infiniFi contracts Security Review

Auditors

Noah Marconi, Lead Security Researcher

R0bert, Lead Security Researcher

Slowfi, Security Researcher

Jonatas Martins, Associate Security Researcher

Report prepared by: Lucas Goiriz

April 1, 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

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

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of infiniFi contracts 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 11 days in total, infiniFi engaged with Spearbit to review the infinifi-contracts protocol. In this period of time a total of **67** issues were found.

Summary

Project Name	infiniFi
Repository	infinifi-contracts
Commit	90bb8d93
Type of Project	DeFi, Stablecoin
Audit Timeline	Mar 3rd to Mar 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	8	8	0
Medium Risk	6	5	1
Low Risk	25	16	9
Gas Optimizations	4	1	3
Informational	24	11	13
Total	**67**	**41**	**26**

5 Findings

5.1 High Risk

5.1.1 Griefing vector via arbitrary minting of iUSD tokens

Severity: High Risk

Context: MintController.sol#L99

Description: The `mint` function in the `InfiniFiGatewayV1` contract allows anyone to mint iUSD receipt tokens to an arbitrary address specified by the `_to` parameter. This functionality introduces a significant griefing vector that can disrupt user and contract operations within the system. Specifically, each call to `mint` triggers an external call to `ReceiptToken(receiptToken).restrictActionUntil(_to, block.timestamp + restrictionDuration);`, which restricts the target address from performing certain actions, such as transferring receipt tokens, for a period defined by the `restrictionDuration` state variable. The `restrictionDuration` is currently set to a minimal value of 1 and so the minimum mint amount, making the cost of repeatedly invoking this function extremely low.

This design enables an attacker to grief users and contracts by minting small amounts of iUSD to targeted addresses, thereby repeatedly extending their restriction periods. For users, this can block operations like `mintAndStake`, `mintAndLock`, `createPosition`... in the `InfiniFiGatewayV1` contract, as well as interfere with voting mechanisms. These operations can be front-run by an attacker calling `mint(<InfiniFiGatewayV1 address>, 1)`, causing subsequent user transactions to fail if they involve restricted actions.

Additionally, this vulnerability impacts contracts that transfer iUSD, such as the `YieldSharing` contract. In the `accrue` function, if a positive yield is detected, `_handlePositiveYield` is called, which includes a transfer of receipt tokens to a performance fee recipient: `ReceiptToken(receiptToken).transfer(performanceFeeRecipient, fee);`. If an attacker mints iUSD to the `YieldSharing` contract address (e.g., via `mint(<YieldSharing address>, 1)`), the contract becomes restricted, and this transfer fails, effectively causing a temporary denial of service (DoS) for the `accrue` function. An attacker could front-run every `accrue` call with such a mint, delaying or blocking yield processing indefinitely.

The root of this issue lies in the ReceiptToken contract's _update function, which enforces action restrictions during token transfers:

```
function _update(address _from, address _to, uint256 _value) internal override {
    if (_from != address(0) && _to != address(0)) {
        // check action restrictions if the transfer is not a burn nor a mint
        _checkActionRestriction(_from);
    }
    return ERC20._update(_from, _to, _value);
}
```

This function checks if the sender (`_from`) is restricted whenever a transfer occurs (excluding minting and burning). If restricted, the transfer reverts, halting any operation dependent on it. Since both `LockedPositionTokens` and `StakedTokens` inherit from `ActionRestriction`, they are similarly vulnerable if targeted by minting, potentially disrupting their respective functionalities.

In summary, the ability to mint iUSD to any address allows an attacker to impose repeated restrictions with minimal cost, denying users access to key operations and causing temporary DoS in contracts that rely on transferring iUSD.

Recommendation: Consider removing the possibility of minting to other addresses and always mint directly to the caller. Alternatively, if minting to other addresses is a required feature, consider increasing the `minMintAmount` to a much higher value so the griefing vector is not feasible due to the increased costs. Finally, consider having a set of whitelisted addresses that will not suffer the transfer restriction constrain and add all the protocol addresses to this whitelist.

infiniFi: Fixed in commit e893df4 by removing `ActionRestriction` from iUSD and siUSD tokens.

Spearbit: Fix verified.

5.1.2 Double-claiming of accrued rewards via unwinding after accrue call

Severity: High Risk

Context: UnwindingModule.sol#L103

Description: When a user is in the `LockingController` during an `accrue` event, they receive their share of newly deposited rewards for that epoch. By immediately calling `startUnwinding` in the same epoch, they move into the `UnwindingModule` just in time for the `UnwindingModule's` reward distribution logic to count them as though they were always present on the unwinding side. This effectively grants them a second portion of the same rewards. In practice, the user gains extra shares at the expense of other unwinding users, whose balances are unfairly diluted. This exploit occurs because the `UnwindingModule's balanceOf` calculation begins iterating from `position.fromEpoch - 1`, meaning that if a user arrives right after an `accrue` call, they are treated as if they had been in the unwinding side throughout that epoch.

Recommendation: Adjust the reward distribution logic so that a user cannot receive two sets of rewards for the same epoch. One option is to remove or revise the `position.fromEpoch - 1` iteration start in the `balanceOf` function and ensure that newly arrived users in the `UnwindingModule` are ineligible for the current epoch's distribution if they were counted on the `LockingController` side.

infiniFi: Fixed in d683c30.

Spearbit: Fix verified.

5.1.3 Incorrect accounting in `LockingController.applyLosses` function

Severity: High Risk

Context: LockingController.sol#L415

Description: In the `applyLosses` function, the code attempts to distribute the negative yield `_amount` among each epoch bucket by computing:

```
uint256 allocation = epochTotalReceiptToken.mulDivUp(_amount, _globalReceiptToken);
_globalReceiptToken -= allocation;
```

Because `_globalReceiptToken` is decremented in each iteration of the `for` loop, each bucket's allocation is calculated over a progressively smaller denominator. This will cause the sum of all allocations to exceed `_amount`, leading to more slash than intended. Worse, at the end of the loop, `globalReceiptToken` and `globalRewardWeight` are incorrectly set to these artificially decremented values, corrupting the final accounting.

Recommendation: To fix this accounting error and ensure accurate loss distribution, the `allocation` calculation should use the state variable `globalReceiptToken` as the denominator instead of `_globalReceiptToken`. The corrected line of code should read:

```
uint256 allocation = epochTotalReceiptToken.mulDivUp(_amount, globalReceiptToken);
```

infiniFi: Fixed in b9af8d.

Spearbit: Fix verified.

5.1.4 Referencing the gateway balance in `LockingController.increaseUnwindingEpochs` can DoS the function

Severity: High Risk

Context: LockingController.sol#L253

Description: `LockedPositionTokens` are transferable unrestricted until used to vote. This allows holders to perform blind transfers to the gateway contract. The above capability coupled with the `LockingController.increaseUnwindingEpochs` use of `msg.sender`'s balance means that the existence of any prior transfers to the gateway result in attempting to `burnFrom` the gateway an amount larger than it will ever approve.

