(
        POAStakeRegistry.initialize, (msg.sender, 100, 1, 1)
    );

    // --- Exploit Execution ---
    // Upgrade proxy to implementation and initialize via proxy
    UpgradeableProxyLib.upgradeAndCall(
        address(poaStakeRegistry), poaStakeRegistryImpl, poaStakeRegistryUpgradeCall
    );
    vm.stopPrank();

    // Directly initialize the implementation contract (should not be allowed)
    vm.startPrank(nonOperator);
    POAStakeRegistry(poaStakeRegistryImpl).initialize(
                    nonOperator,   INITIAL_THRESHOLD_WEIGHT,   INITIAL_QUORUM_NUMERATOR,
INITIAL_QUORUM_DENOMINATOR
    );

    // --- Post-condition Check ---
    // Implementation contract owner is now nonOperator (should not be possible)
    assertEq(
        POAStakeRegistry(poaStakeRegistryImpl).owner(),
        nonOperator
    ); // PoC succeeds if implementation contract can be initialized by attacker

    vm.stopPrank();
```

```
}
```

Impact

Ownership of the implementation enables unauthorized configuration of a live logic instance and can facilitate confusion or social-engineering attacks around "which instance is canonical." It also weakens upgrade hygiene and can interact dangerously with future changes (`e.g.`, if shared state or callbacks are introduced). This is a well-established anti-pattern for upgradeable contracts.

Recommendation

Add a constructor to the implementation that permanently disables initialization on the logic contract.

```
+   constructor() {
+       _disableInitializers();
+   }
```

Status

Resolved

[M-02] POAStakeRegistry#_checkSignatures - Missing Handling of max uint32 Reference Block

Description

The contract documentation states that passing `referenceBlock = type(uint32).max` should allow the system to use the latest stake weight and quorum parameters. However, the actual implementation does not contain any branch logic to handle this case. Instead, the current implementation enforces `referenceBlock < block.number` across all lookup functions, causing a revert when `type(uint32).max` is provided.

Vulnerability Details

In the `_checkSignatures` function, referenceBlock is forwarded to internal functions like `_getOperatorForSigningKey`, `_getOperatorWeight`, and `_getTotalWeight`. Each of these functions checks:

```
if (!(referenceBlock < block.number)) {
    revert InvalidReferenceBlock();
}
```

The NatSpec comments explicitly state that max uint32 should serve as a sentinel value for using the latest values:

```
/// @param referenceBlock The block number for evaluating stake weight;
/// use max uint32 for latest weight.
```

Since there is no conditional branch to handle this case, calls that attempt to use max uint32 will consistently revert instead of falling back to the latest values.

```
function _getTotalWeight(
    uint32 referenceBlock
) internal view returns (uint256) {
    if (!(referenceBlock < block.number)) {
        revert InvalidReferenceBlock();
    }
    return _totalWeightHistory.upperLookup(uint96(referenceBlock));
}
```

This creates a mismatch between intended behavior (documented) and actual implementation.

Impact

Any external system or user relying on the documented max uint32 sentinel value will face unexpected reverts, breaking interoperability with EigenLayer or other clients. This prevents usage of the latest weight/quorum evaluation path and could cause system-level failures in signature validation. The inconsistency between comments and code undermines reliability and may disrupt off-chain services that assume the documented behavior.

Recommendation

Introduce explicit handling for `referenceBlock == type(uint32).max` in all relevant lookup functions, bypassing the referenceBlock < block.number check and directly returning the latest stored values:

```
function _getTotalWeight(
    uint32 referenceBlock
) internal view returns (uint256) {
-   if (!(referenceBlock < block.number)) {
-       revert InvalidReferenceBlock();
-   }
-   return _totalWeightHistory.upperLookup(uint96(referenceBlock));
+   if (referenceBlock == type(uint32).max) {
+       return _totalWeightHistory.latest();
+   }
+   if (!(referenceBlock < block.number)) {
+       revert InvalidReferenceBlock();
+   }
+   return _totalWeightHistory.upperLookup(uint96(referenceBlock));
}
```

Apply the same fix to `_getOperatorForSigningKey`, `_getOperatorWeight`, `_getThresholdStake`, and `_getQuorum`.

Status

Resolved

[M-03] POAStakeRegistry#_updateOperatorSigningKey - Signing Key Front-Running (Impact Escalates Under H-01 Patch)

Description

Only registered operators may update signing keys; however, the update flow is front-runnable. A malicious operator can observe another operator's pending key update and submit a competing transaction first with the same newSigningKey. In the current implementation there is no "key already in use" check and the mapping is effectively last-write-wins, so the system does not immediately break. If the registry is later patched to enforce a strict one-to-one constraint (a signing key may be bound to only one operator, as in [H-01] Signing Key Hijacking can lead to AVS DOS), the front-run becomes impactful and can permanently block the rightful owner.

Vulnerability Details

`_updateOperatorSigningKey` accepts any `newSigningKey` and pushes it into both histories without uniqueness or pre-authorization checks. There is no guard that the key is unbound or pre-committed by the same operator.

