yAudit

April 2026

Prepared for
Alchemix v3

Audited by
Panda
HHK

Alchemix v3 - strategies

Smart Contract Security Assessment

Contents

1 Review Summary **2**

1.1 Protocol Overview 2
1.2 Audit Scope 2
1.3 Risk Assessment Framework 2
1.3.1 Severity Classification 3
1.4 Key Findings 3
1.5 Overall Assessment 3

2 Audit Overview **4**

2.1 Project Information 4
2.2 Audit Team 4
2.3 Audit Timeline 4
2.4 Audit Resources 4
2.5 Critical Findings 5
2.6 High Findings 5
2.7 Medium Findings 5
2.7.1 SiUSDStrategy can't dealocate when dealocation is queued 5
2.7.2 EtherfiEETHStrategy dealocation will always revert 5
2.8 Low Findings 6
2.8.1 Ether.fi direct deallocation can under-redeem due to rounding down weETH input 6
2.8.2 Ensure deallocate penalty is properly set for strategies with potential fees like Etherfi 7
2.8.3 OraclePricedSwapStrategy `previewAdjustedWithdraw` ignores Idle assets 8
2.8.4 OraclePricedSwapStrategy allocation floor uses mismatched units 8
2.9 Gas Savings Findings 9
2.10 Informational Findings 9
2.10.1 SFraxETHStrategy does not account for raw frxETH in position balance . 9
2.10.2 Router reverts on synthetic-only transmuter claims 10
2.10.3 FrxEthEthDualOracleAggregatorAdapter synthesizes timestamps, rendering OraclePricedSwapStrategy staleness checks ineffective 11

1 Review Summary

1.1 Protocol Overview

Alchemix V3 is a self-repaying DeFi lending protocol built around yield-bearing collateral and synthetic asset issuance.