Proof of Concept: Scenario:

- Alice creates position.
- Bob creates position.
- Bob sends his position to the gateway.
- Alice attempts to `increaseUnwindingEpochs` and reverts.

Revert is due to the function attempting to `burnFrom` the gateway both her position and bob's transferred tokens. Below is a modified version of `testIncreaseUnwindingEpochs` which now reverts with the addition of bob's transfer.

```
function testDosIncreaseUnwindingEpochs() public {
    _createPosition(alice, 1000, 10);
    _createPosition(bob, 1000, 10);

    LockedPositionToken liusd = LockedPositionToken(lockingController.shareToken(10));

    // Mirrors `testIncreaseUnwindingEpochs` test with the only difference being bob transfers to the
    ↪  gateway.
    vm.prank(bob);
    liusd.transfer(address(gateway), 1000);

    assertEq(
        lockingController.balanceOf(alice),
        1000,
        "Error: Alice's balance after creating first position is not correct"
    );
    assertEq(
        lockingController.rewardWeight(alice),
        1200,
        "Error: Alice's reward weight after creating first position is not correct"
    );
    assertEq(
        lockingController.shares(alice, 10),
        1000,
        "Error: Alice's share after creating first position is not correct"
    );
    assertEq(
        lockingController.shares(alice, 12),
        0,
        "Error: Alice's share after increasing unwinding epochs is not correct"
    );

    vm.startPrank(alice);
    {
        MockERC20(lockingController.shareToken(10)).approve(address(gateway), 1000);
        gateway.increaseUnwindingEpochs(10, 12);
    }
    vm.stopPrank();

    assertEq(
        lockingController.balanceOf(alice),
        1000,
        "Error: Alice's balance after increasing unwinding epochs is not correct"
    ); // unchanged
    assertEq(
        lockingController.rewardWeight(alice),
        1240,
        "Error: Alice's reward weight after increasing unwinding epochs is not correct"
    ); // +40
    assertEq(
        lockingController.shares(alice, 10),
        0,
```

```
        "Error: Alice's share after increasing unwinding epochs is not correct"
    ); // -1000
    assertEq(
        lockingController.shares(alice, 12),
        1000,
        "Error: Alice's share after increasing unwinding epochs is not correct"
    ); // +1000
}
```

Recommendation: In the gateway make use of the `shares` variable `uint256 shares = liusd.balanceOf(msg.sender);` and pass as argument into `LockingController.increaseUnwindingEpochs`. This requires an interface edit to `LockingController` as well.

infiniFi: Fixed in 708f8bf.

Spearbit: Fix verified.

5.1.5 `StakedToken` holders can circumvent `restriction` by approving another address to withdraw

Severity: High Risk

Context: StakedToken.sol#L139

Description: `StakedTokens` come with transfer restricted functionality. Given that withdraw burns tokens, a path to circumventing the transfer restriction exists.

First, the `_withdraw` function checks that the `caller` is not restricted but not the `owner` or `receiver`. The caller in the exploit scenario may be an address that has no `StakedTokens` and therefore has not been restricted.

```
function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)
    internal
    override
{
    _checkActionRestriction(caller);
    return ERC4626._withdraw(caller, receiver, owner, assets, shares);
}
```

Then on the `_update` the `_from` restriction check is ignored when minting or burning.

```
function _update(address _from, address _to, uint256 _value) internal override {
    if (_from != address(0) && _to != address(0)) {
        // check action restrictions if the transfer is not a burn nor a mint
        _checkActionRestriction(_from);
    }
    return ERC20._update(_from, _to, _value);
}
```

Proof of Concept:

```
function testRestrictionCircumvention() public {
    address anyAddress = makeAddr("anyAddress");

    vm.startPrank(alice);
    siusd.approve(anyAddress, 1000e18);

    vm.expectRevert(abi.encodeWithSelector(ActionRestriction.ActionRestricted.selector, alice,
    ↪  block.timestamp + 1));
    siusd.withdraw(1000e18, alice, alice);
    vm.stopPrank();

    vm.prank(anyAddress);
    siusd.withdraw(1000e18, alice, alice);
    assertEq(siusd.balanceOf(address(alice)), 0e18);
    assertEq(iusd.balanceOf(address(alice)), 1000e18);
}
```

Recommendation: Ensure the `owner` address argument is also validated to not be restricted.

infiniFi: Fixed in commit 913960a9.

Spearbit: Fix verified.

5.1.6 Broken accounting when cancelUnwinding is called one epoch after startUnwinding

Severity: High Risk

Context: UnwindingModule.sol#L189

Description: When a user calls `startUnwinding` the `positions` mapping is updated as:

```
positions[id] = UnwindingPosition({
    shares: newShares,
    fromEpoch: nextEpoch,
    toEpoch: endEpoch,
    fromRewardWeight: rewardWeight,
    rewardWeightDecrease: rewardWeightDecrease
});
```

For example, if a user calls `startUnwinding` in epoch 1000 with 10 as `_unwindingEpochs`, the `positions` mapping would be created as:

```
positions[id] = UnwindingPosition({
    shares: newShares,
    fromEpoch: 1001,
    toEpoch: 1011,
    fromRewardWeight: rewardWeight,
    rewardWeightDecrease: rewardWeightDecrease
});
```

Moreover, these 2 mappings would also be updated:

```
rewardWeightDecreases[1001] += rewardWeightDecrease;
rewardWeightIncreases[1011] += rewardWeightDecrease;
```

Then, once in the epoch 1001, this user would be able to call `cancelUnwinding` as this require check would pass:

```
// currentEpoch = 1001
// position.fromEpoch = 1001
require(currentEpoch >= position.fromEpoch, UserUnwindingNotStarted());
```

`cancelUnwinding` would execute the following logic:

```
function cancelUnwinding(address _user, uint256 _startUnwindingTimestamp, uint32 _newUnwindingEpochs)
    external
    onlyCoreRole(CoreRoles.LOCKED_TOKEN_MANAGER)
{
    uint32 currentEpoch = uint32(block.timestamp.epoch());
    bytes32 id = _unwindingId(_user, _startUnwindingTimestamp);
    UnwindingPosition memory position = positions[id];
    require(position.toEpoch > 0 && currentEpoch < position.toEpoch, UserNotUnwinding());
    require(currentEpoch >= position.fromEpoch, UserUnwindingNotStarted());

    uint256 userBalance = balanceOf(_user, _startUnwindingTimestamp);
    uint256 elapsedEpochs = currentEpoch - position.fromEpoch;
    uint256 userRewardWeight = position.fromRewardWeight - elapsedEpochs *
    ↪  position.rewardWeightDecrease;

    {
        // scope some state writing to avoid stack too deep
        GlobalPoint memory point = _getLastGlobalPoint(); // <-------------------------
        point.totalRewardWeightDecrease -= position.rewardWeightDecrease; // <-------------------------
        point.totalRewardWeight -= userRewardWeight;
        _updateGlobalPoint(point); // <-------------------------
        rewardWeightIncreases[position.toEpoch] -= position.rewardWeightDecrease;

        delete positions[id];

        totalShares -= position.shares;
        totalReceiptTokens -= userBalance;
    }

    uint32 remainingEpochs = position.toEpoch - currentEpoch;
    require(_newUnwindingEpochs >= remainingEpochs, InvalidUnwindingEpochs(_newUnwindingEpochs));
    IERC20(receiptToken).approve(msg.sender, userBalance);
    LockingController(msg.sender).createPosition(userBalance, _newUnwindingEpochs, _user);
}
```