```
function _updateOperatorSigningKey(address operator, address newSigningKey) internal {
                                            address           oldSigningKey           =
address(uint160(_operatorSigningKeyHistory[operator].latest()));
    if (newSigningKey == oldSigningKey) return;
    // No uniqueness check: any operator can claim any key first
                          _operatorSigningKeyHistory[operator].push(uint96(block.number),
uint160(newSigningKey));
                      _signingKeyOperatorHistory[newSigningKey].push(uint96(block.number),
uint160(operator));
    if (oldSigningKey != address(0)) {
        _signingKeyOperatorHistory[oldSigningKey].push(uint96(block.number), uint160(0));
    }
    emit SigningKeyUpdate(operator, block.number, newSigningKey, oldSigningKey);
}
```

Because transactions are ordered by miners/builders, a malicious operator can copy the victim's newSigningKey from the mempool and win the race. Today, a later transaction can overwrite the binding (last-write-wins), so the effect is transient. After an

`H-01-style` fix that enforces unique key→operator bindings, the first writer would permanently capture the key, preventing the intended operator from ever binding it.

Impact

In the current design, this front-running vulnerability is a minor issue, allowing a malicious operator to temporarily cause instability by racing to claim a key. However, if the uniqueness check from issue `H-01` is implemented, this issue becomes impactful. An attacker could then permanently capture a signing key by front-running the legitimate owner's transaction. This would permanently lock the victim operator out of the AVS, resulting in a targeted DOS.

Recommendation

The primary recommendation is to implement a proof-of-possession check alongside the `H-01` uniqueness enforcement. The contract should require the operator to provide a valid signature from the new signing key, proving they control it and making any front-running attack useless. As an alternative to proof-of-possession, a two-step commit-reveal scheme could be implemented, which would allow an operator to first commit to a key's hash before revealing the key itself in a later transaction, thus protecting it from being copied from the mempool.

Status

Resolved

Low

[L-01] POAStakeRegistry#_updateTotalWeight - Redundant History Push and Event When delta == 0

Description

`_updateTotalWeight` appends a new checkpoint and emits TotalWeightUpdated even when `delta == 0`. In those cases, `newTotalWeight == oldTotalWeight`, so the function records an identical value and produces a misleading update event. This wastes gas and pollutes on-chain history with no semantic change, complicating off-chain indexing and audits.

Recommendation

Short-circuit when no change occurs. Only push a new checkpoint and emit the event if `newTotalWeight != oldTotalWeight`. This preserves gas and keeps the history meaningful.

```
function _updateTotalWeight(
    int256 delta
) internal returns (uint256 oldTotalWeight, uint256 newTotalWeight) {
    oldTotalWeight = _totalWeightHistory.latest();
    int256 newWeight = int256(oldTotalWeight) + delta;
    newTotalWeight = uint256(newWeight);
+   if (newTotalWeight == oldTotalWeight) {
+       return (oldTotalWeight, newTotalWeight);
+   }
    _totalWeightHistory.push(uint96(block.number), uint160(newTotalWeight));
    emit TotalWeightUpdated(oldTotalWeight, newTotalWeight);
}
```

Status

Resolved

[L-01] POAStakeRegistry - Missing Upper Bound Check for thresholdWeight, updateOperatorWeight, and updateQuorum

Description

The contract does not enforce an upper bound check on the `thresholdWeight` parameter before casting it from `uint256` to `uint160` in the `_updateStakeThreshold` function. Similarly, the `weight` parameter in `updateOperatorWeight` and the `quorumNumerator` and `quorumDenominator` parameters in `updateQuorum` are not checked for exceeding the `uint160` maximum before being cast and stored. This omission could result in silent truncation if a value exceeding the `uint160` maximum is provided, leading to unexpected or incorrect storage of the threshold, operator weight, or quorum values.

