Angles Security Review

Auditors

MiloTruck, Lead Security Researcher

R0bert, Lead Security Researcher

Sujith Somraaj, Security Researcher

T1moh, Associate Security Researcher

Report prepared by: Lucas Goiriz

February 6, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Angles is a staking platform that uses anS as a liquid ERC20 receipt token that is awarded to users when they stake S. The anS balance of such users steadily increases due to native network staking rewards from validator delegation, which are automatically distributed as regular rebases.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Angles according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 2 days in total, Angles engaged with Spearbit to review the angles protocol. In this period of time a total of **27** issues were found.

Summary

Project Name	Angles
Repository	angles
Commit	411f719d
Type of Project	Vault, Validators
Audit Timeline	Jan 13th to Jan 15th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	1	1	0
Low Risk	13	9	4
Gas Optimizations	0	0	0
Informational	12	8	4
Total	**27**	**19**	**8**

5 Findings

5.1 High Risk

5.1.1 Missing `nonReentrant` modifier on `addWithdrawalQueueLiquidity()` allows re-entrancy attacks

Severity: High Risk

Context: AnglesVault.sol#L675-L679, AnglesVault.sol#L593-L600, AnglesVault.sol#L625-L632.

Description: Unlike all other user-facing functions in the vault, `addWithdrawalQueueLiquidity()` does not have the `nonReentrant` modifier:

```
function addWithdrawalQueueLiquidity() external {
    // Stream any harvested rewards (S) that are available to the Vault
    _claimRewards();
    _addWithdrawalQueueLiquidity();
}
```

This can be exploited by re-entering `addWithdrawalQueueLiquidity()` in a callback when receiving ETH from `claimWithdrawal()`/`claimWithdrawals()`, as `SBalance` is only updated after ETH is transferred to the caller:

```
// Transfer S from the vault to the withdrawer
(bool withdrawn, ) = msg.sender.call{value: amount}("");
require(withdrawn, "Transfer Failed");

SBalance -= amount;

// Prevent insolvency
_postRedeem();
```

When this occurs, `_addWithdrawalQueueLiquidity()` will be called with an inflated `SBalance`, causing withdrawal accounting to break as the amount of S token allocated to withdrawals (i.e. `queue.claimable - queue.claimed`) will be more than the actual S token balance of the contract.

Recommendation: Consider adding the `nonReentrant` modifier to `addWithdrawalQueueLiquidity()`. Additionally, adhere to the Checks Effects Interactions pattern in `claimWithdrawal()`/`claimWithdrawals()` by updating `SBalance` before the ETH transfer:

```
+ SBalance -= totalAmount;

  // Transfer S from the vault to the withdrawer
  (bool withdrawn, ) = msg.sender.call{value: totalAmount}("");
  require(withdrawn, "Transfer Failed");

- SBalance -= totalAmount;
```

Angles: Fixed in commit dca81fd4 and ef95119.

Spearbit: Verified, the issue was fixed by updating `SBalance` before the external call and adding `nonReentrant` to `addWithdrawalQueueLiquidity()`.

5.2 Medium Risk

5.2.1 Missing index validation in `changeValidator()` and `deleteValidator()` functions lead to unintended changes if called incorrectly

Severity: Medium Risk

Context: AnglesVault.sol#L292, AnglesVault.sol#L314

Description: The functions for managing validators (`changeValidator` and `deleteValidator`) do not implement necessary bounds checking on the `index` parameter, which may result in unintended changes to the validator set.

```
function changeValidator(uint8 index, uint256 _validatorId, bool force) public onlyAdmin {
    // Missing index validation
    require(force || sfc.getStake(address(this), validatorsIndexed[index]) == 0, "Old validator has
    ↪  stake");
    // ...
    validatorsIndexed[index] = _validatorId;
}

function deleteValidator(uint8 index, bool force) public onlyAdmin {
    // Missing index validation
    require(force || sfc.getStake(address(this), validatorsIndexed[index]) == 0, "Validator has stake");
    // ...
    delete validatorsIndexed[numberOfValidators - 1];
    numberOfValidators--;
}
```

Impact:

- In `changeValidator`:
  - Allows adding validators at indices numberOfValidators.
  - Creates a mismatch between actual validators and numberOfValidators.
  - Bypasses proper validator registration process.
- In `deleteValidator`:
  - Out-of-bounds indices still execute the deletion.
  - Permanently removes the last validator regardless of the provided index.
  - Emits misleading events with incorrect data.

Recommendation:

1. Consider validating the `index` parameter in both functions:

```
function changeValidator(uint8 index, uint256 _validatorId, bool force) public onlyAdmin {
+     require(index < numberOfValidators, "Index out of bounds");
    // ... rest of the function
}

function deleteValidator(uint8 index, bool force) public onlyAdmin {
+     require(index < numberOfValidators, "Index out of bounds");
    // ... rest of the function
}
```

2. Otherwise, consider `validatorsIndexed` to an array or OpenZeppelin's `EnumerableSet` to simplify addition/removal operations. `numberOfValidators` can then be removed as the number of validators is equal to the array/set's length.

Angles: Fixed in commits 05fbdaea and c2d9a2f3.

Spearbit: Verified fix.

