SHERLOCK
AUDIT ENGINE

SECURITY REVIEW · 2026

AUDIT ENGINE REPORT

Overlayer Audit Engine

AUDIT WINDOW
2026-11-04 → 2026-11-06

REPORT GENERATED
2026-11-06

ENGAGEMENT
Audit Engine

SCOPE

Overlayerfi/contracts-private	44cfb40556eae77ba7b12c75f99a75295a906dfd

SUMMARY

1	3	4
HIGH	LOW	INFO

Total accepted findings: 8

> **Important Note:** Unlike a traditional audit report, this Audit Engine report is a reflection of the sponsoring team's view on submissions, including the sponsoring team's judgment of the validity, severity, and scope of each reported submission. This report does not necessarily reflect Sherlock or Audit Engine's view on the validity, severity, and scope of any submission.

SEVERITY CLASSIFICATION

CRITICAL

Direct theft of funds without limitations of external conditions (outside the malicious actor's control), which can be triggered at any moment. The loss of the affected party must be significant, excluding unclaimed yield and rewards.

Guidelines for significant loss: users lose more than 20% *and* more than $100 of their principal.

HIGH

Direct loss of funds with considerable limitations of external conditions (outside the malicious actor's control), i.e., it requires specific but non-extreme conditions. The loss of the affected party must be considerable.

Guidelines for considerable loss:

- Users lose more than 1% *and* more than $10 of their principal.
- Users lose more than 1% *and* more than $10 of their yield.
- The protocol loses more than 1% *and* more than $10 of the fees.

MEDIUM

1. Causes a loss of funds that requires specific external conditions or protocol states outside a malicious actor's control, or where the loss is highly constrained. The loss must be relevant to the affected party.

2. Core contract functionality fails during intended use, making the contract unusable or causing a loss of funds relevant to the affected party.

Guidelines for relevant loss:

- Users lose more than 0.01% *and* more than $10 of their principal.
- Users lose more than 0.01% *and* more than $10 of their yield.
- The protocol loses more than 0.01% *and* more than $10 of the fees.

LOW

1. Causes a minor loss of funds or requires conditions or protocol states outside a malicious actor's control and far outside normal operation, or the loss is recoverable by admins.
2. Breaks contract functionality without a direct loss of funds, causing minor disruption of the protocol workflow or requiring admin intervention to resolve the issue.

Guidelines for minor loss:

- Users lose more than 0.001% *and* more than $1 of their principal.
- Users lose more than 0.001% *and* more than $1 of their yield.
- The protocol loses more than 0.001% *and* more than $1 of the fees.

INFO

An issue that does not reach Low.

CONTENTS

1 High 5

1.1 V4 cross-form round trips hide accrued yield from the vault entry che... 5

2 Low 7

2.1 Referral list growth disables even paginated pending-reward queries 7

2.2 Contract stakers without ERC721Receiver permanently lose their bonus... 9

2.3 Aave harvest failures roll back otherwise valid Merkl reward claims 11

3 Info 12

3.1 Third-party harvests can erase fractional referral and NFT bonuses 12

3.2 Empty-target harvests can maximize discarded base-reward rounding 14

3.3 Annual farm rewards are rounded down before they can accumulate ove... 15

3.4 Final Lido emergency exit can pay less than credited principal 17

Disclaimers 19

1 High

1.1 HIGH #7 V4 cross-form round trips hide accrued yield from the vault entry checkpoint

SUMMARY

On V4, an underlying redemption may withdraw principal plus harvestable yield and clamp `totalSuppliedCollateral` to zero when the wrap holds enough idle aTokens to support the burn. Reminting with the withdrawn underlying moves that value onto the wrap, while the remaining V4 buffer makes the vault's mandatory pre-entry compound observe no yield. A temporary staker can then resupply both collateral forms, recreate the old yield, and capture a share of it.