Where `_getLastGlobalPoint` would iterate over previous epochs excluding the `currentEpoch`, which is where the `rewardWeightDecreases[1001] += rewardWeightDecrease` update was performed:

```
function _getLastGlobalPoint() internal view returns (GlobalPoint memory) {
    GlobalPoint memory point = globalPoints[lastGlobalPointEpoch];
    // apply slope changes
    uint32 currentEpoch = uint32(block.timestamp.epoch());
    for (uint32 epoch = point.epoch; epoch < currentEpoch; epoch++) { // <<<
        point.totalRewardWeightDecrease -= rewardWeightIncreases[epoch];
        point.totalRewardWeightDecrease += rewardWeightDecreases[epoch];
        point.totalRewardWeight -= point.totalRewardWeightDecrease;
        point.epoch = epoch + 1;
        point.rewardShares = 0;
    }
    return point;
}
```

Despite that, `cancelUnwinding` still performs the following update, incorrectly pushing this update into the global point:

```
GlobalPoint memory point = _getLastGlobalPoint();
point.totalRewardWeightDecrease -= position.rewardWeightDecrease; // <<<
```

At this point:

1. `totalRewardWeightDecrease` is higher than it should as the counter-part of this update (`rewardWeightDecreases[1001]`) was not yet processed.
2. Consequently, `totalRewardWeight` is lower than it should.
3. This error is written permanently into the global point here: `_updateGlobalPoint(point)`.

Therefore, as the slope data is left in an inconsistent state forcing an incorrect `totalRewardWeight` distribution for everyone else in the `UnwindingModule`, users which balance is affected by this slope, will receive a higher amount of receipt tokens than they should.

Another consequence is that this would cause a DoS of the InfiniFi protocol as the `UnwindingModule.totalRewardWeight` calls would revert due to underflow in the following line:

```
point.totalRewardWeightDecrease -= rewardWeightIncreases[epoch];
```

Recommendation: Consider updating the `cancelUnwinding` function require check to disallow users from cancelling when `currentEpoch` is equal to `position.fromEpoch`:

```
require(currentEpoch > position.fromEpoch, UserUnwindingNotStarted());
```

infiniFi: Fixed in a765744.

Spearbit: Fix verified.

5.1.7 Lock of user funds due to zero slashIndex update

Severity: High Risk

Context: UnwindingModule.sol#L292

Description: In the InfiniFi protocol, the `UnwindingModule` and `LockingController` contracts work together to manage the locking and unwinding of user tokens. The `slashIndex` is a pivotal variable that adjusts the value of user positions (e.g., reward weights) to reflect losses in the system. Under normal conditions, `slashIndex` starts at 1.0 (`1e18`) and decreases proportionally with losses, for example, dropping to 0.9 (9e17) after a 10% loss. However, a severe edge case arises when a loss event burns all Receipt Tokens from the `UnwindingModule`, setting `slashIndex` to 0 to indicate a complete loss of value in the `UnwindingModule`.

This design choice introduces a significant issue in the `startUnwinding` function, which users must call to initiate the unwinding of their locked tokens. The function includes the following calculation:

```
uint256 rewardWeight = _rewardWeight.divWadDown(slashIndex);
```

Here, `_rewardWeight` is the user's current reward weight and `divWadDown` performs a division with 18-decimal precision, rounding down. When `slashIndex` is 0, this line attempts to divide `_rewardWeight` by zero, which will cause a panic error and revert. As a result, users can not execute `startUnwinding`, preventing them from transitioning their locked tokens into the unwinding state.

The implications are severe: users with tokens locked in the `LockingController` are trapped, unable to access or recover their funds. This issue leads to a permanent loss of assets, as the protocol does not provide an alternative mechanism for users to bypass the failed `startUnwinding` call or retrieve their tokens when `slashIndex` is 0.

Recommendation: There is not a straightforward fix. A possible solution could be updating the `UnwindingModule.applyLosses` function to reset the `slashIndex` to 1 (`1e18`) whenever the remaining `totalReceiptTokens` are zero:

```
function applyLosses(uint256 _amount) external onlyCoreRole(CoreRoles.LOCKED_TOKEN_MANAGER) {
    if (_amount == 0) return;
    uint256 _totalReceiptTokens = totalReceiptTokens;
    ERC20Burnable(receiptToken).burn(_amount);
    slashIndex = slashIndex.mulDivDown(_totalReceiptTokens - _amount, _totalReceiptTokens);
    totalReceiptTokens = _totalReceiptTokens - _amount;
    if(totalReceiptTokens == 0){
        slashIndex = 1e18;
    }
}
```

However, this would break the `totalRewardWeight` logic. It would also be required to implement a way to reset `totalRewardWeight` to 0 in all the previous slopes until the current epoch.

infiniFi: Fixed in 66182a3 and 06dca30.

Spearbit: Fix verified.

5.1.8 Incorrect Calculation of User Shares in `startUnwinding`

Severity: High Risk

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

Description: The `startUnwinding` function in `LockingController` calculates user shares using the `balanceOf` function instead of using the provided `_shares` parameter. If residual or accidental shares are transferred to the gateway, the computed `userReceiptToken` value will be larger than expected. Additionally, the function transfers the amount specified by the `_shares` parameter from the gateway, meaning the inflated value from `balanceOf` does not correctly reflect the intended user allocation.

Proof of Concept: The following test case demonstrates the issue:

1. Users Bob, Carol, and Alice create locked positions.
2. Alice accidentally transfers extra shares to the gateway. This simulates a scenario where unintended shares accumulate at the gateway.
3. Bob and Carol initiate unwinding using `startUnwinding`. Since the function calculates the shares using `balanceOf`, the extra shares in the gateway cause an incorrect calculation of `userReceiptToken`.
4. After waiting for the required epochs, Bob and Carol withdraw their funds. The extra shares remain in the gateway.
5. An emergency governor action transfers the remaining shares back to Alice. This clears out the extra shares from the gateway.
6. Alice attempts to unwind her shares but encounters a revert. Since the calculation of `userReceiptToken` in previous steps included extra shares, the system state becomes inconsistent.

```
function test_createPositionAndStartUnwinding() public {
    _mintAndLock(bob, 1000e6, 1);
    _mintAndLock(carol, 1000e6, 1);
    _mintAndLock(alice, 1000e6, 1);
    vm.warp(block.timestamp + 7 days);
    LockedPositionToken liusd = LockedPositionToken(lockingController.shareToken(1));

    vm.startPrank(alice);
        liusd.transfer(address(gateway), liusd.balanceOf(alice)); // Accidental transfer
    vm.startPrank(bob);
        liusd.approve(address(gateway), liusd.balanceOf(bob));
        gateway.startUnwinding(liusd.balanceOf(bob), 1); // Bob gets 2k of receipt token to unwind
    vm.stopPrank();
```

```
    vm.startPrank(carol);
        liusd.approve(address(gateway), liusd.balanceOf(carol));
        gateway.startUnwinding(liusd.balanceOf(carol), 1);
    vm.stopPrank();

    // Time to withdraw passing the necessary epochs
    vm.warp(block.timestamp + 14 days);
    vm.prank(bob);
        gateway.withdraw(block.timestamp - 14 days);
    vm.prank(carol);
        gateway.withdraw(block.timestamp - 14 days);

    vm.startPrank(address(gateway)); // Emergency rescue call by governor
        liusd.transfer(alice, liusd.balanceOf(address(gateway))); // Transfers residual shares
    vm.stopPrank();

    iusd.balanceOf(bob);
    iusd.balanceOf(carol);
    iusd.balanceOf(alice);

    vm.startPrank(alice);
        liusd.approve(address(gateway), liusd.balanceOf(alice));
        gateway.startUnwinding(liusd.balanceOf(alice), 1); // It will revert
    vm.stopPrank();
}
```

