Infinifi Contracts

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Slowfi, Security Researcher

July 17, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

infiniFi is a self-coordinated depositor-driven system designed to tackle the challenges of duration gaps in traditional banking.

From Jul 11th to Jul 14th the Cantina team conducted a review of infinifi-contracts on commit hash df916643 respectively. The team identified a total of **16** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	0	3
Low Risk	6	2	4
Gas Optimizations	0	0	0
Informational	7	1	6
Total	**16**	**3**	**13**

3 Findings

3.1 Medium Risk

3.1.1 Inaccurate wrapped yield token tracking

Severity: Medium Risk

Context: PendleV2Farm.sol#L147, PendleV2FarmV2.sol#L202

Description: In `wrapYieldTokenToPt`, the `PendleV2FarmV2` contract updates `totalWrappedYieldTokens` by adding the input amount `_yieldTokenIn`. However, the actual amount received after the swap, accounting for slippage, is calculated as `actualOut = ptToYieldToken(ptReceived)`, which is lower than `_yieldTokenIn` due t solippage. This discrepancy causes `totalWrappedYieldTokens` to overstate the true wrapped value by up to the `maxSlippage` threshold (0.3%).

For example, providing 1000(1e21) `_yieldTokenIn`, `actualOut` in the integration test is 9.997e20, as shown in the logs below:

```
emit DebugUint(a: "_yieldTokenIn", b: 1000000000000000000000 [1e21])
emit DebugUint(a: "actualOut", b: 999747383618229566197 [9.997e20])
```

This inflated state variable is fed into the interpolatingYield calculation:

```
uint256 totalWrappedAssets = yieldTokensToAssets(totalWrappedYieldTokens);
int256 totalYieldRemainingToInterpolate = int256(maturityAssetAmount) - int256(totalWrappedAssets) -
↪  int256(alreadyInterpolatedYield);
```

leading to overstated remaining yield. Pre-maturity, `assets()` includes this via `yieldTokensToAssets(totalWrappedYieldTokens) + interpolatingYield()`, overvaluing the farm.

The same pattern affects the `PendleV2Farm` contract in its `wrapAssetToPt` function, where `totalWrappedAssets += _assetsIn` despite slippage reducing effective value.

Recommendation: Track the effective post-slippage amount by updating `totalWrappedYieldTokens += actualOut` instead of `_yieldTokenIn`. Apply the same adjustment in `PendleV2Farm` for `totalWrappedAssets`.

infiniFi: Added a comment for it in commit 15069030. The farm does not report losses when investing in the PTs, as the yield generated towards maturity should make up for this, that's why before maturity it is only reported an increasing amount of assets.

Cantina Managed: Acknowledged and documented by infiniFi team.

3.1.2 Prematurity asset overvaluation from undiscounted slippages

Severity: Medium Risk

Context: PendleV2FarmV2.sol#L141-L159

Description: The `assets` function in the `PendleV2FarmV2` contract overstates value before maturity by not accounting for slippages in the full deployment path. When `block.timestamp < maturity`, it returns:

```
return supportedAssetBalance + yieldTokensToAssets(totalWrappedYieldTokens) + interpolatingYield();
```

Here, `supportedAssetBalance` includes USDC (`assetToken`) that must first be swapped to `yieldTokens` (e.g., sUSDe) via `signSwapOrder`, incurring up to 0.3% slippage (`maxSlippage = 0.997e18`). Then, those `yieldTokens` are wrapped to PTs via `wrapYieldTokenToPt`, adding another 0.3% slippage. The calculation treats these as fully deployable without loss, inflating the valuation.

On the other hand, `interpolatingYield` does apply `maturityPTDiscount` (0.998e18, 0.2%) for unwrap slippage:

```
maturityAssetAmount = maturityAssetAmount.mulWadDown(maturityPTDiscount);
```

but this only covers post-maturity unwrap, not the entry slippages. Unwrapping is 1:1 after maturity in Pendle, but prematurity valuation ignores the ~0.6% cumulative entry cost. After maturity, the only slippage we should account for is the conversion from `yieldToken` to USDC (if executed).

Recommendation: In the pre-maturity branch of the `PendleV2FarmV2.assets` function, discount `supportedAssetBalance` and `yieldTokensToAssets(totalWrappedYieldTokens)` by a cumulative entry slippage factor.

infiniFi: Code commented on commit 15069030 to highlight the uncounted fees.

Cantina Managed: Acknowledged and documented by infiniFi team.