5.3 Low Risk

5.3.1 Missing validator status check in `deposit()` function across all execution paths

Severity: Low Risk

Context: AnglesVault.sol#L515

Description: The deposit function only validates validator status through SFC's delegation checks. However, these checks are bypassed if funds are allocated to the withdrawal queue instead of being delegated. Deposits continue when validators are slashed or offline until the admin toggles the `capitalDisabled` flag.

```
function deposit() public payable nonReentrant whenCapitalEnabled {
    // ... value checks ...

    _addWithdrawalQueueLiquidity();  // No validator checks
    _allocate();  // Validator checks only if delegating

    ans.mint(msg.sender, amount);
    SBalance += amount;
    emit Mint(msg.sender, amount);
}
```

Proof of Concept: This proof of concept shows how validator status is affected by `requestWithdrawal` before a user deposit. If the withdrawal queue is empty, the deposit fails; otherwise, it succeeds.

```
function test_e2e() external {
    // Initial deposits
    vm.startPrank(user);
    anglesVault.deposit{value: 1 ether}();

    vm.startPrank(user2);
    anglesVault.deposit{value: 1 ether}();

    // Validator 15 gets slashed
    vm.mockCallRevert(
        address(sfc),
        abi.encodeWithSelector(SFCMock.delegate.selector, 15),
        bytes("")
    );

    // Create withdrawal request, adding liquidity to queue
    anglesVault.requestWithdrawal(1e18);

    // New deposit succeeds despite slashed validator
    vm.startPrank(user3);
    anglesVault.deposit{value: 1 ether}();
}
```

Recommendation: Add validator status checks before accepting deposits:

```
function deposit() public payable nonReentrant whenCapitalEnabled {
    // Add upfront validator checks
    for (uint8 i = 0; i < numberOfValidators; i++) {
        require(
            sfc.getValidator[validatorsIndexed[i]].status == OK_STATUS
        ); // OK_STATUS = 0
    }

    // ... rest of the function
}
```

Angles: Fixed in commit 0ca70aab.

Spearbit: Verified fix. If the validator has stake, then the slashing status is checked before allowing user deposits.

5.3.2 Inconsistent validator state from transaction reordering in `deleteValidator` function

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `deleteValidator()` function in `AnglesVault.sol` allows the admin to delete a validator from the `validatorsIndexed` mapping. This action halts delegation to those validators for any future deposit transactions.

To remove a validator, the admin two deletion transactions provide the index of the validator from the mapping, not the validator ID. Therefore, if two deletion transactions are within the same block, changing their order will lead to two entirely different final states (validator sets) due to unsafe array manipulation practices.

For example, let's assume we have 4 validators, 15, 16, 17, and 18, at indices 0, 1, 2, and 3, respectively, and the admin submits two transactions:

```
Tx 1: deleteValidator(3, true)  // Expected: Remove validator 18

Tx 2: deleteValidator(0, true)  // Expected: Remove validator 15

Reordering them leads to different final validator arrays: [18, 16, 17, 0] vs [17, 16, 0, 0]
```

Recommendation:.

1. Consider passing in the `validatorId`, which is expected to be removed to avoid resulting in unexpected behavior.

```
function deleteValidator(uint8 index, uint256 validatorId, bool force) public onlyAdmin {
    /// ...
+    require(validatorsIndexed[numberOfValidators - 1] == validatorId);
    delete validatorsIndexed[numberOfValidators - 1];
    numberOfValidators--;
}
```

2. Alternatively, a batch delete function could be introduced to avoid running into re-ordering issue.

Angles: Acknowledged, thank you. We will not be deleting multiple validators in a single block.

Spearbit: Acknowledged.

5.3.3 Potential front-run of delegation migrated funds

Severity: Low Risk

Context: AnglesVault.sol#L409

Description: The `newRebalanceUndelegateOp` function can be used to migrate capital from one validator (call it the "old validator") to another ("new validator") in three operator-only steps:

1. Operator calls `newRebalanceUndelegateOp(_oldValidatorId, _amount)`, which undelegates the specified amount.
2. Operator calls `execRebalanceWithdrawOp(_oldValidatorId, _wrID, _safetyCheck, _doApplyToQueue=false)`, which withdraws the unstaked amount to `SBalance` without allocating it to the withdrawal queue.
3. Operator calls `delegateSingleFromStashOp(_newValidatorId, _amount)`, re-delegating those same funds from the vault's liquid stash to the new validator.

However, the final step can be front-run by a user who calls `requestWithdrawal(0)` (or another function that triggers `_addWithdrawalQueueLiquidity`). Because `_addWithdrawalQueueLiquidity` is called on `requestWithdrawal`, the newly released stash in `SBalance` would be "pulled" into the withdrawal queue before the operator re-delegates it, preventing a successful re-delegation to the new validator.

Recommendation: Ensure that the funds freed from `execRebalanceWithdrawOp` cannot be reallocated to the withdrawal queue until the operator completes the new delegation. A possible solution would be combining the last two steps of the flow(`execRebalanceWithdrawOp` and `delegateSingleFromStashOp` calls) into a single, uninterruptible function or transaction, so there is no window for users to front-run and pull the funds into the queue.

Angles: Acknowledged.

Spearbit: Acknowledged.