Recommendation: Consider using the `_shares` parameter directly instead of relying on `balanceOf` to determine the amount of user shares. This approach prevents the issue of accumulating accidental transfers in the gateway and may also help reduce gas costs by avoiding unnecessary balance lookups.

infiniFi: Fixed in d8b0616 by using the given input parameter instead if the balance of the `gateway`.

Spearbit: Fix verified.

5.2 Medium Risk

5.2.1 DoS in all deposits by reaching max. cap in one of the farms

Severity: Medium Risk

Context: AfterMintHook.sol#L36-L44

Description: The `AfterMintHook` implementation could lead to a denial of service (DoS) condition, halting all deposit operations across the protocol. This issue occurs due to the behavior of the `_findOptimalDepositFarm` function, which is tasked with selecting the most appropriate farm for depositing assets based on parameters such as farm weights, total power and available assets. The flaw is that this function does not check whether the selected farm has already reached its maximum deposit capacity, as enforced by the farm's own deposit function. In the farm contract, the deposit function includes a cap check:

```
function deposit() external onlyCoreRole(CoreRoles.FARM_MANAGER) whenNotPaused {
    uint256 currentAssets = assets();
    if (currentAssets > cap) {
        revert CapExceeded(currentAssets, cap);
    }
    _deposit();
}
```

If the farm's current asset level exceeds its predefined cap, the function reverts with a `CapExceeded` error. In the `AfterMintHook` contract, after `_findOptimalDepositFarm` selects a farm, the code proceeds to call `IFarm(farm).deposit()`:

```
address farm = _findOptimalDepositFarm(farms, weights, totalPower, totalAssets, _assetsIn);
if (farm == address(0)) {
    // No optimal farm found, skip the deposit
    return;
}
IFarm(msg.sender).withdraw(_assetsIn, address(farm));
IFarm(farm).deposit();
```

If the selected farm has reached its cap, this `deposit` call will revert, causing the entire transaction in `AfterMintHook` to fail. Since `_findOptimalDepositFarm` does not account for farm capacities, it will repeatedly select a farm that is already full leading to a persistent failure of all deposit attempts. This effectively creates a DoS condition, blocking new deposits across the protocol until the situation is manually addressed, such as by adjusting farm caps or weights.

Recommendation: To address this, the `_findOptimalDepositFarm` function should be modified to verify each farm's available capacity before selecting it for a deposit. Specifically, it should ensure that adding the incoming deposit amount (`_assetsIn`) to the farm's current assets does not exceed its cap. If a farm would surpass its cap with the new deposit, it should be skipped, and the function should continue evaluating other farms. Only if no farm can accommodate the deposit without exceeding its cap should the function indicate that the system is fully utilized, reverting with an error that indicates that the protocol has no capacity to handle that deposit.

infiniFi: Partially fixed in d12c936 by adding pausability on the hooks which will allow faster reaction if there is an issue on farm movements. Guardian can be used to unstuck the situation & manual rebalancer to deploy capital, in the mean-time, before governor can change the hooks configuration.

Spearbit: While the new implementation does not totally prevent the issue it really allows the InfiniFi team to react and correct it in case it occurs.

5.2.2 Rounding error in partial funding burns fewer receipt tokens than consumed assets

Severity: Medium Risk

Context: RedemptionPool.sol#L71-L72

Description: In the `RedemptionPool` contract, the following code determines how many receipt tokens should be burned when a redemption request is partially fulfilled using the available assets(`remainingAssets`):

```
receiptToBurn = remainingAssets.divWadDown(_convertReceiptToAssetRatio);
uint96 newReceiptAmount = request.amount - uint96(receiptToBurn);
```

The variable `_convertReceiptToAssetRatio` is a fixed-point number that defines the conversion rate between receipt tokens and underlying assets, typically with 6 decimal places of precision if the underlying asset is USDC which has just 6 decimals instead of 18. The function `divWadDown` performs fixed-point division and rounds the result downward, which introduces a subtle but significant issue: the calculated `receiptToBurn` is slightly less than the exact amount that should correspond to the assets being used. As a result, `newReceiptAmount`, which represents the remaining unfunded portion of the redemption request, becomes slightly higher than it should be.

To illustrate, imagine the contract has 10,000 units of remaining assets available to fund a redemption request. If `_convertReceiptToAssetRatio` is 1 (for simplicity), the precise calculation of receiptToBurn should be 10,000 receipt tokens. However, because `divWadDown` rounds down, any fractional component in the real division is discarded, potentially resulting in receiptToBurn being 9,999 instead. This means the protocol burns 9,999 receipt tokens for 10,000 assets, leaving the remaining request amount overstated by 1 unit.

This small discrepancy accumulates each time the queue partially funds a redemption, inflating the user's pending claims by more than they truly deserve. Under extreme conditions (e.g., a large deposit ratio and repeated partial fills), an attacker could exploit this rounding gap to gain free receipt tokens or drain additional assets from the protocol. Although this is less severe for 6-decimal assets like USDC (as `_convertReceiptToAssetRatio` will return a 6 decimal precision integer), 18-decimal assets such as WETH or DAI make the attack more feasible due to larger precision.

Recommendation: To address this vulnerability, the calculation of `receiptToBurn` should be modified to round up instead of down, ensuring that the protocol burns at least the exact number of receipt tokens corresponding to the assets used, or slightly more. This change prevents the under-burning of receipt tokens and eliminates the accumulation of precision loss that an attacker could exploit. The recommended adjustment is to simply replace `divWadDown` with `divWadUp`.

infiniFi: Fixed in 2d84c51.

Spearbit: Fix verified.