DESCRIPTION

`contracts/overlayerbacking/AaveHandlerV2.sol`, `_withdraw()`, lines 218-234, and `contracts/overlayerbacking/AaveHandler.sol`, `updateSuppliedAmounts()` and `_accountSupply()`, lines 327-367.

```
if (getUserSuppliedAssets(...) < amountCollateral_) revert ...;
(, uint256 assets) = spoke.withdraw(..., amountCollateral_, address(this));
updateSuppliedAmounts(assets);

if (collateralTaken_ > totalSuppliedCollateral) {
    totalSuppliedCollateral = 0;
}
```

The V4 withdrawal is limited by the full principal-plus-yield position, not by tracked principal. The wrap determines an underlying shortfall using only its idle underlying balance; idle aTokens backing other wrap supply do not reduce that request. Consequently, when idle aTokens exceed the harvestable yield, existing wrap supply can withdraw tracked principal plus all harvestable yield as underlying. The counter is then zero even though only the 100-unit V4 buffer remains. Reminting with the same underlying restores the burned wrap supply but leaves the economic surplus on the wrap, outside `_takeV4Yield()`. Later, `_accountSupply()` caps the two resupplies only in aggregate at wrap supply, so the same surplus becomes visible again after vault shares have been issued.

IMPACT

The attacker captures `X / (V + X)` of the staking allocation of the hidden yield, where `X` is temporary stake and `V` is prior vault assets. With enough flash liquidity, this approaches the entire vault allocation and removes the corresponding yield from existing stakers. For 100,000 units of accrued yield and a 50% staking allocation, an attacker controlling 90% of the temporarily enlarged vault captures about 45,000 units.

Limits: V4 must hold harvestable yield, the wrap must simultaneously hold idle aTokens at least equal to that yield, and V3 must redeem those aTokens exactly when they are later resupplied. The attacker also

needs enough temporary wrap liquidity and available mint/redeem capacity.

ATTACK PATH

Preconditions: V4 tracks principal `P`, its position is `P + Y + 100`, the wrap holds at least `Y` idle aTokens from ordinary aToken mints, and victim vault assets are `V`.

1. The attacker temporarily obtains `P + Y` wrap and redeems it for underlying. The wrap has no idle underlying, so it asks V4 for the full `P + Y` even though `Y` is yield.
2. V4 pays `P + Y`. `updateSuppliedAmounts` clamps tracked principal to zero, leaving only the 100-unit buffer in the Spoke.
3. The attacker remints `P + Y` wrap with the received underlying. Supply is restored, but the underlying and existing aTokens now sit on the wrap.
4. The attacker deposits temporary amount `X` into the staking vault. Its pre-deposit compound sees supplied assets equal to the 100-unit buffer and therefore harvests zero.
5. The attacker calls `supplyToBacking`. The underlying is supplied first; the idle aTokens are then redeemed on V3 and supplied to V4. `_accountSupply` records only `P + A` total principal against wrap supply, so `Y` becomes harvestable again.
6. The attacker compounds, withdraws their vault position, and repays the temporary wrap. They retain their share of the old yield, while prior stakers receive less.

RECOMMENDATION

On V4, never reduce tracked principal by more than the principal portion of a redemption. Before servicing an underlying withdrawal above `totalSuppliedCollateral`, reconcile or harvest the excess, and account for both collateral forms held by the wrap. More generally, make the vault's pre-action checkpoint measure aggregate surplus across the wrap and active venue so moving value between them cannot hide historical yield.

Trade-off: Aggregate reconciliation adds cross-contract reads and may require syncing wrap-held aTokens through V3 before vault actions, coupling those actions to V3 liquidity.

ASSUMPTIONS AND UNCERTAINTIES

This does not rely on a donation or on withdrawing aTokens from V4. The idle aTokens arise from the documented ability to mint wrap with either collateral form while `supplyToBacking` remains a separate permissionless step. The route is venue-specific because V4's underlying withdrawal includes yield and its fixed buffer makes the emptied counter appear fully checkpointed.