5.3.4 Lack of access control in multiple functions

Severity: Low Risk

Context: AnglesVault.sol#L671-L686

Description: Multiple external functions in the `AnglesVault` contract can be called by anyone, but they appear to have operator-level consequences. Specifically:

- `rebase()`:
  - Currently callable by any external account.
  - Triggers `_rebase`, which may mint tokens for performance fees, adjust total anS supply, etc...
- `addWithdrawalQueueLiquidity()`:
  - Anyone can call this to force `_claimRewards` from SFC pushing those funds into the withdrawal queue.
- `allocate()`:
  - Calls `_addWithdrawalQueueLiquidity` and `_allocate`. It can be used to fill the withdrawal queue with the `SBalance` available.

Recommendation: Consider adding the `onlyOperator` modifier to these functions.

Angles: Fixed in commit dca81fd4 by adding `onlyOperator` modifier.

Spearbit: Verified fix.

5.3.5 Wrong performance fee calculation if `feeCollector` is a rebasing account

Severity: Low Risk

Context: AnglesVault.sol#L867

Description: The `AnglesVault` contract calculates its "yield" as `vaultValue - tokenSupply`, then mints a fee as `yield * feeNom / feeDenom` to the `feeContainerAddress`:

```
// Performance fee collection
address _feeContainer = feeContainerAddress;
if (_feeContainer != address(0) && (vaultValue > tokenSupply)) {
    uint256 yield = vaultValue - tokenSupply;
    uint256 fee = (yield * feeNom) / feeDenom;
    require(yield > fee, "Fee must not be greater than yield");
    if (fee > 0) {
        ans.mint(_feeContainer, fee);
    }
    emit YieldDistribution(_feeContainer, yield, fee);
}
```

Let's imagine the following scenario where there is a single user deposited into the vault with 1e18 tokens and a single validator that distributes 10e18 as rewards. Operator then calls `claimSingle(15, true)` triggering a rebase:

- vaultValue = 11e18 (user deposit + accrued rewards)
- tokenSupply = 1e18
- yield = 11e18 - 1e18 = 10e18
- fee = 10e18 * 10 / 100 = 1e18

`1e18` anS tokens are minted to the `feeCollector` address.

Hence, `1e18` anS tokens are minted to the `feeCollector`. Before the `changeSupply` rebase occurs, the user's anS balance is `1e18` and `feeCollector` also holds 1e18. Effectively, 50% of the newly accrued 10e18 reward ended up going to the `feeCollector`, rather than the nominal 10%.

Recommendation: In order to mitigate this, `feeCollector` should call `anS.rebaseOptOut` so it's balance does not `_rebase()` automatically taking an extra cut from the users' rewards. Another valid solution would be rebasing up to `vaultValue - fee` first and minting the fee afterwards.

Angles: Acknowledged. Doing the same math, but with realistic rewards, the difference this makes would be very minimal & almost non-existent (10.0026% instead of the intended 10%). We are okay with the fee receiver to participate in rebase since it's a legitimate holder of anS.

Spearbit: Acknowledged.

5.3.6 `_totalValue()` will overestimate in case any validator is slashed in full

Severity: Low Risk

Context: AnglesVault.sol#L442

Description: There are 2 steps to rebalance validator: 1. Undelegate stake, 2. Withdraw stake. It uses `SInTransit` is used as a cached value between 2 steps.

```
// in case of rebalance we'll need to withdraw delegation & not lose vault value before actual claiming
↪  liquid S
uint256 public SInTransit;

function _newRebalanceUndelegateSingle(
    uint256 _validatorId,
    uint256 _amount
) internal {
    uint256 _wrID = _incrementWithdrawIdCounter();
    withdrawToAmount[_wrID] = _amount;
    SInTransit += _amount; // <<<
    sfc.undelegate(_validatorId, _wrID, _amount);
    // ...
}

function _execRebalanceWithdrawSingle(
    uint256 _validatorId,
    uint256 _wrID,
    bool _safetyCheck
) internal {
    uint256 _balanceBefore = address(this).balance;

    sfc.withdraw(_validatorId, _wrID);

    uint256 _received = address(this).balance - _balanceBefore;
    SInTransit -= withdrawToAmount[_wrID];
    if (_safetyCheck && !(_received >= withdrawToAmount[_wrID])) {
        revert("amount withdrawn less than requested");
    }
    SBalance += _received; // <<<
    // ...
}
```

Problem is that `sfc.withdraw()` reverts in case validator was slashed in full, code can be found here:

```
function _withdraw(address delegator, uint256 toValidatorID, uint256 wrID, address payable receiver)
↪  private {
    // ...

    uint256 amount = getWithdrawalRequest[delegator][toValidatorID][wrID].amount;
    bool isCheater = isSlashed(toValidatorID);
    uint256 penalty = getSlashingPenalty(amount, isCheater, slashingRefundRatio[toValidatorID]);
    delete getWithdrawalRequest[delegator][toValidatorID][wrID];

    if (amount <= penalty) {
        revert StakeIsFullySlashed(); // <<<
    }
    // ...
}
```

Because of revert 2nd step can't be completed. It means `SInTransit` contains non-existent value which can't be deducted. As a result `_totalValue()` overestimates result, and `_postRedeem()` won't flag an error.

Recommendation: Handle case when validator was slashed in full.

Angles: Fixed in commit 0ca70aab. Now, the protocol pauses if slashing is detected in any of the validators.

Spearbit: Verified fix.

