ANATOMIST

SECURITY

Gimo Finance LSD Contracts

Security Assessment

September 20th, 2025 — Prepared by Anatomist Security

Ginoah Chu ginoah@anatomi.st

Celix Chen celix@anatomi.st

Table of Contents

1 —— Severity Level

CRITICAL	**Vulnerabilities enabling direct theft or irrecoverable financial loss.** • Direct loss of funds • Misconfigured authorization or access controls
HIGH	**Vulnerabilities causing significant financial or operational damage, but are more difficult to exploit.** • Loss of funds dependent on specific victim interactions • Exploitation requiring high capital relative to potential profit
MEDIUM	**Vulnerabilities that cause a recoverable DoS or extra fees/time.** • Exceeding Computational Limits • Partial data corruption that doesn't result in unrecoverable loss
LOW	**Issues with low impact or requiring specific conditions.** • Design oversights that do not threaten core operations • Minor race conditions unlikely to cause serious harm
INFO	**Opportunities for improvement with no immediate threat, typically addressing best practices or clarity.** • Aligning with coding standards or project conventions • Simplifying code to improve readability and maintainability

2 —— Scope

This assessment covered the lsd-contracts repository, pinned to commit `4b818c3`.

These components handle core functions of liquid staking, including user deposits and LSD minting, validator delegation/undelegation and rebalancing via stake pools, era-based reward accrual and exchange-rate computation, unbonding queues and withdrawals, protocol-fee accrual and fee-reserve management, and upgradeable access control across StakeManager, StakePool, and LsdToken (with associated bases and interfaces). All security assessment and analysis was conducted between *September 19, 2025* and *September 20, 2025*, using the specified commit hashes as reference points for code stability. Code modifications or commits beyond this timeframe were excluded from the scope of this audit.

3 —— Summary

Overall, we identified 6 findings. These findings are categorized into vulnerabilities and informational suggestions. Vulnerabilities present immediate security risks and should be remediated with high priority. Informational recommendations, while not posing immediate threats to system integrity, address potential security weaknesses that could lead to vulnerabilities if left unaddressed in future development cycles.

Informational Recommendations
Partial freeze of `reservedFee` in `undelegateMulti`
Missing validator address and length sanity checks in `addValidator` and `initialize`
Era-boundary stake race shares pending rewards
Public burn allows rate manipulation
`newEra` lacks reentrancy hardening against unknown validator behavior
Missing storage gap in UUPS Upgradeable implementation risks state corruption on upgrade

4 —— Informational Recommendations

4.1 Partial freeze of `reservedFee` in `undelegateMulti`

Description

During an undelegation, `undelegateMulti` calls `_govUndelegate` to retrieve the Ether required for unbonding. The call returns `(excessUnbondAmount, fee)`, where `excessUnbondAmount` is surplus that should be rolled into `pendingBond` for the next update cycle. However, the function also subtracts `excessUnbondAmount` from `reservedFee` in addition to subtracting the actual `fee`. This double debit causes `reservedFee` to be over-charged, forcing users to provide more funds for subsequent undelegations; the over-deducted portion is never consumed by any operation and effectively remains permanently frozen in the contract.

```
function undelegateMulti(address[] calldata _validators, uint256
    _amount) external override onlyStakeManager {
    ...

    for (
        uint256 i = (lastUndelegateIndex + 1) % _validators.
           length;
        totalCycle < _validators.length;
        (i = (i + 1) % _validators.length, ++totalCycle)
    ) {
        ...

        uint256 willUndelegate = needUndelegate < govDelegated ?
            needUndelegate : govDelegated;

        (uint256 excessUnbondAmount, uint256 fee) =
           _govUndelegate(val, willUndelegate);

        pendingBond += excessUnbondAmount;
        reservedFee -= fee;
        reservedFee -= excessUnbondAmount; // BUG: unintended
           subtraction

        ...
    }

    if (needUndelegate > 0) {
        revert NotEnoughAmountToUndelegate();
    }
}
```

Impact

The extra subtraction on `reservedFee` leads to over-charging of fees, requiring users to pre-fund more than necessary, and permanent freezing of the over-deducted amount since it can not used by any subsequent operation.

Remediation

Remove the unintended subtraction.

4.2 Missing validator address and length sanity checks in `addValidator` and `initialize`

Description

Although `addValidator` is restricted to the owner, misconfiguration can still cause `newEra()` failures (e.g., delegating to a non-validator or wrong contract) and operational issues.

```
function addValidator(address _poolAddress, address _validator)
    external onlyOwner {
     if (validatorsOf[_poolAddress].length() >=
        MAX_VALIDATORS_LEN) revert ValidatorsLenExceedLimit();
     if (!validatorsOf[_poolAddress].add(_validator)) revert
        ValidatorDuplicated();
}
```

Moreover, `initialize()` bypasses the validator-set size guard used elsewhere (`MAX_VALIDATORS_LEN`). It only checks non-empty input and then bulk-adds `_validators`, allowing an oversized set at deployment that violates the intended invariant.

```
function initialize(address _lsdToken, address _poolAddress,
    address[] memory _validators, uint256 _eraSeconds)
     external
     initializer
{
     _initOwner(msg.sender);
     _initManagerParams(_lsdToken, _poolAddress, 22, 0,
        _eraSeconds);

     minStakeAmount = 1e15;

     if (_validators.length == 0) {
         revert ValidatorsEmpty();
     }

     for (uint256 i = 0; i < _validators.length; ++i) {
         validatorsOf[_poolAddress].add(_validators[i]);
     }
}
```