5.2.3 Temporary DoS in redemptions due to incorrect asset withdrawal logic in farms

Severity: Medium Risk

Context: BeforeRedeemHook.sol#L62

Description: In the `BeforeRedeemHook` contract, within the `beforeRedeem` function, the `IFarm(farm).withdraw(_assetAmountOut, msg.sender);` call attempts to withdraw a specified amount of assets from a farm selected by the `_findOptimalRedeemFarm` function and transfer them to the `RedeemController`. However, `_findOptimalRedeemFarm` selects a farm based on the total assets it reports via its `assets()` function, without distinguishing between assets that are immediately withdrawable (e.g., supplied to an underlying protocol like Aave) and those that are idle (e.g., sitting in the farm contract waiting to be supplied). This mismatch can cause the withdrawal to fail, reverting the entire redemption transaction and creating a temporary denial-of-service (DoS) condition.

To understand this, consider a scenario involving the `AaveV3Farm`. Suppose the farm holds 1,000 USDC as idle assets (not yet supplied to Aave) and 1,000 aUSDC (already supplied to Aave). The `assets()` function in `AaveV3Farm` is designed to report the total value of assets under its control:

```
function assets() public view override returns (uint256) {
    return super.assets() + ERC20(aToken).balanceOf(address(this));
}
```

Here, `super.assets()` returns the idle USDC (1,000), and `ERC20(aToken).balanceOf(address(this))` returns the supplied aUSDC (1,000), yielding a total of 2,000 USDC. If a user attempts to redeem 1,500 iUSDC, the `_findOptimalRedeemFarm` function might select `AaveV3Farm` because its reported total of 2,000 USDC exceeds the requested 1,500 USDC. However, when the withdrawal is executed via `IFarm(farm).withdraw(1500e6, msg.sender)`, the `AaveV3Farm`'s internal `_withdraw` function attempts to pull the entire 1,500 USDC directly from Aave:

```
function _withdraw(uint256 _amount, address _to) internal override {
    IAaveV3Pool(lendingPool).withdraw(assetToken, _amount, _to);
}
```

Since Aave only holds 1,000 aUSDC, the withdrawal reverts, as it can not fulfill the 1,500 USDC request. This failure blocks the redemption process entirely, affecting all users until the idle 1,000 USDC are supplied to Aave (increasing the withdrawable aUSDC) in the next deposit that allocates to this farm.

A similar vulnerability exists in the rebalancing logic within the `singleMovement` function, where the withdrawal amount can be set to `IFarm(_from).assets()` if _amount is `type(uint256).max`. This again assumes all reported assets are immediately withdrawable, which may not be true if the farm holds idle funds, leading to failed withdrawals and stalled rebalances.

The core issue is that the protocol assumes all assets reported by `assets()` are readily available for withdrawal from the underlying protocol, whereas in reality, only the supplied portion (e.g., aUSDC) can be withdrawn instantly. This discrepancy can lock the system into a state where redemptions and rebalances fail until external conditions change, such as a deposit supplying the idle assets, an event that is not guaranteed to occur promptly, especially if the farm has low voting weight or is nearly full.

Recommendation: To address this vulnerability, upon withdrawals, farms should just withdraw the actual underlying amount that is part of the protocol, not the whole requested amount. A portion of the requested assets might be already available in the farm contract.

infiniFi: Partially fixed in d12c936 by adding pausability on the hooks which will allow faster reaction if there is an issue on farm movements. Guardian can be used to unstuck the situation & manual rebalancer to deploy capital, in the mean-time, before governor can change the hooks configuration.

Spearbit: While the new implementation does not totally prevent the issue it really allows the InfiniFi team to react and correct it in case it occurs.

5.2.4 First depositor inflation attack in `StakedToken` contract

Severity: Medium Risk

Context: StakedToken.sol#L14

Description: The `StakedToken` contract, inherits from the `ERC4626` standard. This contract is susceptible to a well-known exploit referred to as the "first depositor inflation attack." This vulnerability occurs due to the lack of protective mechanisms in the contract's design, allowing the initial depositor to manipulate the share price to their advantage. Such manipulation can enable the first depositor to disproportionately claim assets from the vault, effectively draining value at the expense of subsequent depositors. The problem is worsened by the `minMintAmount` parameter in the `MintController` contract being set to 1, which allows extremely small deposits. This low threshold makes the attack both practical and economically viable for an attacker with minimal upfront capital.

An attacker could simply:

1. Stake 1 receipt token into the contract. 1:1 share price.
2. Donate 10000 receipt tokens directly to the `StakedToken` contract. The share price becomes 10001 units per share.
3. Following deposits lower than 10000 receipt tokens will receive no shares and the deposit will be absorbed by the single share in the system that belongs to the first depositor.

In the `StakedToken` contract, no specific safeguards exist to prevent this attack. There is no minimum initial deposit requirement nor use of virtual shares to stabilize the share price and no other mechanisms to ensure fairness.

Recommendation: Prevent the vault from assigning full ownership of the share supply to a minuscule deposit. One approach is to set an initial “seed" deposit so that subsequent deposits must buy shares at a fair price.

infiniFi: Acknowledged as an already known issue. The deployer will initialize the vaults with an initial deposit to prevent the issue.

Spearbit: Acknowledged.

5.2.5 UnwindingModule users can avoid slashes by withdrawing before the accrue call

Severity: Medium Risk

Context: UnwindingModule.sol#L103

Description: In the `UnwindingModule`, users who have begun unwinding and are part of it, still are exposed to any earned yield and losses from `accrue` calls. However, as `accrue` is not called upon withdrawals, users could front-run any `accrue` call that will result in a slash by calling `withdraw`. The remaining users in the `UnwindingModule` would absorb the full impact of the negative yield. This could be easily abused by users within the infiniFi protocol to be exposed to yield gains without risking any losses/future slash.

Recommendation: Consider calling `YieldSharing.accrue` before any `UnwindingModule.withdraw` call. This way, no one in the `UnwindingModule` can avoid their share of a slash simply by withdrawing preemptively.

infiniFi: Fixed in 24513bd by reverting if there are pending unaccrued losses when a user does siUSD $\rightarrow$ iUSD, liUSD $\rightarrow$ iUSD or iUSD $\rightarrow$ USDC.

Spearbit: Fix verified.

5.2.6 `StakedToken` holders can avoid losses

Severity: Medium Risk

Context: StakedToken.sol#L64

Description: Holders of `StakedToken` are exposed to losses when the `accrue` function triggers `applyLosses`. However, they can circumvent these losses by frontrunning the `accrue` function, redeeming their tokens moments before execution. This vulnerability enables users to sidestep their intended risk exposure entirely.

Recommendation: Consider implementing a minimum unstaking period for `StakedToken` holders to prevent avoidance from loss.

infiniFi: Fixed in 24513bd by reverting if there are pending unaccrued losses when a user does siUSD → iUSD, liUSD → iUSD or iUSD → USDC.

Spearbit: Fix verified.