3.1.3 Undiscounted swap fees in asset valuation in `EthenaFarm`

Severity: Medium Risk

Context: EthenaFarm.sol#L50-L67

Description: The `assets` function values all holdings (USDC, USDe, sUSDe) in USDC-equivalent terms using oracle prices, but doesn't apply any discount for swap fees or slippage when converting USDC → USDe → sUSDe and sUSDe → USDe → USDC:

```
uint256 usdcBalance = IERC20(_USDC).balanceOf(address(this));
uint256 usdcPrice = Accounting(accounting).price(_USDC);

uint256 usdeBalance = IERC20(_USDE).balanceOf(address(this));
uint256 usdePrice = Accounting(accounting).price(_USDE);

uint256 susdeBalance = IERC20(_SUSDE).balanceOf(address(this));
uint256 susdePrice = Accounting(accounting).price(_SUSDE);

// add USDe in the process of unstaking
usdeBalance += ISUSDe(_SUSDE).cooldowns(address(this)).underlyingAmount;

usdcBalance += usdeBalance.mulDivDown(usdePrice, usdcPrice);
usdcBalance += susdeBalance.mulDivDown(susdePrice, usdcPrice);

return usdcBalance;
```

Swaps between USDC ↔ USDe (or indirectly for sUSDe) via `signSwapOrder` incur up to 0.2% loss (`maxSlippage = 0.998e18`). Unstaking sUSDe to USDe is fee-free, but the subsequent USDe → USDC swap still applies. This overstates the farm's true liquid value, especially when holding large USDe/sUSDe balances.

For example, 1000 USDe valued at ~1000 USDC (assuming 1:1 peg) reports as 1000, but actual redemption after swap might yield only 998 USDC.

Recommendation: Discount non-USDC balances in `assets()` by `maxSlippage` when converting to USDC-equivalent (e.g., `usdcBalance += usdeBalance.mulDivDown(usdePrice, usdcPrice).mulWadDown(maxSlippage)`).

infiniFi: Code commented on commit 15069030 to highlight the uncounted fees.

Cantina Managed: Acknowledged and documented by infiniFi team.

3.2 Low Risk

3.2.1 Unaccounted direct PT token transfers

Severity: Low Risk

Context: PendleV2FarmV2.sol#L65-L66

Description: The `assetTokens` function defines supported assets as only `assetToken` (e.g., USDC) and `yieldToken` (e.g., sUSDe):

```
address[] memory tokens = new address[](2);
tokens[0] = assetToken;
tokens[1] = yieldToken;
return tokens;
```

PT tokens (e.g., PT-sUSDe) are not included and `isAssetSupported` would return `false` for them. However, if PTs are transferred directly to the farm, `IERC20(ptToken).balanceOf(address(this))` increases without updating `totalWrappedYieldTokens` or `totalReceivedPTs`.

Prematurity, `assets()` ignores this extra balance, as it relies on `totalWrappedYieldTokens` and `interpolatingYield`, which don't factor in unsolicited PTs. Post-maturity, `assets()` includes the full `balanceOfPTs`:

```
uint256 balanceOfPTs = IERC20(ptToken).balanceOf(address(this));
uint256 ptAssetsValue = ptToAssets(balanceOfPTs).mulWadDown(maturityPTDiscount);
```

This creates a discontinuity: understated value pre-maturity (yield interpolation misses extra PTs) and sudden inclusion post-maturity.

Recommendation: In `assets()` and `interpolatingYield`, calculate effective PT value from `balanceOfPTs` always, treating excess over `totalReceivedPTs` as a bonus (add to `maturityAssetAmount`).

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.2.2 Abrupt asset valuation changes from discount updates

Severity: Low Risk

Context: PendleV2FarmV2.sol#L119-L121

Description: The `setMaturityPTDiscount` function allows `PROTOCOL_PARAMETERS` role to adjust `maturityPTDiscount` at any time, including before maturity:

```
maturityPTDiscount = _maturityPTDiscount;
```

This discount (default 0.998e18, ~0.2%) factors into `interpolatingYield` for pre-maturity yield projection:

```
maturityAssetAmount = maturityAssetAmount.mulWadDown(maturityPTDiscount);
```

which feeds into `totalYieldRemainingToInterpolate` and the interpolated amount. `assets()` pre-maturity includes this via `+ interpolatingYield()`, so altering the discount causes an immediate revaluation of the farm's reported value.