2 Low

2.1 LOW #2 Referral list growth disables even paginated pending-reward queries

SUMMARY

The referral reward view loads every referred address before applying its requested page. Untrusted accounts can enlarge that list without staking, making even single-member queries exceed ordinary RPC gas limits. Referrers lose access to the contract's pending-reward aggregation, although farming and withdrawals remain available.

DESCRIPTION

`contracts/liquidity/Liquidity.sol`, `pendingRewardsReferral()`, lines 992-1001.

```
address refSource = referral.referralCodes(code);
IOverlayerReferral.ReferralType type_ = referral.referralCodeTypes(code);
address[] memory referredUsers = referral.seeReferredByType(refSource, type_);
if (startIndex == 0 && endIndex == 0) {
    endIndex = referredUsers.length;
}
for (uint256 i = startIndex; i < endIndex; ++i) {
    total += pendingReward(pid, referredUsers[i]);
}
```

`seeReferredByType()` copies the complete storage array into memory and returns it. The caller then decodes that complete array. Consequently, `startIndex` and `endIndex` bound only the final summation. They never bound the expensive storage read or response allocation.

The configured `OverlayerReferral.consumeReferral()` appends each consumer permanently. A fresh account can consume a Ref code without depositing or earning points. No bound or removal mechanism limits this collection. The same aggregation function is inherited by the in-scope `EthLidoStake`.

IMPACT

A referrer's pending-reward aggregation becomes unavailable once the full list exceeds the caller's gas allowance. For example, 25,000 addresses require at least 52,500,000 gas in cold element reads alone. This exceeds a 50,000,000-gas RPC allowance before copying, decoding, or calculating any reward. Querying `[0,1)` still incurs that cost.

The list cannot be pruned, so reducing the requested page does not restore this interface. An indexer or individual `pendingReward()` calls are required as a workaround.

Limits: No principal or minted points are lost. Deposits, withdrawals, referral binding, and individual harvests remain available. The exact failure length depends on the RPC or transaction gas limit.

ATTACK PATH

Preconditions: `EthLidoStake` uses the documented referral contract, and an honest user owns a Ref code. No privileged action or principal deposit is required.

1. Numerous fresh accounts consume that Ref code through the existing permissionless referral interface.
2. Each successful consumption appends another address to the same permanent referred-user array.
3. With 25,000 entries, loading the array alone exceeds a 50,000,000-gas query allowance.
4. The referrer queries `pendingRewardsReferral(code, 0, 0, 1)` to inspect one member.
5. The full-array external call exhausts the allowance before the one-member summation can run.

RECOMMENDATION

Add a bounded referred-user getter accepting an offset and count, with a separate length getter. Update `pendingRewardsReferral()` to request only the selected page. Extend the referral interface and implementation accordingly.

Trade-off: The referral contract and interface need a compatible paginated accessor; existing full-list getters can remain available.

ASSUMPTIONS AND UNCERTAINTIES

This is a static-review finding; no code, tests, or proof of concept were executed. The 50,000,000-gas allowance is an illustrative RPC limit, not a verified deployment setting. The root cause is the in-scope aggregator's ineffective pagination; the referral contract supplies context. S22 confirms that Liquidity is in scope through EthLidoStake. KI-FARM-3/S176 concerns self-referral bonuses, not this availability failure.

2.2 LOW #8 Contract stakers without ERC721Receiver permanently lose their bonus NFTs

SUMMARY

A contract account can validly own and stake an NFT without implementing `IERC721Receiver`, because entry transfers the NFT to the farm. Both exit paths force `safeTransferFrom` back to `msg.sender` and offer no recipient choice. The callback check then always reverts, permanently locking that account's Origin or Special NFT in the immutable farm.