5.3 Low Risk

5.3.1 Precision loss in `PendleV2Farm` yield interpolation causes asset value step-jump

Severity: Low Risk

Context: PendleV2Farm.sol#L235

Description: In the `PendleV2Farm` contract, the calculation of `yieldPerSecond` introduces a precision loss that affects the reported value of `assets()` before and after the farm's maturity date. The problematic line is:

```
uint256 yieldPerSecond = totalYieldRemainingToInterpolate / (maturity - _lastWrappedTimestamp);
```

This calculation aims to determine the rate at which yield is interpolated over time, distributing the remaining yield (`totalYieldRemainingToInterpolate`) linearly across the seconds between the last wrapping event (`_lastWrappedTimestamp`) and the maturity date (`maturity`). The issue occurs due to the disparity in scale between the numerator and denominator, coupled with Solidity's integer division, which truncates fractional results. The `totalYieldRemainingToInterpolate` is expressed in USDC decimals (6 decimal places, e.g., 1 USDC = `1,000,000` units), while the denominator (`maturity - _lastWrappedTimestamp`) is in seconds, potentially spanning a duration such as 90 days (`7,776,000` seconds). This large denominator, when divided into a relatively smaller numerator, results in significant precision loss, underestimating the daily yield increments and leading to a noticeable step-jump in the `assets()` value when the contract switches from pre-maturity interpolation to post-maturity accounting.

This could be abused, in a very extreme scenario, by locking 1 second before `maturity` date is reached for a guaranteed increased yield and therefore an easy profit.

Recommendation: There is no straightforward fix for this issue. Given the value of USDC is highly unlikely that this will be exploited. However, if an asset like WBTC (8 decimals and a way higher USD value) was used, this attack vector could become feasible.

infiniFi: Fixed in b27feef.

Spearbit: Fix verified.

5.3.2 Risk of griefing attack via small redemptions in RedeemController contract

Severity: Low Risk

Context: RedeemController.sol#L30

Description: The `RedeemController` enforces only a `minRedemptionAmount` of 1. This opens the door to a spam or queue-stuffing attack: an attacker can enqueue 10000x minuscule redemption requests, quickly filling the queue (which has a maximum length of 10000). As a result, legitimate users with valid redemption requests are blocked until the queue is partially cleared with a deposit, causing a temporary denial of service. Do note, that the attack, even if is expensive in terms of gas costs as it will require 10000+ external calls, can be performed in a single transaction through the use of a deployed smart contract that interacts directly with the Infinifi protocol.

Recommendation: Increase `minRedemptionAmount` to a more meaningful threshold so that each redemption request has a non-trivial size. This would discourage attackers from cheaply flooding the queue.

infiniFi: Acknowledged as an known issue as it is already documented and indicated in the code.

5.3.3 Arbitrary `address _router` and `bytes memory _calldata` allow the `FARM_SWAP_CALLER` to perform malicious transactions

Severity: Low Risk

Context: SwapFarm.sol#L101

Description: While trusted, the `FARM_SWAP_CALLER` would be assumed to be less trusted than `GOVERNOR`. The ability to control the `_router.call` address and `_calldata` make malicious transactions possible. The built in slippage protection prevents making the router an arbitrary address, such as the USDC token address, as the call must result in `assetsReceived > minAssetsOut`.

That said, is still possible to drain the contracts with repeated calls. `_SWAP_COOLDOWN` prevents atomic looping and lowers the severity of this issue. With slippage permitted to be 0.5% per `wrap` / `unwrap`. A malicious `FARM_SWAP_CALLER` can repeat calls to a contract they control and keep 0.5% of the Farm's assets each pass. With 10 minute cooldown delays and a compromised `FARM_SWAP_CALLER`, ~50% per day can be drained with diminishing returns (day one 52% of original total drained, day two 76% of original total drained, day three 88% of original total drained).

Recommendation: Ensure monitoring is in place and remove degrees of freedom from `FARM_SWAP_CALLER`:

- Check router against whitelisted addresses.
- Consider constructing the `_calldata` payload within the contract.

Caution, malicious hooks could similarly retain the `0.5%` slippage each call.

infiniFi: Fixed in 1ce0b0b.

Spearbit: Fix verified.

5.3.4 Inheriting `CoreControlled` means that `GOVERNOR` can circumvent the timelock

Severity: Low Risk

Context: Timelock.sol#L15

Description:`CoreControlled` contains an `emergencyAction` function to permit them to make arbitrary calls to arbitrary addresses. `(bool success, bytes memory returned) = target.call{value: value}(callData);`. Paired with the timelock contract, this function allows circumventing all timelock protections.

Recommendation: Override the `emergencyAction` function to `noop` or `revert`.

infiniFi: Fixed in commit 572ba3a9.

Spearbit: Fix verified.

5.3.5 AAVE's `PoolDataProvider` address should not be immutable

Severity: Low Risk

Context: AaveV3Farm.sol#L25

Description: The `PoolDataProvider` address is one that can change due to AAVE governance proposals and should not be an immutable value. See recent upgrade (AAVE v3 Governance proposal 252).

Impact Explanation: This information is only used in `maxDeposit` referenced elsewhere in the system only by `FarmRebalancer.singleMovement`. In the event of a catastrophic upgrade on the Aave side, `CoreControlled` allows for relevant handling and the broken farm could be removed from the registry and replaced by a new farm. For this reason the overall Severity is low.

Recommendation: Whenever the `PoolDataProvider` is needed, fetch it from the `AddressProvider` using `IAddressProvider(_addressProvider).getPoolDataProvider()`. Broadly, anticipate AAVE changes that can impact the Farm wrapper. Monitoring of the governance forum along with on chain monitoring is advised. The `CoreControlled` functionality of these Farms does assist in migrations if needed in the future.

infiniFi: Fixed in 54c7d35.

Spearbit: Fix verified.

5.3.6 Reward multiplier skew in `LockingController` causes disproportionate reward allocation over time

Severity: Low Risk

Context: YieldSharing.sol#L143-L152

Description: In the `LockingController` contract, a static `rewardMultiplier` is used to enhance the reward weight of locked tokens relative to staked tokens, incentivizing token locking. While this mechanism intends to balance participation between staking and locking, its static implementation introduces a significant skew in the reward distribution over time. As rewards are periodically distributed with every `accrue` call, the fixed multiplier fails to adjust to the changing proportions of staked and locked tokens, disproportionately favoring locked positions (due to the higher multiplier). Over time, this results in an escalating concentration of rewards among locked token holders, which undermines the fairness and long-term sustainability of the protocol's incentive framework.

The issue stems from the interplay between the unchanging `rewardMultiplier` (e.g., 1.2x) and the dynamic total reward weight of the system. Rewards are distributed according to the weighted contributions of staked and locked tokens, with locked tokens consistently receiving an amplified share due to the multiplier. Over successive reward cycles, this effect compounds, enabling locked positions to accrue a disproportionately large portion of rewards, even if the quantity of locked tokens remains constant. Consequently, staked tokens experience a diminishing share of rewards, which discourages staking participation and disrupts the intended equilibrium between the two mechanisms.

Example:

1. 10000 iUSDC in the `StakedToken` contract. Staked tokens multiplier = 1x.
2. 10000 iUSDC locked in the `LockingController`. Locked tokens multiplier = 1.2x.
3. 30000 iUSDC are distributed as yield through the `accrue` call.
4. Due to the higher multiplier, `LockingController` receives the 55% of the yield distributed while the `StakedToken` contract receives the 45%.
5. However, after the addition of these rewards, the next `accrue` call will allocate the 42% of the yield to the `StakedToken` and the 48% to the `LockingController`.
6. Locked token holders progressively capture a larger share of rewards, diminishing returns for staked token holders and skewing the distribution in favor of locking.