Integration tests demonstrate the issue: After wrapping 1000e18 `yieldTokens` and warping 45 days (halfway to maturity), `assets()` returned value is 1188.032912. Setting `maturityPTDiscount` to 0.9e18 (10% discount) drops it to 1179.379584, a ~0.73% step decrease:

```
farm.assets() before -> 1188032912
>Call to setMaturityPTDiscount(0.9e18)<
farm.assets() after -> 1179379584
```

Such jumps disrupt protocol yield accounting in the `YieldSharingV2` contract and should minimized as much as possible.

Recommendation: Restrict `setMaturityPTDiscount` to only callable after maturity or when no PTs are held (`balanceOfPTs == 0`).

infiniFi: Added a comment to it in commit 15069030 to highlight this behavior.

Cantina Managed: Acknowledged and documented by infiniFi team.

3.2.3 Hardcoded ethena cooldown duration

Severity: Low Risk

Context: EthenaFarm.sol#L81-L83

Description: The maturity function in `EthenaFarm` hardcodes a 7-day cooldown:

```
return block.timestamp + 7 days;
```

This assumes a fixed unstaking period for sUSDe, but Ethena admins can update it via `setCooldownDuration`, restricted to `DEFAULT_ADMIN_ROLE` and capped at `MAX_COOLDOWN_DURATION` (90 days):

```
function setCooldownDuration(uint24 duration) external onlyRole(DEFAULT_ADMIN_ROLE) {
  if (duration > MAX_COOLDOWN_DURATION) {
    revert InvalidCooldown();
  }

  uint24 previousDuration = cooldownDuration;
  cooldownDuration = duration;
  emit CooldownDurationUpdated(previousDuration, cooldownDuration);
}
```

If Ethena increases the cooldown (e.g., to 14 days for risk management), the farm underreports maturity, treating positions as "matured" prematurely. This misleads allocation voting in the protocol, directing funds to the farm expecting quicker liquidity.

Recommendation: Replace the hardcoded value with a dynamic read: `return block.timestamp + ISUSDe(_SUSDE).cooldownDuration()`. Considering that you want to keep this farm illiquid with a minimum maturity of 7 days, the final implementation could look like this:

```
function maturity() public view virtual override returns (uint256) {
    uint256 ethenaCooldown = ISUSDe(_SUSDE).cooldownDuration();
    uint256 minMaturityDuration = 7 days;
    return block.timestamp + (ethenaCooldown > minMaturityDuration ? ethenaCooldown : minMaturityDuration);
}
```

infiniFi: Fixed in commit 12ef280b.

Cantina Managed: Fix verified.

3.2.4 Missing require check in withdrawSecondaryAsset function

Severity: Low Risk

Context: MultiAssetFarm.sol#L96-L108

Description: The `withdrawSecondaryAsset` function validates the asset with `isAssetSupported(_asset)`, which returns `true` for the primary `assetToken` since `assetTokens()` includes it. This allows calling `withdrawSecondaryAsset` with `_asset == assetToken`, withdrawing the primary asset via a direct `safeTransfer`:

```
require(isAssetSupported(_asset), InvalidAsset(_asset));

uint256 assetsBefore = assets();
IERC20(_asset).safeTransfer(_to, _amount);
uint256 assetsAfter = assets();

emit AssetsUpdated(block.timestamp, assetsBefore, assetsAfter);
```

The primary withdraw path uses `_withdraw` (also a `safeTransfer` in `MultiAssetFarm`) and emits with assumed `assetsBefore - amount`. While currently equivalent, this overlap could bypass subclass-specific logic if `_withdraw` is overridden in the future (e.g., for slippage checks or custom unwrapping in farms like `PendleV2Farm`). It also muddles intent, as secondary withdrawals are for non-primary assets.

Recommendation: Add a check in the `withdrawSecondaryAsset` function after the `isAssetSupported` call to ensure `_asset != assetToken`, reverting with `InvalidAsset` or a new error like `PrimaryAssetNotSecondary`. This enforces separation and future-proofs against overrides in `_withdraw`.

infiniFi: Fixed in commit 15069030.

Cantina Managed: Fix verified. The `withdrawSecondaryAsset` function now checks that `_asset != assetToken` preventing from withdrawing the primary asset.