5.3.7 Deposits and withdrawals will be blocked after big enough slashing occurs

Severity: Low Risk

Context: AnglesVault.sol#L849-L853

Description: `sfc.getStake()` returns staked amount and doesn't adjust it with slash penalty. It is used to calculate total value held in vault:

```
function _totalValue() public view returns (uint256 value) {
    WithdrawalQueueMetadata memory queue = withdrawalQueueMetadata;
    uint256 balance = SBalance + SInTransit + _getStakes(); // <<<
    if (balance + queue.claimed < queue.queued) {
        return 0;
    }
    return balance + queue.claimed - queue.queued;
}
```

Suppose following scenario:

1. Total staked = 100 ETH. `anS` supply is 100 ETH too. Each validator has 25 ETH staked.
2. 1 of 4 validators is slashed with 20%.
3. `_totalValue()` still returns 100 ETH:

   ```
   function _totalValue() public view returns (uint256 value) {
       WithdrawalQueueMetadata memory queue = withdrawalQueueMetadata;
       uint256 balance = SBalance + SInTransit + _getStakes(); // <<<
       if (balance + queue.claimed < queue.queued) {
           return 0;
       }
       return balance + queue.claimed - queue.queued;
   }
   ```

4. Bad validator is removed, it means functions `newRebalanceUndelegateOp()` and `execRebalanceWithdrawOp()` are called. Because of slashing it receives `25 ETH - 20% = 20 ETH`.

5. And now `_totalValue()` returns 95 ETH. There is fail fast check `_postRedeem()` which ensures that $anS supply doesn't deviate from `_totalValue()`, i.e. 100 ETH ± 3% in described scenario:

```
function _postRedeem() internal {
    uint256 totalUnits = 0;
    if (!rebasePaused) {
        totalUnits = _rebase();
    } else {
        totalUnits = _totalValue();
    }
    if (maxSupplyDiff > 0) {
        require(totalUnits > 0, "Too many outstanding requests");

        uint256 diff = ans.totalSupply().divPrecisely(totalUnits);
        require(
            (diff > 1e18 ? diff - 1e18 : 1e18 - diff) <= maxSupplyDiff, // <<<
            "Backing supply liquidity error"
        );
    }
}
```

As a result, deposits and withdrawals revert because of the check in `_postRedeem()`.

Recommendation: Refactor logic by handling potential slashing even though it has Low Likelihood.

Angles: Now the protocol pauses, if slashing is detected in any of the validators. See commit 0ca70aab.

Spearbit: Verified. Now user facing functions stop working in case any validator is slashed. In such situation admin intervention is required to delete slashed validator to unpause.

5.3.8 Missing `!rebasePaused` checks before rebasing

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `AnglesVault` contract, some functions call `_rebase()` without first verifying that `rebasePaused == false`. As a result, even when an administrator intends to pause rebasing, these calls can still trigger a supply rebase. Below are the specific occurrences:

- `restakeRewardsSingle(uint256 _validatorId, bool _withRebase)`:

```
function restakeRewardsSingle(
    uint256 _validatorId,
    bool _withRebase
) public onlyOperator nonReentrant {
    sfc.restakeRewards(_validatorId);
    if (_withRebase) {
        _rebase(); // <-- No check for !rebasePaused
    }
    emit SingleRewardsRestaked(_validatorId);
}
```

- `claimSingle(uint256 _validatorId, bool _withRebase)`:

```
function claimSingle(
    uint256 _validatorId,
    bool _withRebase
) public onlyOperator nonReentrant {
    _claimRewardsSingle(_validatorId);
    if (_withRebase) {
        _rebase(); // <-- No check for !rebasePaused
    }
    emit SingleRewardsClaimed(_validatorId);
}
```

- `rebase()`:

```
function rebase() external whenCapitalEnabled nonReentrant {
    _rebase(); // <-- No check for !rebasePaused
}
```

In contrast, other parts of the code do explicitly check `!rebasePaused` before rebasing (e.g., the deposit flow), ensuring a consistent pause behavior. The above lines can inadvertently rebase supply despite `rebasePaused` being true.

Recommendation: Consider adding `!rebasePaused` require checks to the suggested functions.

Angles: Fixed in commit acf60bbd by removing `rebasePaused` completely.

Spearbit: Verified fix.

5.3.9 Handle rewards in `deposit()` irrespective of `rebasePaused` flag

Severity: Low Risk

Context: AnglesVault.sol#L522

Description: The `deposit()` function of the `AnglesVault.sol` contract claims the pending rewards before proceeding with a user deposit. While the `rebasePaused` flag manages the rebase operation, the `_processRewards()` function must be called irrespective of the rebase state. Failing to do so will lead to inconsistent deposit behavior.

Recommendation: Consider separating the reward processing logic from the rebase condition.

```
function deposit() public payable nonReentrant whenCapitalEnabled {
    /// ... // rest of the code

    // Separate conditions for rewards and rebase
    if (_value > 0) {
         _processRewards();

        if (!rebasePaused) {
            _rebase();
        }
    }

    // ... // rest of the code
}
```

Angles: Fixed in commits acf60bbd and da2c9b5d by removing the rebase flag entirely from the code.

Spearbit: Verified fix.

5.3.10 `_processRewards()` misses out on pending rewards if `restakeRewards()` reverts for a validator

Severity: Low Risk

Context: AnglesVault.sol#L784-L790.