The reward rate should fairly reflect user actions or, which is the same, deposits into and withdrawals from the `StakedToken` or `LockingController` contracts. These actions represent the users' intentional allocation of tokens and the reward system should adjust accordingly to maintain equity. Instead, the static multiplier distorts this balance, allowing locked token holders to accrue disproportionate rewards without further effort, while penalizing staked token holders. This violates the fairness principle that reward distribution should be driven solely by user-initiated changes in the system.

Recommendation: To mitigate this skew and restore fairness to the reward distribution, do not count the distributed rewards when calculating `stakedReceiptTokens` and `lockingReceiptTokens`. These variables should only held user provided receipt tokens and not earned yield.

infiniFi: Acknowledged as this is by design. All users are auto-compounding, result would be the same if we distributed iUSD rewards and everyone compounded into their own position. A growing locked tranche is also allowing more allocation to illiquid farms, which increase the yield for everyone.

Spearbit: Acknowledged.

5.3.7 Incorrect handling of `AaveDataProvider` data when checking max deposit

Severity: Low Risk

Context: AaveV3Farm.sol#L62

Description: When calculating the maximum deposit through the `_underlyingProtocolMaxDeposit` in `AaveV3Farm` contract, the contract calls `AaveDataProvider` to get the `supplyCap`. A `supplyCap` of 0 indicates an uncapped pool, but the function treats this as 0 and reverts when calculating the max deposit, instead of returning `type(uint256).max`. Every pool that has an uncapped max deposit amount will revert when calling the `maxDeposit()` function.

Recomendation: Consider returning `type(uint256).max` when `supplyCap` is 0.

infiniFi: Fixed in c8bed20.

Spearbit: Fix verified.

5.3.8 `AaveV3Farm` supply cap not considering treasury accrued amount

Severity: Low Risk

Context: AaveV3Farm.sol#L69-L72

Description: In the `AaveV3.ValidationLogic` contract, the maximum deposit calculation considers the `accruedToTreasury` variable when checking against `supplyCap`:

```
require(
  supplyCap == 0 ||
    ((IAToken(reserveCache.aTokenAddress).scaledTotalSupply() +
      uint256(reserve.accruedToTreasury)).rayMul(reserveCache.nextLiquidityIndex) + amount) >=
    supplyCap * (10 ** reserveCache.reserveConfiguration.getDecimals()),
  Errors.SUPPLY_CAP_EXCEEDED
);
```

However, the `AaveV3Farm` contract calculates the maximum deposit using only `aToken.totalSupply()`, while it should also consider the amount accrued to treasury. By returning an incorrect maximum deposit, it could revert when trying.

Recommendation: Consider changing the code to the following:

```
- uint256 currentUnderlyingProtocolSupply = ERC20(aToken).totalSupply();

- return supplyCapInAssetTokenDecimals - currentUnderlyingProtocolSupply;
+ (, uint256 _accruedToTreasuryScaled, uint256 _totalAToken,,,,,,,,) =
+    IAaveDataProvider(dataProvider).getReserveData(address(asset));

+ //Supply cap already reached
+ if (_totalAToken + _accruedToTreasuryScaled >= _supplyCap) {
+     return 0;
+ }

+ return supplyCapInAssetTokenDecimals - (_totalAToken + _accruedToTreasuryScaled);
```

infiniFi: Fixed in 0b20a51.

Spearbit: Fix verified.

5.3.9 `AaveV3Farm` liquidity does not account for paused or inactive pools

Severity: Low Risk

Context: AaveV3Farm.sol#L40

Description: In `AaveV3Farm`, the `liquidity` function returns the maximum withdrawable amount. However, the `balanceOf` function of `AToken` does not accurately reflect the withdrawable amount. When the protocol is `paused` or `inactive`, withdrawals are impossible, but the liquidity calculation does not consider this state.

Recommendation: Consider checking the pool's paused and active status before including the `AToken` balance in liquidity calculations.

infiniFi: Fixed in 346c908.

Spearbit: Fix verified.

5.3.10 User can bypass `StakedToken` time restriction

Severity: Low Risk

Context: InfiniFiGatewayV1.sol#L71

Description: The `InfiniFiGatewayV1` contract enforces a timelock when users call the `mintAndStake()` function. However, this timelock can be bypassed by minting `iUSD` and then depositing directly to `StakedToken` through the `mint()` function. This allows users to mint and burn in the same block, potentially enabling manipulation attacks.

Recommendation: Consider adding minting and burning only through the `InfiniFiGatewayV1`.

infiniFi: Fixed in 913960a and 2d8b972.

Spearbit: Fix resolved by removing the time restriction and modifying the `LockedPositionToken` locking mechanism.

5.3.11 Use `forceApprove` instead of `approve`

Severity: Low Risk

Context: PendleV2Farm.sol#L129, PendleV2Farm.sol#L157

Description: The `PendleV2Farm` uses the `ERC20.approve()` function to handle tokens. While this works fine with USDC initially, issues can arise when integrating other tokens like USDT. Specifically, if previously approved tokens are not completely spent, USDT will revert when attempting to increase the approval amount.

Recommendation: Consider using `forceApprove` instead of `approve`.

infiniFi: Fixed in 8f5f706.

Spearbit: Fix verified.

5.3.12 `FarmRebalancer` doesn't consider the origin farm's max liquidity

Severity: Low Risk

Context: FarmRebalancer.sol#L71

Description: The `FarmRebalancer` considers the maximum deposit of the destination farm but overlooks the available liquidity in the origin farm. When the destination farm's maximum deposit limit exceeds the origin farm's available liquidity, withdrawal attempts will revert.

Recommendation: Consider checking the origin farm's available liquidity by adding the following code:

```
_amount = _amount > IFarm(_from).liquidity() ? IFarm(_from).liquidity() : _amount;
```

infiniFi: Acknowledged, won't fix.

Spearbit: Acknowledged.

5.3.13 Dust amounts may be lost during `increaseUnwindingEpochs`

Severity: Low Risk

Context: LockingController.sol#L256-L257

Description: The `increaseUnwindingEpochs()` function rebalances shares from the old bucket to the new bucket. The `receiptTokens` calculation uses `oldShares`, `totalReceiptTokens`, and `oldTotalSupply` in the line:

```
uint256 receiptTokens = oldShares.mulDivDown(oldData.totalReceiptTokens, oldTotalSupply);
```

When a user has a small amount of shares (`oldShares`), this calculation can round down to zero, causing the function to return early. As a result, any tokens transferred to the gateway become lost.

Recommendation: Consider returning any dust amount of shares to the user at the end of execution by checking the gateway balance and transferring the remainder back to the user.

infiniFi: Acknowledged, won't fix.

Spearbit: Acknowledged.