3.2.5 Dangling dust approvals in `PendleV2FarmV2` contract

Severity: Low Risk

Context: PendleV2FarmV2.sol#L190-L191, PendleV2FarmV2.sol#L231-L232

Description: In `unwrapPtToYieldToken` and `wrapYieldTokenToPt`, the contract sets an approval for `pendleRouter` using `forceApprove`, but doesn't reset it to zero afterward:

```
IERC20(ptToken).forceApprove(pendleRouter, _ptTokensIn);
```

followed by `pendleRouter.call(_calldata)`. This can leave a non-zero approval indefinitely if the call doesn't consume the full amount. The same occurs in `wrapYieldTokenToPt` with `IERC20(yieldToken).forceApprove(pendleRouter, _yieldTokenIn)`.

If `pendleRouter` is compromised or behaves maliciously, it could drain the farm's PT or `yieldToken` balances via a `transferFrom` call. While `pendleRouter` is trusted, this violates least-privilege by maintaining open-ended access.

Recommendation: After the `pendleRouter.call`, explicitly reset approval to zero with `IERC20(token).forceApprove(pendleRouter, 0)` in both functions to limit exposure to the current transaction.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.2.6 Potential Denial of Service on redeems due to pending losses

Severity: Low Risk

Context: PendleInfiniFisiUSD.sol#L60-L80

Description: In the InfiniFi protocol, redemption operations, such as those handled via `InfiniFiGatewayV2.redeem` and `LockingController.withdraw` (invoked indirectly through `PendleInfiniFisiUSD` redemptions to USDC or iUSD), include a check in `_revertIfThereAreUnaccruedLosses` that reverts if `YieldSharingV2.unaccruedYield()` returns a negative value, indicating pending losses from farms. Similarly, `StakedToken.maxRedeem` and `maxWithdraw` revert under the same condition to prevent users from exiting positions before losses are propagated, ensuring fair loss distribution across locking users, staked holders and general iUSD holders.

The `unaccruedYield` function calculates the difference between the protocol's total asset value (converted to receipt tokens like iUSD) and the total supply of receipt tokens, where a negative result signifies unrealized losses. To resolve this and enable redemptions, the `YieldSharingV2.accrue` function must be called, which applies the yield (positive or negative) by minting/burning tokens or slashing positions as needed. In Pendle integrations, redeeming SY shares to USDC involves unstaking siUSD to iUSD via the gateway and then redeeming iUSD to USDC, inheriting these revert conditions and requiring accrual if losses are pending.

Relevant code from `InfiniFiGatewayV2._revertIfThereAreUnaccruedLosses`:

```
require(yieldSharing.unaccruedYield() >= 0, PendingLossesUnapplied());
```

And from `YieldSharingV2.accrue`:

```
int256 yield = unaccruedYield();
if (yield > 0) _handlePositiveYield(uint256(yield));
else if (yield < 0) _handleNegativeYield(uint256(-yield));

emit YieldAccrued(block.timestamp, yield);
```

The `accrue` function is permissionless, allowing any user to call it and apply pending yields. For negative yields, it sequentially slashes locking positions (via `LockingController.applyLosses`), staked tokens (via `StakedToken.applyLosses`) and finally adjusts the receipt token oracle price if necessary. This ensures losses are applied fairly but requires an extra transaction during loss events. While this mechanism prevents users from redeeming at inflated values before losses are realized, it introduces a temporary barrier to redemptions until accrue is invoked. In the context of `PendleInfiniFisiUSD`, where redemptions to USDC or iUSD rely on InfiniFi's gateway and staked token logic, this could manifest as a short-term denial-of-service (DoS), forcing users to wait for accrual, performed by themselves or another party, before completing the transaction.

Recommendation: Given that calling `YieldSharingV2.accrue` presents high gas costs, cosinder adding an informative revert error so users are aware that the revert is temporary until the losses are realized within Infinifi.

Infinifi team: Will not fix. Redeem will revert if losses were not applied but do not want to increase gas cost on the redemption.

Cantina Managed: Acknowledged by infiniFi team.

3.3 Informational

3.3.1 Asset valuation jump on early unwrap

Severity: Informational

Context: PendleV2FarmV2.sol#L219-L227

Description: In `unwrapPtToYieldToken`, the `MANUAL_REBALANCER` role can exit PT positions early (before maturity), but must unwrap the full `totalReceivedPTs` amount. This resets tracking variables like `totalWrappedYieldTokens = 0` and `alreadyInterpolatedYield = 0`. Pre-unwrap, `assets()` (when `block.timestamp < maturity`) uses a conservative valuation: `supportedAssetBalance + yieldTokensToAssets(totalWrappedYieldTokens) + interpolatingYield()`, where `interpolatingYield` discounts the projected maturity value by `maturityPTDiscount` (0.998e18) for expected unwrap slippage.