DESCRIPTION

`contracts/liquidity/Liquidity.sol`, `_stakeOriginNft()`, `_unstakeOriginNft()`, `_stakeWhitelistedNft()`, and `_unstakeWhitelistedNft()`, lines 408-457 and 483-533.

```
IERC721(collection).safeTransferFrom(msg.sender, address(this), tokenId);
...
IERC721(stake.collection).safeTransferFrom(address(this), msg.sender, stake.tokenId);
```

An ERC-721 contract checks `onERC721Received` on the recipient, not the sender. A contract that obtained an NFT through `transferFrom` or a non-safe mint can therefore approve the farm and stake it even if that contract has no receiver callback. On exit, however, the farm always makes that same contract the safe-transfer recipient. The ERC-721 callback check reverts, and neither unstake overload permits an alternate address or unsafe return transfer.

IMPACT

The affected account cannot recover the entire staked NFT. The record and NFT remain in the farm because the reverting transfer rolls back the preceding deletion. There is no owner rescue function and the farm is immutable.

Limits: EOAs and contracts implementing `IERC721Receiver` are unaffected. An upgradeable account may recover by adding the callback, but immutable accounts and accounts without an upgrade path remain locked. The NFT's market value is external to this repository.

ATTACK PATH

Preconditions: A contract account owns an eligible Origin or Special NFT but does not implement `onERC721Received`. It can execute calls and approvals, and obtained the NFT through an unsafe ERC-721 transfer or non-safe mint. No attacker is required.

1. The contract account approves the farm for its NFT.
2. It calls `stakeOriginNft` or `stakeWhitelistedNft`.
3. The NFT safely transfers to the farm, whose receiver callback succeeds; the sender needs no callback.
4. Later, the contract calls the corresponding unstake function.
5. The farm calls `safeTransferFrom(address(this), msg.sender, tokenId)`.
6. The NFT contract calls `onERC721Received` on the staker, which is absent, and reverts. Every retry has the same fixed recipient, so the NFT remains locked.

RECOMMENDATION

Add unstake overloads that accept a non-zero `recipient` and return the NFT to that address. Keep authorization tied to `originNftStaked[msg.sender]` or `whitelistedNftStaked[msg.sender]`, but call `safeTransferFrom(address(this), recipient, tokenId)`. Apply the same recipient support to both NFT types.

Trade-off: Users must protect the recipient argument from frontend mistakes; default no-argument overloads can remain for EOAs.

ASSUMPTIONS AND UNCERTAINTIES

Standard ERC-721 permits a contract without `IERC721Receiver` to own a token through `transferFrom`; this does not require non-standard token behavior. The disclosed Lido exit-recipient limitation concerns ETH queue claims and does not cover these independent ERC-721 custody paths.

2.3 LOW #11 Aave harvest failures roll back otherwise valid Merkl reward claims

SUMMARY

The V2 backing exposes Merkl claiming only through `compound(bool, claims)`, which always performs the Aave yield harvest afterward. A failed Aave harvest reverts the valid claim and forwarding, leaving Merkl rewards undistributed until the unrelated harvest failure is resolved.

DESCRIPTION

`compound(bool, MerkleClaim[])` (lines 177-180) enters `_compoundWithMerkle` (lines 237-244). That function first runs `_claimMerkle`, then unconditionally calls `_takeV4Yield` plus `_mintRewards` on V4 or `super._compound` on V3. The backing has no external function that calls `_claimMerkle` alone. If a V4 Spoke read or withdraw reverts while lenient mode is off, or if pending Aave yield cannot be minted or distributed because the wrap mint cap is exhausted or a downstream reward transfer reverts, EVM atomicity rolls back the Merkl claim and token forwarding too. The Merkl reward itself does not need any Aave withdrawal or wrap mint.

IMPACT

An otherwise claimable Merkl reward cannot be forwarded to its configured recipient while an Aave harvest failure persists. A time-limited campaign can expire before a keeper succeeds, or reward delivery can require an unrelated owner intervention.

Limits: The failure requires pending Aave yield or an unavailable V4 Spoke read, plus an independently failing harvest step. On V3, a keeper can choose `withdrawAave_ = false` to avoid a pure Pool withdrawal outage. An owner may restore minting, repair the downstream receiver, or enable lenient V4 mode; Merkl operators may also claim on the backing's behalf outside this call, though forwarding still needs a later route.