Remediation

Validate the candidate address by probing the `IValidator` getters (e.g., `tokens()`, `delegatorShares()`, `withdrawalFeeInGwei()`, `commissionRate()`) under `try/catch`, and reject any address that reverts or returns out-of-policy values.

```
try IValidator(_validator).tokens() returns (uint256) {} catch {
     revert InvalidValidatorInterface(); }
```

Also, enforce the same limit during initialization:

```
if (_validators.length > MAX_VALIDATORS_LEN) revert
    ValidatorsLenExceedLimit();
```

4.3 Era-boundary stake race shares pending rewards

Description

A timing gap exists between validator reward accrual and the `newEra()` rate update. During this gap, an account can stake at the previous (outdated) exchange rate and still participate in the next era's repricing, capturing rewards it did not help earn.

```
function stakeWithPoolReceiver(address _poolAddress, address
    _receiver, string calldata _memo) public payable {
    uint256 stakeAmount = msg.value;
    ...
    uint256 lsdTokenAmount = (stakeAmount * EIGHTEEN_DECIMALS) /
        rate;
    ...
}

function newEra() external {
    ...
    uint256 newRate = _calRate(newTotalActive, ERC20(lsdToken).
       totalSupply());
    _setEraRate(_era, newRate);
    ...
}
```

Under current mainnet parameters (era $\approx$ 86,400 seconds; unbonding period $\approx$ 22 days; token/reward update cadence typically < 1 day), exploiting this requires locking capital for $\sim$22 days to capture at most $\sim$1 day of rewards. There is no economically viable profit opportunity and no practical grief/drain against other stakers. This is therefore classified as informational.

Remediation

Optionally enforce an era cutoff: queue deposits placed after reward accrual for minting at the post-`newEra` rate (i.e., apply new stakes effective next era).

4.4 Public burn allows rate manipulation

Description

The `LsdToken` inherits from `ERC20Burnable`, allowing any token holder to publicly burn their tokens, which directly affects the exchange rate calculation for subsequent eras. The exchange rate is calculated using the current total supply during `newEra()` execution, making it susceptible to manipulation through strategic burning.

```
contract LsdToken is ERC20Burnable, ILsdToken, IRateProvider {
    ... }

function newEra() external {
    ...
    uint256 newRate = _calRate(newTotalActive, ERC20(lsdToken).
        totalSupply());
    _setEraRate(_era, newRate);
    ...
}

function _calRate(uint256 _totalActive, uint256 _totalLst)
    internal view virtual returns (uint256) {
    ...
    uint256 calRate = (_totalActive * EIGHTEEN_DECIMALS) /
        _totalLst;
    ...
    return calRate;
}
```

Burning LSD immediately before `newEra()` lowers `totalSupply` and raises the next era's exchange rate, so later deposits mint fewer tokens per unit of underlying. This does not increase the burner's redeemable assets or capture previously accrued rewards; it merely redistributes value pro rata to remaining holders and alters pricing for new deposits. Accordingly, this is a pricing-mechanics artifact and is classified as informational.

Remediation

Override the `burn` and `burnFrom` functions in `LsdToken` to restrict public burning capa-

bilities. Allow token burning only through the `StakeManager` contract during legitimate unstaking operations to maintain proper exchange rate integrity.

4.5 `newEra` lacks reentrancy hardening against unknown validator behavior

---

Description

`newEra()` makes multiple external calls (`updateValidators`, `delegateMulti`, `undelegateMulti`, `getTotalDelegated`) per pool. Because it is not easily verifiable whether validators are uniformly restricted to the same behavior, we recommend adding additional hardenings such as reentrancy protection to guard against benign but unexpected behaviors.

```
function newEra() external {
    ...
    IStakePool(poolAddress).updateValidators(validators);
    ...
        IStakePool(poolAddress).delegateMulti(validators,
           needDelegate);
    ...
        IStakePool(poolAddress).undelegateMulti(validators,
           needUndelegate);
    ...
}
```

Remediation

Add a function-level guard by inheriting `ReentrancyGuardUpgradeable` and marking `newEra()` `nonReentrant` to prevent cross-function reentrancy during era transitions.

4.6 Missing storage gap in UUPS Upgradeable implementation risks state corruption on upgrade

---

Description

The contract uses the UUPS proxy pattern but omits the standard *storage gap* pattern in its upgradeable implementation. Without a reserved gap, any future change to the inheritance tree or the insertion of new state variables (especially via newly added parent contracts) can shift storage slots. When an upgraded implementation with a different layout is deployed, previously written state is reinterpreted under the new layout, causing silent corruption.

Even if child contracts only *append* variables, upgrading to a newer upstream library that adds storage to a base (e.g., switching `OwnableUpgradeable` or adding a new mixin) can shift slots without a gap. A dedicated gap allows safe insertion of new storage in parents while preserving the concrete layout of deployed children.

Remediation

Adopt one of the following patterns:

- **Add storage gaps** to every upgradeable contract that may gain parents or variables
- **Prefer namespaced storage (ERC-7201-style)** for complex systems to isolate layouts by namespace, reducing collision risk across upgrades