Description: `_restakeRewards()` calls `sfc.restakeRewards()` for each validator, wrapped in a try-catch:

```
function _restakeRewards() internal {
    for (uint8 i = 0; i < numberOfValidators; i++) {
        if (sfc.pendingRewards(address(this), validatorsIndexed[i]) != 0) {
            try sfc.restakeRewards(validatorsIndexed[i]) {} catch {}
        }
    }
}
```

Therefore, if `sfc.restakeRewards()` reverts for any reason, `_restakeRewards()` will not revert and rewards processing for that validator will be skipped. However, there are cases where the validator's pending rewards should be processed even when it is not possible to call `restakeRewards()`:

1. The validator's delegated stake limit is reached.
2. The validator is slashed (i.e. `status` is not `OK_STATUS`).

If `restakeRewards()` reverts for either of these reasons, `claimRewards()` can still and should be called to collect the validator's pending rewards. However, the current implementation does not do so, causing the vault to miss out on pending rewards.

Recommendation:. In `_processRewards()`, call `claimRewards()` instead of `restakeRewards()` for a validator when either of the conditions listed above are true.

Angles: Fixed in commit 83506233. Now, we check if there are pending rewards before calling `sfc.restakeRewards()` or `sfc.claimRewards()` instead of using the try-catch.

Spearbit: Verified, the issue is fixed as `_restakeRewards()` is no longer called when a validator is slashed. Additionally, the delegated stake limit is not expected to be reached easily.

5.3.11 Handling rewards with `_restakeRewards()` is inconsistent with other parts of the vault

Severity: Low Risk

Context: AnglesVault.sol#L757-L764, AnglesVault.sol#L696-L710, AnglesVault.sol#L716-L725.

Description: In `_processRewards()`, when `maxVaultValueReached` is set to `false`, `_restakeRewards()` is called to handle pending rewards in all validators:

```
function _processRewards() internal {
    if (maxVaultValueReached) {
        // Cannot restake, so we claim instead
        _claimRewards();
    } else {
        _restakeRewards();
    }
}
```

This will individually call `restakeRewards()` for each validator, which claims rewards and delegates them back to the same validator. However, redirecting the pending rewards of each validator back into itself breaks two of the vault's behaviors exhibited in other parts of the codebase.

1. The vault always prioritizes funds for withdrawals. Since un-delegating from validators can only be performed by the operator, any liquid S token in the vault will always be used to service withdrawals first before delegating to validators. This behavior can be seen in `_allocate()`, where `allocateAmouint` is calculated after subtracting the total amount of pending withdrawals:

```
uint256 sAvailableInVault = _sAvailable();
// No need to do anything if there isn't any S in the vault to allocate
if (sAvailableInVault == 0) return;

// Calculate the target buffer for the vault using the total supply
uint256 totalSupply = ans.totalSupply();
uint256 targetBuffer = totalSupply.mulTruncate(vaultBuffer);

// If available S in the Vault is below or equal the target buffer then there's nothing to
↪  allocate
if (sAvailableInVault <= targetBuffer) return;

// The amount of assets to allocate to the default strategy
uint256 allocateAmount = sAvailableInVault - targetBuffer;

_delegate(allocateAmount);
```

However, `_restakeRewards()` violates this as it immediately delegates all claimable rewards back into validators, regardless of whether there are any pending withdrawals in the vault.

2. Funds are split evenly amongst all validators. When delegating to validators in `_delegate()`, the amount delegated to each validator is the same:

```
uint256 chunk = amount / numberOfValidators;
uint256 lastChunk = amount - chunk * (numberOfValidators - 1);
if (chunk > 0) {
    for (uint8 i = 0; i < numberOfValidators - 1; i++) {
        sfc.delegate{value: chunk}(validatorsIndexed[i]);
    }
}
sfc.delegate{value: lastChunk}(
    validatorsIndexed[numberOfValidators - 1]
);
```

However, `_restakeRewards()` does not adhere to this behavior as the pending rewards for one validator could be higher than the others. If so, the amount of funds delegated back into that validator will be higher.

Note that the operator directly triggering `sfc.restakeRewards()` by calling `restakeRewardsSingle()` also breaks the two behaviors mentioned above.

Recommendation: Consider refactoring `_processRewards()` to only call `_claimRewards()`. Delegating into validators can be handled using `_allocate()` instead of `_restakeRewards()`.

Angles: Acknowledged. The vault does not always prioritize funds for withdrawals, that is only implemented for deposits. Validator rewards are meant to be restaked by design and that is how it works. In addition, APR and rewards are expected to be roughly identical, so the behavior you describe is acceptable.

Spearbit: Acknowledged.

5.3.12 Setting `rebasePaused = true` breaks the `_postRedeem()` check

Severity: Low Risk

Context: AnglesVault.sol#L840-L844, AnglesVault.sol#L849-L853

Description: When `rebasePaused` is set to `true` by the vault's admin, the vault will not rebase the `anS` token's total supply to match the vault's `_totalValue()`:

```
if (!rebasePaused) {
    totalUnits = _rebase();
} else {
    totalUnits = _totalValue();
}
```

However, if `anS.totalSupply()` is not rebased up to match `_totalValue()`, the `_postRedeem()` check that ensures solvency will eventually revert:

```
uint256 diff = ans.totalSupply().divPrecisely(totalUnits);
require(
    (diff > 1e18 ? diff - 1e18 : 1e18 - diff) <= maxSupplyDiff,
    "Backing supply liquidity error"
);
```

When pending rewards are collected from validators, `_totalValue()` increases. Since `totalSupply()` does not rebase to match `_totalValue()`, when a sufficient amount of rewards are collected, `1e18 - diff` eventually becomes smaller than `maxSupplyDiff`, eventually causing the check to fail. As a result, withdrawals from the vault will not be possible as `_postRedeem()` is called in all withdrawal-related functions.