Post-unwrap, the farm holds `yieldTokens` (e.g., sUSDe), valued directly via oracle prices without that discount. Integration tests showed this caused a small step increase in the reported `assets()` value:

```
farm.assets() before -> 1188.032912
>Call to farm.unwrapPtToYieldToken(119897002182381834879, _PENDLE_ROUTER_CALLDATA_9)<
farm.assets() after  -> 1188.905564
```

The ~0.07% jump (from 1188.03 to 1188.91) reflects recovering more value than the discounted interpolation assumed, as actual early unwrap slippage may be less than anticipated. Impact is minor: sudden "yield spike" in the total protocol yield accounting.

Recommendation: Document this behavior in comments or protocol docs as expected due to conservative pre-maturity discounting.

infiniFi: Documented in commit 15069030.

Cantina Managed: Fix verified. Code commented as suggested.

3.3.2 Unchecked deposit cap in secondary movements

Severity: Informational

Context: ManualRebalancer.sol#L134-L161

Description: The `_singleMovementSecondaryAsset` function withdraws secondary assets from the source farm and transfers them directly to the destination farm address, then calls `IFarm(_to).deposit()` without first checking the destination's `maxDeposit()`:

```
if (_asset == _fromAssetToken) {
    IFarm(_from).withdraw(_amount, _to);
} else {
    MultiAssetFarm(_from).withdrawSecondaryAsset(_asset, _amount, _to);
}

// trigger deposit in destination farm
IFarm(_to).deposit();
```

In contrast, the primary asset path in `_singleMovement` caps `_amount` to `maxDeposit` before withdrawing:

```
uint256 maxDeposit = IFarm(_to).maxDeposit();
_amount = _amount > maxDeposit ? maxDeposit : _amount;
```

Without this check for secondary assets, if the transfer would exceed the cap, the subsequent `deposit()` reverts (as `Farm.deposit()` enforces cap), causing the entire transaction to revert. This includes `batchMovement`, where one over-cap secondary move reverts the whole batch.

Recommendation: Mirror the primary path by querying `maxDeposit()` before the transfer and capping `_amount` accordingly, allowing partial movements and avoiding unnecessary reverts.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.3.3 Unused custom error definition

Severity: Informational

Context: ManualRebalancer.sol#L14

Description: The `ManualRebalancer` contract defines a custom error `InactiveRebalancer`:

```
error InactiveRebalancer();
```

but it is never thrown or referenced anywhere in the codebase.

Recommendation: Remove the unused error definition to streamline the contract and reduce deployment costs.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.3.4 Potential redemption failures if low liquid USDC in infiniFi

Severity: Informational

Context: SYBaseUpg.sol#L67-L68

Description: In the protocol's redemption flow, the `InfiniFiGatewayV2.redeem` function includes a strict minimum output check: `require(assetsOut >= _minAssetsOut, MinAssetsOutError(_minAssetsOut, assetsOut));`, where `_minAssetsOut` is set to the full expected USDC amount based on the iUSD input (computed via `RedeemController.receiptToAsset`). The underlying `RedeemController.redeem` supports partial fulfillment by sending immediate liquidity and queueing the remainder in `userPendingClaims` for later manual claim.

If liquidity is insufficient (or the redemption queue is not empty), `redeemController.redeem` returns only the immediate `assetsOut` (potentially 0 or partial).

In the Pendle integration (`PendleInfiniFisiUSD._redeem` for USDC out), the call to `gateway.redeem(receiver, receiptOut, assetsOut)` (with `assetsOut` as full expected) reverts if only a partial amount of USDC is redeemed, reverting always in the following check:

```
// InfiniFiGatewayV2.redeem
uint256 assetsOut = redeemController.redeem(_to, _amount);
require(assetsOut >= _minAssetsOut, MinAssetsOutError(_minAssetsOut, assetsOut));  // Reverts on partial
```

This design prioritizes atomicity (full or nothing) but at the cost of availability. Therefore, if there is not too much USDC on liquid farms that can be pulled and the amount of USDC redeemed is high, a partial redemption with its respective revert might be triggered.

Example scenario:

1. Assume low liquidity in RedeemController: `liquidity() = 500 USDC`, queue empty.
2. User/Pendle calls `gateway.redeem(receiver, 1000 iUSD, 1000 USDC min-out)` (full expected from preview).
3. `redeemController.redeem`: Computes `assetAmountOut=1000`, but available=500 → sends 500 USDC, enqueues 500 equivalent iUSD.
4. Returns `assetsOut=500` → gateway checks `500 >= 1000` → reverts with `MinAssetsOutError`.
5. Entire tx (including Pendle unstake/redeem) reverts → no redemption, no queue entry.