ATTACK PATH

Preconditions: The backing has a valid Merkl claim, and its Aave yield harvest currently reverts, for example because V4 withdrawals are paused while `lenientV4Harvest` is false or wrap minting is disabled with positive yield.

1. A keeper calls `compound(withdrawAave, claims)` with a valid claim proof.
2. `_claimMerkle` claims and forwards the reward.
3. `_takeV4Yield` or `_mintRewards` reverts for the unrelated Aave or wrap condition. The whole transaction, including the Merkl claim and forwarding, reverts.
4. Repeating the claim cannot succeed until the harvest condition changes, because there is no claim-only method.

RECOMMENDATION

Expose a non-reentrant Merkl claim-and-forward entry point independent of Aave compounding, and have the combined function call the same internal claim logic when both operations are desired.

Trade-off: Separating the operations adds an external surface and requires keepers to decide whether to claim, harvest, or do both; the existing recipient checks and balance-delta handling must be preserved.

ASSUMPTIONS AND UNCERTAINTIES

This is distinct from KI-SOW-1, which discloses that vault entry and exit calls depend on successful `compound`. The affected operation here is Merkl reward distribution, which is independent of vault share accounting. Merkl campaign expiry is an additional possible loss, not required for the availability failure.

3 Info

3.1 INFO #1 Third-party harvests can erase fractional referral and NFT bonuses

SUMMARY

Anyone can settle another account through `harvestFor` or `harvestAllFor`. Each settlement floors referral, self-referral, and NFT bonuses independently and discards every fractional remainder. Repeated third-party settlements therefore reduce the target's and referrer's cumulative OVERP bonuses.

DESCRIPTION

`contracts/liquidity/Liquidity.sol`, `harvestFor()`, `_harvestPool()`, and `_payBonusForType()`, lines 645-660 and 1032-1152.

```
function harvestFor(uint256 pid, address target) external override nonReentrant {
    _harvestPool(pid, target);
}

currentUser.rewardDebt = accumulated;
_payBonus(pending, pool.rewardAsset, user);

uint256 bonus = originalAmount.mulDiv(referralBonus_, 1000);
uint256 selfBonus = originalAmount.mulDiv(selfReferralBonus_, 1000);
```

The caller does not need authorization from `target`. `_harvestPool` permanently advances the target's `rewardDebt`, while every bonus uses a floor division and stores no remainder. The base reward reaches the target, but fractional bonus entitlement is lost instead of carrying into the next settlement. NFT bonuses have the same issue in `_bonusFromCollection()`.

IMPACT

A malicious caller can reduce referral and NFT rewards for any farmer and their referrers. For a 5% Team referrer bonus, twenty forced settlements of 19 wei each mint zero bonus, while one settlement of 380 wei mints 19 wei.

Limits: Each forced settlement loses less than one wei per bonus component. Calls cannot create new pending rewards within the same block, so the total economic loss is negligible under the documented rates and season length.

ATTACK PATH

Preconditions: A target has an active referral or NFT bonus and accrues base OVERP between blocks.

1. The target accrues 19 wei of base OVERP.
2. An attacker calls `harvestFor(pid, target)`. The target receives 19 wei, but a 5% Team bonus floors to zero.
3. The call advances the target's `rewardDebt`, discarding the fractional bonus remainder.
4. The attacker repeats this after each 19 wei interval twenty times.
5. The referrer receives zero instead of the 19 wei that one 380 wei settlement would mint. Other active bonus components lose their own rounding remainders similarly.

RECOMMENDATION

Track bonus entitlements cumulatively and subtract a per-user paid-bonus debt, or store each division remainder for the next settlement. Apply the same approach to referral, self-referral, Origin, OG, and Special NFT bonuses.

Trade-off: Preserving remainders adds storage writes and gas to every settlement.

ASSUMPTIONS AND UNCERTAINTIES

OVERP's future conversion value is outside this codebase. The loss is therefore rated Info and is distinct from the documented permissionless-harvest behavior, which only states that rewards go to the target.