Recommendation: Consider removing `rebasePaused` from the vault as there is no reason for rebasing to be disabled.

Angles: Fixed in commit acf60bbd.

Spearbit: Verified, `rebasePaused` has been removed.

5.3.13 Additional input validation for admin/operator functions

Severity: Low Risk

Context: AnglesVault.sol#L258-L262, AnglesVault.sol#L264-L268, AnglesVault.sol#L297-L300, AnglesVault.sol#L316-L319, AnglesVault.sol#L364-L369, AnglesVault.sol#L460-L470, AnglesVault.sol#L472-L482, AnglesVault.sol#L484-L494.

Description/Recommendation:

1. AnglesVault.sol#L258-L262 - `changeFeeNom()` and `changeFeeDenom()` should check that `feeNom` is not greater than `feeDenom` after the new values are set. This ensures that the vault fee cannot be configured to greater than 100%.
2. AnglesVault.sol#L297-L300, AnglesVault.sol#L316-L319 - `changeValidator()` and `deleteValidator()` should check that the pending rewards of the validator to be removed is also `0`. Otherwise, the vault will lose out on rewards if to a validator that still has pending rewards is removed:

   ```
   require(
   -     force || sfc.getStake(address(this), validatorsIndexed[index]) == 0,
   +     force || (
   +       sfc.getStake(address(this), validatorsIndexed[index]) == 0
   +       && sfc.pendingRewards(address(this), validatorsIndexed[index]) == 0
   +     )
         "Old validator has stake"
   );
   ```

3. AnglesVault.sol#L364-L369, AnglesVault.sol#L460-L470, AnglesVault.sol#L472-L482, AnglesVault.sol#L484-L494 - `newRebalanceUndelegateOp()`, `delegateSingleFromStashOp()`, `restakeRewardsSingle()` and `claimSingle()` do not check that `_validatorId` is in the `validatorsIndexed` mapping. Therefore, it is possible for the operator to perform operations (e.g. delegate) on behalf of the vault to validators that are not whitelisted by the admin. Consider ensuring `validatorsIndexed` contains `_validatorId` in these functions.

Angles: Fixed (1) in commit 53c2a5bc and (3) in commit 4a65c0ce. (2) is not an issue as the operator is allowed to delete validator with pending rewards. It's up to the operator to call claimSingle after the fact.

Spearbit: Verified. The recommended check has been added for (1), and (3) has been fixed by changing `validatorsIndexed` to an array.

5.4 Informational

5.4.1 `WithdrawalClaimed` is emitted twice in `claimWithdrawal()`

Severity: Informational

Context: AnglesVault.sol#L602

Description: The `WithdrawalClaimed` event is emitted twice during the withdrawal claim:

```
function claimWithdrawal(uint256 _requestId) external whenCapitalEnabled nonReentrant returns (uint256
↪   amount) {
    amount = _claimWithdrawal(_requestId);  // First emission inside this function
    // ...
    emit WithdrawalClaimed(msg.sender, _requestId, amount);  // Second emission
}

function _claimWithdrawal(uint256 requestId) internal returns (uint256 amount) {
    // ...
    emit WithdrawalClaimed(msg.sender, requestId, request.amount); // First emission
    return request.amount;
}
```

Recommendation: Consider removing the event emission from the external `claimWithdrawal` function.

Angles: Fixed in effbf724.

Spearbit: Verified fix.

5.4.2 Resolve all TODOs for production readiness

Severity: Informational

Context: AnglesVault.sol#L315, AnglesVault.sol#L584

Description: Multiple TODOs across the entire repository under review must be addressed, and all TODO comments must be removed/fixed. This will improve code quality, reduce technical debt, and implement all pending tasks correctly.

```
function deleteValidator(uint8 index, bool force) public onlyAdmin {
    // TODO: Remove force. Does not really make sense.
    // .... rest of the code
}

function claimWithdrawal(
  uint256 _requestId
) external whenCapitalEnabled nonReentrant returns (uint256 amount) {
   // ...
   // TODO: Replace with _claimRewards() ?
   // .... rest of the code
}
```

Recommendation: Please ensure all pending tasks are properly tracked and implemented.

Angles: Fixed in commits c4d8d8cb and da1ecb5a.

Spearbit: Verified fix. In `claimWithdrawal`, the TODO is removed without any changes. However, in `deleteValidator`, the force flag is completely removed, and additional checks are implemented to ensure that the validator being deleted has no funds in transit, no pending rewards, and a zero stake.