Recommendation: Merely informative. If this happens user can always perform a smaller `redeem` call pulling USDC and then pull the rest as iUSDC.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.3.5 Inefficient capital deployment due to cooldown between CowSwap orders

Severity: Informational

Context: CoWSwapFarmBase.sol#L31, CoWSwapFarmBase.sol#L51-L69

Description: In the current design of CowSwap-integrated farms-such as `EthenaFarm`, `PendleV2FarmV2`, and any others inheriting from `CoWSwapFarmBase`-USDC is converted into sUSDe via CowSwap limit orders. This process is regulated by a cooldown constraint enforced through the `_SIGN_COOLDOWN` constant, which defines a global minimum interval between consecutive order signings.

The core logic resides in the `_checkSwapApproveAndSignOrder` function within `CoWSwapFarmBase`, and its behavior affects all derived contracts. When an order is signed, the `lastSignTime` is updated and subsequent attempts to sign a new order must wait until the cooldown period elapses.

This design introduces inefficiencies in capital utilization. For example, when a first deposit is received and swapped, any additional deposits arriving during the cooldown window must wait idle, without generating yield. In periods of frequent deposits or volatile user behavior, this can lead to multiple periods of unproductive capital across all CowSwap-based farms.

Recommendation: Consider to make the cooldown duration dynamic based on current conditions or remove the constraint altogether to avoid delays in capital deployment.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.3.6 `CowSwap` approval logic may overwrite allowances in future extensions

Severity: Informational

Context: CoWSwapFarmBase.sol#L67, EthenaFarm.sol#L87, PendleV2FarmV2.sol#L190, PendleV2FarmV2.sol#L231

Description: The CowSwap-integrated farms rely on `forceApprove` to authorize token transfers by the CowSwap relayer. This approach unconditionally sets the approval amount, overwriting any existing allowance. While this is not currently an issue in the code, as approvals are not interleaved or shared across multiple flows, it could become problematic if the cooldown restriction is removed or relaxed and multiple approvals are performed more frequently or concurrently.

Specifically, in `CoWSwapFarmBase`, the `_checkSwapApproveAndSignOrder` function calls `forceApprove` for USDC before initiating a swap. If deposits arrive at high frequency and swaps are triggered more dynamically, repeated calls to `forceApprove` could lead to race conditions or overwrite approval amounts unexpectedly.

Although not currently exploitable, using `safeIncreaseAllowance` would follow a more defensive pattern, especially when integrating with third-party protocols. It avoids unintentional overwrites and provides more granular control over allowance changes.

The use of `forceApprove` is also present in `PendleInfinifiSIUSD` during interactions with Pendle contracts. While appropriate in those cases due to the deterministic nature of approvals before a one-time interaction, it's worth reviewing all occurrences for consistency and future extensibility.

Recommendation: Consider to replace `forceApprove` with `safeIncreaseAllowance` where possible to follow best practices and avoid allowance overwriting in future changes that introduce higher-frequency or multi-party interactions.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.

3.3.7 Consider swapping USDC to USDe before staking to optimize for solver fees

Severity: Informational

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

Description: In the current implementation of `EthenaFarm`, the `signSwapOrder` function consistently performs swaps from USDC to sUSDe via CowSwap, as seen in the integration tests. While this flow is functional and aligns with the intended behavior, it may not always be optimal in a production environment.

Specifically, CowSwap solvers may charge higher fees when sourcing sUSDe directly from USDC, particularly in situations where sUSDe has lower available liquidity or fragmented pools. This could reduce the total net asset value (NAV) of the farm due to increased slippage or fee spread.

An alternative and potentially more cost-effective route would be to first swap USDC to USDe (which typically has deeper liquidity), and then stake the USDe into sUSDe. This would avoid excessive solver premiums and preserve more value for users.

The `EthenaFarm` architecture appears to support this adjusted flow without requiring structural changes, making it a viable improvement path.

Recommendation: Consider to perform swaps from USDC to USDe before staking into sUSDe in production deployments. This may reduce solver fees and improve the effective NAV of the farm, especially under volatile or high-volume conditions.

infiniFi: Acknowledged.

Cantina Managed: Acknowledged by infiniFi team.