1.2 Audit Scope

This audit covers 7 smart contracts totaling approximately 600 lines of code across 3 days of review.

```
src
├── FrxEthEthDualOracleAggregatorAdapter.sol
├── router
│   └── AlchemistRouter.sol
└── strategies
    ├── EtherfiEETHStrategy.sol
    ├── OraclePricedSwapStrategy.sol
    ├── SFraxETHStrategy.sol
    ├── SiUSDStrategy.sol
    └── WStethStrategy.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	4
Informational	3

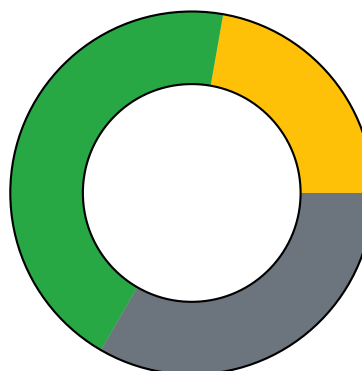

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

Well-structured codebase. All issues acknowledged and fixed.

2 Audit Overview

2.1 Project Information

Protocol Name: Alchemix v3
Repository: https://github.com/alchemix-finance/v3
Commit Hash: e506c27af72118a46d468e16695f7afb6d7ee052
Commit URL:
https://github.com/alchemix-finance/v3/tree/e506c27af72118a46d468e16695f7afb6d7ee052

2.2 Audit Team

Panda, HHK

2.3 Audit Timeline

The audit was conducted from April 15 to 17, 2026.

2.4 Audit Resources

Code repositories and documentation Additional resources: whitepaper, previous audits, etc.

Category	Mark	Description
Access Control	Good	Clear role separation with protected token functionality.
Mathematics	Average	Rounding and unit conversion issues found and fixed.
Complexity	Good	Multiple deallocation paths and external integrations.
Libraries	Good	Uses OpenZeppelin and standard interfaces appropriately.
Decentralization	Good	Privileged roles require governance oversight.
Code Stability	Good	All issues promptly acknowledged and fixed.
Documentation	Good	Well-commented with clear strategy explanations.
Monitoring	Good	Minor gaps in position balance tracking.
Testing and verification	Average	Core covered, edge cases around fees/rounding missed.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 SiUSDStrategy can't dealocate when dealocation is queued

Technical Details

`SiUSDStrategy._deallocate()` assumes the full exit completes in one call:

- unstake `siUSD` into `iUSD`
- redeem `iUSD` into `USDC`
- revert unless the full `USDC` amount is already available at the end of the transaction

InfiniFi's own docs say:

- Redemptions are processed through a queue
- They are only popped instantly if the protocol has enough liquid assets

When redemption is queued, partially filled, or loss-gated, `SiUSDStrategy._deallocate()` reverts because it requires the requested `USDC` to be fully redeemed.

Impact

Medium.

Recommendation

Reconsider the integration with the protocol, since immediate redemption isn't always possible, or add a swap `_deallocate(uint256 amount, bytes memory callData)` function.

Developer Response

Commit : 776898b7

2.7.2 EtherfiEETHStrategy dealocation will always revert

Technical Details

`EtherfiEETHStrategy._deallocate()` in `src/strategies/EtherfiEETHStrategy.sol` does the following on the direct path:

- Compute `shortfall = amount - idleBalance`
- Check `redemptionManager.canRedeem(shortfall, address(eETH))`
- Compute `weETHToRedeem = weETH.getWeETHByeETH(shortfall)`
- Redeem that amount through `redemptionManager.redeemWeEth(...)`
- Revert unless `ethReceived >= shortfall`

The problem is that the strategy sizes the redemption to the requested net WETH amount, but Ether. fi's instant redemption incurs a fee. With the current `0.30%` instant redemption fee, the direct path should revert for essentially any meaningful non-dust withdrawal that actually enters the redemption branch.

Impact

Medium. The strategy still has a swap-based exit path

Recommendation

Consider the etherfi fee when deallocating.

Developer Response

Fixed in e6bbf29.

2.8 Low Findings

2.8.1 Ether.fi direct deallocation can under-redeem due to rounding down weETH input

Technical Details

`EtherfiEETHMYTStrategy._deallocate()` computes the weETH amount to redeem using `weETH.getWeETHByeETH(shortfall)`, which rounds down. The rounded-down `weETHToRedeem` may correspond to an underlying eETH value slightly below the requested `shortfall`. When redeemed via `redemptionManager.redeemWeEth()`, the ETH received falls short by as little as 1 wei, causing `require(ethReceived >= shortfall)` to revert. Additionally, this calculation ignores Ether.fi's instant redemption fee (currently 0.3% via `exitFeeInBps`). Even with the rounding fixed, `redeemWeEth` charges a fee on output, so `ethReceived` will be less than the eETH value of the redeemed weETH. This means `ethReceived >= shortfall` can fail systematically — not just at rounding boundaries — whenever the shortfall is close to the full weETH position value.

Impact

Low. Direct deallocation requests revert on rounding boundaries and whenever the Ether.fi redemption fee makes the output insufficient. The strategy holds adequate value but cannot serve the withdrawal via this path. The allocator must fall back to the swap-based deallocation (`ActionType.swap`), causing a liveness disruption for the direct path.

Recommendation

Round up the weETH input and account for the redemption fee:

```
uint256 weETHToRedeem = weETH.getWeETHByeETH(shortfall);
if (weETH.getEETHByWeETH(weETHToRedeem) < shortfall) {
    weETHToRedeem += 1;
}
```

For the fee, consider over-redeeming by the fee percentage or removing the strict `ethReceived >= shortfall` check in favor of a post-wrap idle balance check (which already exists at line 99).

Developer Response

Fixed in this commit : e6bbf2

2.8.2 Ensure deallocate penalty is properly set for strategies with potential fees like Etherfi

Technical Details

The Morpho VaultV2 `forceDeallocatePenalty` mapping defaults to zero for all adapters. The `forceDeallocate()` function is permissionless — anyone can call it on any strategy for any amount.
When `forceDeallocate()` is called on the `EtherfiEETHStrategy`, it is forced to the `ActionType.direct` path which uses Ether.fi's instant redemption via `RedemptionManager.redeemWeEth()`. This incurs a 0.3% fee from Ether.fi (configurable by Ether.fi via `exitFeeInBps`).
If the curator does not explicitly set `forceDeallocatePenalty` for the Ether.fi adapter after deployment, any user can force-deallocate weETH positions and the 0.3% redemption fee is entirely socialized across all vault depositors. The caller bears no cost.
Additionally, if Ether.fi increases their `exitFeeInBps` in the future, the penalty may become insufficient even if initially configured correctly.

Impact

Low. Vault depositors absorb Ether.fi instant redemption fees when `forceDeallocate()` is called without an adequately calibrated penalty. Requires the curator to not set the penalty (or set it below Ether.fi's fee), and is limited by Ether.fi's rate-limiting (buffer liquidity must be above low watermark).

Recommendation

Ensure the `forceDeallocatePenalty` for the Ether.fi adapter is set to at least Ether.fi's current `exitFeeInBps` (currently 0.3%) before the strategy receives any allocations. Consider monitoring Ether.fi's `exitFeeInBps` and updating the penalty accordingly if it changes.

Developer Response

Thanks, we don't have a per strategy property for this but will log this in our strategy deployment set up (as noted by koala : currently using forceDeallocatePenalty of 2%). Acknowledged.

2.8.3 OraclePricedSwapStrategy `previewAdjustedWithdraw` ignores Idle assets

Technical Details

`OraclePricedSwapStrategy._totalValue()` includes both:

- idle vault assets already sitting on the strategy
- the oracle-priced wrapper position

But `OraclePricedSwapStrategy._previewAdjustedWithdraw()` only applies its preview math to the invested wrapper position:

- it computes `maxAsset = _oracleTokenToAsset(_positionBalance())`
- it does not include `_idleAssets()`

So the preview can understate immediately withdrawable funds.

Impact

Low.

Recommendation

Include idle vault assets in the preview path.

Developer Response

Fixed in a715d83.

2.8.4 OraclePricedSwapStrategy allocation floor uses mismatched units

Technical Details

`OraclePricedSwapStrategy._allocationSwapGuard()` in `src/strategies/OraclePricedSwapStrategy.sol` is documented as enforcing a minimum output floor "expressed in basis points of the asset amount in".
However, the implementation computes:

- `minAllocationOut = assetAmountIn * minAllocationOutBps / 10_000`
- and compares it directly to `oracleTokenReceived`

This compares two different unit systems:

- `assetAmountIn` is denominated in the vault asset, e.g. WETH.
- `oracleTokenReceived` is denominated in the received oracle token, e.g. `weETH`, `frxETH`, or `wstETH`.

No oracle conversion is performed before the comparison.

Impact

Low.

Recommendation

Compare values in the same unit system.

```
1 -        if (oracleTokenReceived < minAllocationOut) revert InvalidAmount(
  minAllocationOut, oracleTokenReceived);
2 +        uint256 allocationValueOut = _oracleTokenToAsset(oracleTokenReceived);
3 +        if (allocationValueOut < minAllocationOut) revert InvalidAmount(
  minAllocationOut, allocationValueOut);
```

Developer Response

Commit: 9a7721. Got it, since this was aimed as an independent floor guard for frxETH, updated the guard to be optional/overridable and did the floor implementation only for frxETH. Then the admin can update the floor as needed.

2.9 Gas Savings Findings

None.

2.10 Informational Findings

2.10.1 SFraxETHStrategy does not account for raw frxETH in position balance

Technical Details

`SFraxETHStrategy._positionBalance()` only counts sfrxETH holdings converted to frxETH-equivalent. Raw frxETH sitting on the contract is excluded from both `_positionBalance()` and `_totalValue()`.
During unwrap-and-swap deallocations, `_prepareIntermediateForSwap()` unwraps sfrxETH → frxETH and uses a balance delta to determine `sellAmount`, excluding any pre-existing frxETH. If the 0x swap doesn't consume the full `sellAmount`, leftover frxETH remains on the contract but is invisible to accounting. Since frxETH is marked as protected, it cannot be recovered via `rescueTokens()`.
In practice this requires the trusted allocator to supply calldata that partially consumes frxETH — an unlikely misconfiguration. Under normal operations, no frxETH accumulates outside of sfrxETH.

Impact

Informational. Any raw frxETH on the contract would be invisible to `realAssets()` and unrecoverable through normal flows. The precondition (trusted allocator error or donation) makes this largely theoretical.

Recommendation

Include frxETH in `_positionBalance()`:

```
function _positionBalance() internal view override returns (uint256) {
    return frxETH.balanceOf(address(this)) + sfrxETH.convertToAssets(sfrxETH.balanceOf(address(this)));
}
```

Developer Response

Fixed in 89ddb2.

2.10.2 Router reverts on synthetic-only transmuter claims

Technical Details

`AlchemistRouter._claimRedemption()` enforces `require(claimYield > 0, "No MYT to redeem")` after calling `ITransmuter.claimRedemption()`. This rejects redemption outcomes where the transmuter returns zero MYT shares but nonzero synthetic tokens (e.g., very small or early claims where `amountTransmuted` rounds to zero).
The router already has logic to forward `syntheticReturned` at lines 406-408, but the premature `claimYield > 0` check prevents it from executing for synthetic-only claims. The revert rolls back the transmuter's NFT burn, so no funds are lost — the user can call `Transmuter.claimRedemption()` directly instead.

Impact

Informational. The router cannot process a narrow class of valid transmuter claims where only synthetic tokens are returned. Users can bypass the router and claim directly from the transmuter.

Recommendation

Guard on either output being nonzero and skip the MYT redemption path when `claimYield == 0`:

```
- require(claimYield > 0, "No MYT to redeem");
+ require(claimYield > 0 || syntheticReturned > 0, "Nothing to claim");
```

Then wrap the MYT redeem/unwrap logic in an `if (claimYield > 0)` block.

Developer Response

Acknowledged. Will be documented that user can go through transmuter directly.

2.10.3 FrxEthEthDualOracleAggregatorAdapter synthesizes timestamps, rendering OraclePricedSwapStrategy staleness checks ineffective

Technical Details

`FrxEthEthDualOracleAggregatorAdapter.latestRoundData()` returns `updatedAt = block.timestamp` rather than forwarding a real publication time from the underlying Frax dual oracle:

```
1 return (uint80(block.number), int256(averagePrice), block.timestamp, block.timestamp,
  uint80(block.number));
```

`OraclePricedSwapStrategy._oracleAnswer()` enforces a `MAX_ORACLE_STALENESS` check against `updatedAt`:

```
1 require(updatedAt <= block.timestamp && block.timestamp - updatedAt <=
  MAX_ORACLE_STALENESS, "Stale oracle answer");
```

Since the adapter always returns the current `block.timestamp`, this staleness check will **never** trigger — it always evaluates to `0 <= MAX_ORACLE_STALENESS`. Any strategy using this adapter (i.e., `SFraxETHStrategy`) effectively has no oracle freshness protection. If the underlying Frax dual oracle returns a stale-but-not-flagged price (`isBadData == false`), the strategy will accept it unconditionally.
The strategy still relies on the dual oracle's own `isBadData` flag and the DEX swap slippage checks as secondary defenses, but the explicit staleness guard is rendered dead code for this oracle source.

Impact

Informational. The staleness check in `OraclePricedSwapStrategy` becomes a no-op for strategies using this adapter. The practical risk is mitigated by the dual oracle's own `isBadData` flag and swap-level slippage protection.

Recommendation

If the Frax dual oracle exposes no publication timestamp, document that the staleness check does not apply to this adapter.

Developer Response

Comment added on 2f7af34e.