5.4.3 Code quality issues: documentation, license, typos and naming conventions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Several code quality issues and inconsistencies were found in the `AnglesVault.sol` contract:

- Licensing Issues: The AnglesVault contract is currently marked as UNLICENSED, as indicated by the SPDX license identifier at the top of the file:

  ```
  // SPDX-License-Identifier: UNLICENSED
  ```

  Using an unlicensed contract could result in legal uncertainties and conflicts regarding the code's usage, modification, and distribution rights. This may deter other developers from using or contributing to the project or even lead to legal issues in the future. Choosing and applying an appropriate open-source license (MIT, GNU, Apache License 2.0) to the smart contracts is recommended.

- Documentation Issues: Missing NatSpec comments for many functions. Inline documentation enhances code quality and understanding.
- Naming Convention: Public functions start with an underscore (_totalValue, _sAvailable). According to the solidity style guide, only internal functions should begin with an _.
- Typos:

  ```
  - No need to do anything is the withdrawal queue is full funded
  + No need to do anything if the withdrawal queue is fully funded

  - // cumulative total of all withdrawal requests included the ones that have already been claimed
  + // cumulative total of all withdrawal requests, including the ones that have already been
  ↪  claimed
  ```

Recommendation: Consider fixing the issues mentioned above to improve code quality and readability.

Angles: Fixed typos and grammar issues in commits fef4c2aad, 33894b9d..

Fixed the license issue in commits 653208fb and corrected the naming convention in 36e146ed.

Spearbit: Verified fix.

5.4.4 `maxVaultValue` can be slightly bypassed up to the amount of unclaimed yield

Severity: Informational

Context: AnglesVault.sol#L519

Description: There is a check in `deposit()` to enforce `maxVaultValue`:

```
function deposit() public payable nonReentrant whenCapitalEnabled {
    uint256 _value = _totalValue();
    uint256 amount = msg.value;
    require(amount >= minDeposit, "too small deposit");
    require(amount + _value <= maxVaultValue, "exceeds limit"); // <<<
    // ...
}

function _totalValue() public view returns (uint256 value) {
    WithdrawalQueueMetadata memory queue = withdrawalQueueMetadata;
    uint256 balance = SBalance + SInTransit + _getStakes(); // <<<
    if (balance + queue.claimed < queue.queued) {
        return 0;
    }
    return balance + queue.claimed - queue.queued;
}

function _getStakes() internal view returns (uint256 val) {
    for (uint8 i = 0; i < numberOfValidators; i++) {
        val += sfc.getStake(address(this), validatorsIndexed[i]); // <<<
    }
}
```

However `sfc.getStake()` only contains staked amount, and doesn't contain pending yield. As a result, total amount in the vault can be greater than `maxVaultValue`.

Recommendation: Check the `maxVaultValue` limit after calling `_processRewards()`:

```
  uint256 _value = _totalValue();
  uint256 amount = msg.value;
  require(amount >= minDeposit, "too small deposit");
- require(amount + _value <= maxVaultValue, "exceeds limit");
  // process rewards before rebase
  if (_value > 0 && !rebasePaused) {
      _processRewards();
      //rebase before minting new staked receipts
      _rebase();
  }
+ _value = _totalValue();
+ require(amount + _value <= maxVaultValue, "exceeds limit");
```

Angles: Acknowledged.

Spearbit: Acknowledged.

5.4.5 Smart contracts will not receive any yield by default

Severity: Informational

Context: AnglesVault.sol#L879

Description: By design, the `anS` contract treats all contract addresses as non-rebasing unless they explicitly call `rebaseOptIn()`. This includes multi-signature wallets. Consequently, if a multi-sig wallet deposits into the `AnglesVault` receiving anS tokens, its balance will not scale up during a rebase event. The multi-sigs will miss out any validator rewards unless they manually call `rebaseOptIn` to switch their account status to rebasing.

Recommendation: Warn that contract addresses, including multi-sigs, default to non-rebasing accounts. State clearly that `rebaseOptIn()` must be called in order to receive validator rewards. Provide frontend/backend integration that automatically calls `rebaseOptIn()` for any contract address executing a deposit.

Angles: Acknowledged. This is known behavior & how it works by design. There is a way contracts can opt-in

themselves or a governor can use administrative function to opt-in those who support rebases but can't opt-in themselves.

Spearbit: Acknowledged.

5.4.6 Missing zero amount check in requestWithdrawal() function

Severity: Informational

Context: AnglesVault.sol#L535

Description: In the `AnglesVault` contract, the `requestWithdrawal(uint256 _amount)` function does not enforce `_amount > 0`. Consequently, any user can call `requestWithdrawal(0)`, which will create a useless withdrawal request with zero tokens.

Recommendation: Consider adding the following require check to the `requestWithdrawal()` function:

```
```solidity
require(_amount > 0, "Zero withdrawal not allowed");
```
```

Angles: Fixed in commit 29346617.

Spearbit: Verified fix.

5.4.7 Redundant zero address check for `adminTempAddr` variable

Severity: Informational

Context: AnglesVault.sol#L212

Description: Inside the ownership transfer logic, there is a `require` statement like:

```
require(
    msg.sender == adminTempAddr && adminTempAddr != address(0),
    "Not allowed."
);
```

The second condition `adminTempAddr != address(0)` is redundant. Since `msg.sender` can never be `address(0)` in a valid transaction, checking that `adminTempAddr != address(0)` once you've already confirmed `msg.sender == adminTempAddr` is unnecessary.

Recommendation: Consider removing the redundant `adminTempAddr != address(0)` condition.