3.2 INFO #3 Empty-target harvests can maximize discarded base-reward rounding

SUMMARY

`harvestFor` lets anyone checkpoint a pool for an address with no stake. Every checkpoint floors the reward-per-share increment and stores no division remainder, so an attacker can update each pool every block and permanently maximize the amount of base OVERP that is never allocated.

DESCRIPTION

`harvestFor` calls `_harvestPool`, which calls `updatePool` before reading the target's position; the target need not have a deposit. Both `Liquidity.updatePool` and `SingleStableStake.updatePool` add `rewards.mulDiv(1e18, stakedAssetSupply)` to the accumulator and then advance `lastRewardTime`. The remainder of `rewards * 1e18 / stakedAssetSupply` is discarded. One checkpoint over a long interval discards one remainder, whereas permissionless per-block checkpoints discard a new remainder every block. Calling for an empty target is sufficient and mints nothing to the caller.

IMPACT

The attack reduces base OVERP allocated to all stakers. Per checkpoint, aggregate loss is less than `stakedAssetSupply / 1e18` wei of OVERP. Under the live 18-decimal pools and rates this is economically negligible (for example, a one-million-token pool loses less than one million wei per checkpoint), so the issue is Info.

Limits: The attack requires one paid transaction per checkpoint, produces no attacker profit, and the documented live supply and rate ranges keep the aggregate loss tiny.

ATTACK PATH

Preconditions: A pool has nonzero stake, time is advancing, and its reward emission is nonzero.

1. In each block, an attacker calls `harvestFor(pid, attacker)` without depositing. 2. `updatePool` calculates only that block's reward, floors its reward-per-share increment, discards the remainder, and advances `lastRewardTime`. 3. Repeating this causes many rounding losses instead of the single rounding loss that a later organic checkpoint would incur.

RECOMMENDATION

Carry the accumulator division remainder per pool and add it to the next numerator before division. Alternatively, restrict third-party settlement calls to an authorized integration, although remainder accounting is the robust fix for ordinary frequent user actions too.

Trade-off: Remainder accounting adds one storage slot and write per pool update; restricting callers removes the documented keeper behavior.

ASSUMPTIONS AND UNCERTAINTIES

The attack requires repeated transactions and does not benefit the attacker economically. Its absolute impact depends on pool supply, block cadence, and reward parameters; with the documented live configuration it remains far below Low severity.

3.3 INFO #9 Annual farm rewards are rounded down before they can accumulate over time

SUMMARY

`SingleStableStake` converts each pool's annual TVL-based entitlement into a whole-wei per-second rate before multiplying by elapsed time. Fractional per-second rewards are discarded permanently even when a long interval would make them payable. Every staker receives slightly less than the configured annual rate.

DESCRIPTION

`contracts/liquidity/SingleStableStake.sol`, `rewardsForStakedAssets()` and `updatePool()`, lines 109-117 and 129-142.

```
uint256 rewards = multiplier * rewardsForStakedAssets(pool.stakedAsset, pool.rewardAsset) ...;

return stakedAmount.mulDiv(
    rewardsPerYearMultiplierNum[address(reward)],
    rewardsPerYearMultiplierDen[address(reward)]
) / SECONDS_IN_YEAR;
```

The annual entitlement is divided by `SECONDS_IN_YEAR` before `multiplier` contains the elapsed seconds. For an annual entitlement of `31,536,001` reward wei, the stored rate is one wei per second. A full year therefore allocates `31,536,000` wei instead of `31,536,001`. The correct interval calculation multiplies the annual entitlement by elapsed time before dividing by the year length.

IMPACT

All stakers are systematically underpaid relative to the configured annual multiplier. The discarded amount is less than one reward-token wei per elapsed second per pool, or less than `31,536,000` wei over a year.

Limits: OVERP has 18 decimals, so the maximum annual loss from this boundary is about `3.15e-11` OVERP per pool. It is economically negligible and rated Info.