```
function _updateStakeThreshold(
    uint256 thresholdWeight
) internal {
    _thresholdWeightHistory.push(uint96(block.number), uint160(thresholdWeight));
    emit ThresholdWeightUpdated(thresholdWeight);
}

function _updateOperatorWeight(address operator, uint256 weight) internal returns (int256) {
    // ...existing code...
    _operatorWeightHistory[operator].push(uint96(block.number), uint160(newWeight));
    // ...existing code...
}

function _updateQuorum(uint256 quorumNumerator, uint256 quorumDenominator) internal {
    // ...existing code...
    _quorumNumeratorHistory.push(uint96(block.number), uint160(quorumNumerator));
    _quorumDenominatorHistory.push(uint96(block.number), uint160(quorumDenominator));
    // ...existing code...
}
```

Recommendation

Add a require statement to ensure that `thresholdWeight`, `weight`, `quorumNumerator`, and `quorumDenominator` do not exceed the maximum value of `uint160` before casting and storing:

```
require(thresholdWeight <= type(uint160).max, "thresholdWeight exceeds uint160 max");

require(weight <= type(uint160).max, "weight exceeds uint160 max");

require(quorumNumerator <= type(uint160).max, "quorumNumerator exceeds uint160 max");

require(quorumDenominator <= type(uint160).max, "quorumDenominator exceeds uint160 max");
```

Status

Resolved

[L-03] POAStakeRegistry# - Missing Input Validation in POAStakeRegistry Contract

Description

The POAStakeRegistry contract is missing several critical input validation checks. Functions such as `updateStakeThreshold`, `registerOperator`, `deregisterOperator`, `updateOperatorWeight`, and `updateOperatorSigningKey` do not validate for zero-value inputs. This could lead to registering the zero address as an operator(inflating totaloperators count), setting a zero address as a signing key, or disabling the stake threshold.

Recommendation

It is recommended to validate inputs across the functions. Specifically, add checks to ensure that operator and signing key addresses are not `address(0)`, and that the thresholdWeight is greater than zero.

Status

Resolved

[L-04] POAStakeRegistry# - Use of floating pragma

Description

All the contracts use floating pragma version (`^0.8.27`). It's crucial to compile and deploy contracts with the same compiler version and settings used during development and testing. Fixing the pragma version prevents compilation with different versions. Using an outdated pragma version could introduce bugs that negatively affect the contract system, while newly released versions may have undiscovered security vulnerabilities.

Recommendation

Change the pragma statement in contracts to a fixed version, such as pragma solidity `0.8.27;`. This locks the contract to a specific compiler version.

Status

Resolved

QA

[Q-01] POAStakeRegistry - Redundant uint160 Type Conversion

Description

The code performs an unnecessary type conversion by wrapping a value that is already of type `uint160` with another `uint160` cast. This redundancy does not affect functionality but may reduce code clarity and could confuse future maintainers.

```
address oldSigningKey = address(uint160(_operatorSigningKeyHistory[operator].latest()));
```

Recommendation

Remove the redundant `uint160` conversion to improve code readability. The following change is suggested:

```
- address oldSigningKey = address(uint160(_operatorSigningKeyHistory[operator].latest()));
+ address oldSigningKey = address(_operatorSigningKeyHistory[operator].latest());
```

Status

Resolved

[Q-02] POAStakeRegistry#_getOperatorWeight - Misleading Parameter Naming

Description

The internal function `_getOperatorWeight` uses a parameter named signer, but in reality, this value represents the operator's address rather than the signer's key. This mismatch between the parameter name and its actual meaning can confuse developers and auditors, potentially leading to misinterpretation of the logic and integration mistakes in dependent code.

Recommendation

Rename the parameter to operator and update the `NatSpec` comments accordingly to ensure clarity and consistency. For example:

```
function _getOperatorWeight(
    address operator,
    uint32 referenceBlock
) internal view returns (uint256) {
    if (!(referenceBlock < block.number)) {
        revert InvalidReferenceBlock();
    }
    return _operatorWeightHistory[operator].upperLookup(uint96(referenceBlock));
}
```

Status

Resolved

Centralisation

The Lay3r Labs project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Lay3r Labs project seems to have a sound and well-tested code base,now that our vulnerability findings have been resolved. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au