Angles: Fixed in commit 721dc56c by removing condition.

Spearbit: Verified fix.

5.4.8 Emit allocated buffer value in the `AssetAllocated` event

Severity: Informational

Context: AnglesVault.sol#L712

Description: The internal function `_allocate()` in the `AnglesVault.sol` contract delegates funds to validators. Before delegation, the function verifies that a sufficient buffer exists in the contract; any surplus is delegated.

After delegating, the function emits an event `AssetAllocated` with the amount delegated to the validators, missing out on the buffer allocated during the call, potentially leading to incomplete off-chain tracking.

Recommendation: Consider emitting the buffer allocated value in the `AssetAllocated` event.

```
event AssetAllocated(uint256 allocatedAmount, uint256 bufferAmount);
```

Angles: Fixed in commit 61efd196.

Spearbit: Verified fix.

5.4.9 Missing reward processing in `requestWithdrawal`

Severity: Informational

Context: AnglesVault.sol#L568

Description: The `requestWithdrawal()` function in the `AnglesVault.sol` contract fails to handle pending rewards before executing a withdrawal request. Unlike the `deposit()` and `claimWithdrawal()` functions, `requestWithdrawal()` does not call `_processRewards()`, which can affect the timely addition of liquidity to the withdrawal queue.

Recommendation: Consider calling `_processRewards()` to handle any pending rewards while requesting withdrawal.

```
function requestWithdrawal(
    uint256 _amount
)
    external
    whenCapitalEnabled
    nonReentrant
    returns (uint256 requestId, uint256 queued)
{
    // ...
    _processRewards();
    _addWithdrawalQueueLiquidity();
    // ...
}
```

Angles: Acknowledged. We can't do that as rebase will increase the user's balance and `_amount` as a parameter won't burn all anS and this will create an endless dust cycle, which we want to avoid, even forfeiting a minor portion of rewards for a user.

Spearbit: Acknowledged.

5.4.10 Inconsistent state handling in `claimOwnership()`

Severity: Informational

Context: AnglesVault.sol#L210-L218.

Description: In `claimOwnership()`, `adminTempAddr` is not reset to the zero address:

```
function claimOwnership() public nonReentrant {
    require(
        msg.sender == adminTempAddr && adminTempAddr != address(0),
        "Not allowed."
    );
    adminAddr = adminTempAddr;

    emit OwnershipTransferred(adminAddr);
}
```

This causes `adminTempAddr` to be inconsistent as it will be set to the new admin's address, even after ownership has been transferred. However, there is no impact apart from the new admin being able to repeatedly call `claimOwnership()`, even after becoming the admin.

Recommendation: Reset `adminTempAddr` to the zero address after `adminAddr` is set.

Angles: Fixed in commit 721dc56c.

Spearbit: Verified, the recommendation was implemented.

5.4.11 Minor issues with code

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Context: AnglesVault.sol#L839, AnglesVault.sol#L446

Description/Recommendation:

1. AnglesVault.sol#L839 - There's no need to initialize `totalUnits` as it is `0` by default:

```
- uint256 totalUnits = 0;
+ uint256 totalUnits;
```

2. AnglesVault.sol#L446 - Code can be simplified:

```
- if (_safetyCheck && !(_received >= withdrawToAmount[_wrID])) {
+ if (_safetyCheck && _received < withdrawToAmount[_wrID]) {
```

3. Many of the `public` functions in the contract can be changed to `external`.

Angles: Fixed in the following commits:

1. acf60bb - The recommendation is no longer applicable.
2. 686ab14 - Fixed as recommended.
3. da2c9b5 - Fixed as recommended.

Spearbit: Verified.

5.4.12 `_addWithdrawalQueueLiquidity()` should always be called In `claimWithdrawal()`/`claimWithdrawals()`

Severity: Informational

Context: AnglesVault.sol#L580-L589, AnglesVault.sol#L614-L617.

Description: In `claimWithdrawal()` and `claimWithdrawals()`, `_addWithdrawalQueueLiquidity()` is only called if `maxVaultValueReached` is set to `true`:

```
if (
    withdrawalRequests[_requestId].queued >
    withdrawalQueueMetadata.claimable
) {
    // TODO: Replace with _claimRewards() ?
    if (maxVaultValueReached) {
        _processRewards();
        _addWithdrawalQueueLiquidity();
    }
}
```

```
if (maxVaultValueReached) {
    _processRewards();
    _addWithdrawalQueueLiquidity();
}
```

However, `_addWithdrawalQueueLiquidity()` should be called even when `maxVaultValueReached` is `false` as withdrawal request are prioritized and the vault should allocate S token for withdrawals when possible.

Recommendation: In both functions, consider calling `_addWithdrawalQueueLiquidity()` outside the if-blocks:

```
  if (
      withdrawalRequests[_requestId].queued >
      withdrawalQueueMetadata.claimable
  ) {
      // TODO: Replace with _claimRewards() ?
      if (maxVaultValueReached) {
          _processRewards();
-         _addWithdrawalQueueLiquidity();
      }
+     _addWithdrawalQueueLiquidity();
  }
```

```
  if (maxVaultValueReached) {
      _processRewards();
-     _addWithdrawalQueueLiquidity();
  }
+ _addWithdrawalQueueLiquidity();
```

Angles: Acknowledged.

Spearbit: Acknowledged.