ATTACK PATH

Preconditions: A pool has nonzero stake and its exact annual entitlement is not divisible by `31,536,000`. No attacker is required.

1. A pool's exact annual entitlement is `31,536,001` OVERP wei.
2. `rewardsForStakedAssets` divides it by `31,536,000` and returns one wei per second.
3. A full year elapses before `updatePool`.
4. `updatePool` multiplies the already-rounded rate by `31,536,000` seconds.
5. Stakers receive `31,536,000` wei; the remaining one wei is never carried into later intervals.

RECOMMENDATION

Calculate interval rewards before dividing by the year length. Use overflow-safe `mulDiv` to evaluate elapsed time against the annual entitlement, then apply allocation points. Carry any final division remainder between updates if exact frequency independence is required.

Trade-off: Combining the ratios requires careful overflow-safe ordering. Remainder storage adds one write per pool update.

ASSUMPTIONS AND UNCERTAINTIES

This is independent of allocation-point flooring: it exists with one pool whose allocation ratio is exactly 100%. It assumes the annual multiplier describes an entitlement that should accrue continuously, consistent with the function and parameter names.

3.4 INFO #10 Final Lido emergency exit can pay less than credited principal

SUMMARY

After a positive stETH rebase, the last emergency exit can transfer fewer shares than its credited principal requires. The user receives less stETH than their recorded stake, while a residual share becomes fee-collector surplus.

DESCRIPTION

`EthLidoStake.sol`, `_takeExitValue` (lines 281-292) sets `pay = amount` when the stETH balance covers principal and removes that full principal. `_transferStEthValue` (lines 294-307) transfers all shares only when `pay == stETH.balanceOf(this)`; otherwise it uses `getSharesByPooledEth(pay)`, which rounds shares down. `_skimSurplus` (lines 309-320) keeps the smallest share count covering principal, so the balance of those shares can exceed principal by one wei. In that state, the full-balance branch is skipped and the rounded-down share transfer can pay less than the principal just removed.

IMPACT

The last emergency-exiting staker can receive one smallest stETH unit less than their credited principal. For example, 85 shares can display 102 wei after a rebase against 101 wei of principal. An emergency exit transfers 84 shares displaying 100 wei, leaving one share for a later fee-collector skim. **Limits:** The ordinary shortfall is at most roughly one stETH share, around one or two wei at current share rates. This is below the Low monetary threshold and requires a rounding-sensitive rebase and emergency exit.

ATTACK PATH

Preconditions: A staker has 101 wei of credited principal backed by 85 shares. After a positive rebase, those shares display 102 wei at an illustrative 1.20 pooled-ETH-per-share rate. The fee collector is configured. No attacker is required.

1. The user calls `emergencyWithdraw` for their entire 101-wei principal.
2. `_skimSurplus` keeps all 85 shares because 84 shares display only 100 wei, below principal.
3. `_takeExitValue` sets `pay = 101` and reduces `totalPrincipal` to zero.
4. Since the farm balance displays 102, `_transferStEthValue` converts 101 wei to 84 shares and sends only 100 wei of displayed stETH to the user.
5. The remaining share is no longer needed for principal and can be skimmed to `feeCollector`.

RECOMMENDATION

On a final emergency exit, reconcile the actual `transferShares` payout against the principal being removed. After an enabled surplus skim, transfer the final backing share to the staker when rounding down would underpay credited principal. Alternatively, retain a fee-funded dust cushion for such exits. **Trade-off:** The final staker may receive up to one share of otherwise skimmable Lido yield, or the fee recipient must fund that dust.

ASSUMPTIONS AND UNCERTAINTIES

The example uses the interface's stated downward rounding for `getSharesByPooledEth` and an illustrative share rate. It differs from deposit-side undercredit: the user has already been credited 101 wei, and the

loss happens only at emergency payout. Normal queued withdrawals use a separate request path.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all related code and software is at the respective user’s sole risk and is the user’s responsibility.