5.3.14 Votes can be carried forward multiple epochs

Severity: Low Risk

Context: AllocationVoting.sol#L171

Description: `AllocationVoting._storeUserVotes` includes a conditional branch to handle prior epochs for a farm's weight data. When `farmWeightData[farm].epoch` is not equal to the current epoch, the `nextWeight` is written to `currentWeight` for this farm during this epoch. There are no checks to ensure that `farmWeightData[farm].epoch` is the previous epoch as in `AllocationVoting._getFarmWeight` (which checks for and explicitly states `do not persist votes if they are older than 1 epoch ago`).

Anytime a farm is not voted on in an epoch, following an epoch where votes were cast, the votes carry forward until the farm receives a vote.

Proof of Concept:

```
function testAccumulateVote() public {
    _initAliceLocking(1000e6);
    uint256 aliceWeight = lockingController.rewardWeight(alice);

    AllocationVoting.AllocationVote[] memory liquidVotes = new AllocationVoting.AllocationVote[](1);
    AllocationVoting.AllocationVote[] memory illiquidVotes = new AllocationVoting.AllocationVote[](1);
    liquidVotes[0] = AllocationVoting.AllocationVote({farm: address(farm1), weight:
    ↪  uint96(aliceWeight)});
    illiquidVotes[0] = AllocationVoting.AllocationVote({farm: address(illiquidFarm1), weight:
    ↪  uint96(aliceWeight)});

    // cast vote
    vm.prank(alice);
    gateway.vote(address(usdc), 4, liquidVotes, illiquidVotes);

    // Move past lockup
    vm.warp(block.timestamp + (EpochLib.EPOCH * 4));

    // No votes.
    assertEq(allocationVoting.getVote(address(farm1)), 0, "Error: Vote for farm1 should be discarded");
    assertEq(
        allocationVoting.getVote(address(illiquidFarm1)), 0, "Error: Vote for illiquidFarm1 should be
        ↪  discarded"
    );

    vm.prank(alice);
    gateway.vote(address(usdc), 4, liquidVotes, illiquidVotes);

    // Votes reintroduced for current epoch, reverts when should not:
    assertEq(allocationVoting.getVote(address(farm1)), 0, "Error: Vote for farm1 should be discarded");
    assertEq(
        allocationVoting.getVote(address(illiquidFarm1)), 0, "Error: Vote for illiquidFarm1 should be
        ↪  discarded"
    );
}
```

Recommendation: Coupled with other issues, such as the voting DoS, this issue may increase in severity. Although rated low severity, it is recommended to enforce that votes are not carried forward further than intended.

infiniFi: Fixed in 53db79c.

Spearbit: Fix verified.

5.3.15 Enforce bounds on `setBucketMultiplier`

Severity: Low Risk

Context: LockingController.sol#L88

Description: The protocol references assumptions that the `rewardMultiplier` remains withing a particular range. Namely, the public rewardMultiplier suggests `totalWeight/totalBalance` should be `/// Expressed as a WAD (18 decimals). Should be between` [1.0e18, 2.0e18] `realistically.`.

Recommendation: Consider codifying a safe range. Assessing underflow in some areas is based on this assumption holding true.

infiniFi: Fixed in 0bbde85.

Spearbit: Fix verified.

5.3.16 `StakedToken` does not conform to spec when paused

Severity: Low Risk

Context: StakedToken.sol#L14

Description: Pausable makes the getters not conform to spec. E.g. "`maxMint` MUST factor in both global and user-specific limits, like if mints are entirely disabled (even temporarily) it MUST return 0". For more info see ERC-4626: Tokenized Vaults.

Recommendation: Override the getters to respond correctly during paused state. Confirm the issue does not exist on any of the ERC4626 vaults integrated with.

infiniFi: Fixed in f29f664.

Spearbit: Fix verified.

5.3.17 Getter for `getRoleAdmin` will be inaccurate as it looks locally and not to `core.getRoleAdmin`

Severity: Low Risk

Context: Timelock.sol#L13

Description: Consider overriding and reverting when called to avoid confusion between the inherited but overridden functionality and the `CoreControlled` functions.

infiniFi: Acknowledged, won't fix.

Spearbit: Acknowledged.

5.3.18 `CoreControlled` allows many calls on behalf of users in the Gateway

Severity: Low Risk

Context: InfiniFiGatewayV1.sol#L18

Description: The use of `CoreControlled` allows a many calls on behalf of users. `GOVERNANCE` may call each of the following with arbitrary arguments:

- `usdc.transferFrom`.
- `iusd.transferFrom`.
- `liusd.transferFrom`.
- `LockingController.startUnwinding`.
- `LockingController.increaseUnwindingEpochs`.
- `LockingController.cancelUnwinding`.
- `LockingController.withdraw`.
- `RedeemController.claimRedemption`.
- `AllocationVoting.vote`.

Recommendation: The project team noted a solution of "consider overriding the `emergencyAction` for the Gateway contract with a noop". This would remove the elevated CoreControlled capability as it related to the Gateway. If it is determined that `emergencyAction` must remain on the Gateway, an override is still recommended to confirm that particular functions are not being called during and `emergencyAction`.

infiniFi: Acknowledged, won't fix. the role to use `emergencyAction` on the gateway is the same that can upgrade the protocol, so even if we override the `emergencyAction` with a noop, it would still be possible for `GOVERNOR` to upgrade the implementation to an implementation that does not override to no-op. `GOVERNOR` role is trusted and behind a long timelock.

Spearbit: Acknowledged.

5.3.19 `GOVERNOR` may enable a bucket that blocks unwinding

Severity: Low Risk

Context: LockingController.sol#L75

Description: There is no validation on the `_unwindingEpochs` argument of `LockingController.enableBucket`. This value is later used in `UnwindingModule.startUnwinding` as the denominator for a calculation: `uint256 rewardWeightDecrease = totalDecrease / uint256(_unwindingEpochs);`. If `GOVERNOR` accidentally enabled a bucket with 0 unwinding epochs, unwinding would revert.

Recommendation: Enforce a reasonable range to prevent `GOVERNOR` footguns.

infiniFi: Fixed in 89d123a.

Spearbit: Fix verified.

5.3.20 `_from` should be `_to` in the FarmRebalancer `maxDeposit` check

Severity: Low Risk

Context: FarmRebalancer.sol#L68

Description: The current implementation checks the wrong farm's max deposit function.

Recommendation:

```
- _amount = _amount > IFarm(_from).maxDeposit() ? IFarm(_from).maxDeposit() : _amount;
+ uint256 toMaxDeposit = IFarm(_to).maxDeposit();
+ _amount = _amount > toMaxDeposit ? toMaxDeposit : _amount;
```

infiniFi: Fixed in PR 85.

Spearbit: Fix verified.

5.3.21 Depositing of rewards is blocked when `_globalRewardWeight` and `unwindingRewardWeight` are 0

Severity: Low Risk

Context: LockingController.sol#L350

Description: Currently there are no likely scenarios identified where this issue surfaces however, with pending changes / fixes intended, it is worth noting that in some scenarios depositing of rewards will revert.

Proof of Concept: Reverts due to underflow:

```
function testRewards() public {
    // _createPosition(alice, 1000, 10); // 1200 reward weight
    // _createPosition(bob, 2000, 5); // 2200 reward weight

    _depositRewards(34);

    assertApproxEqAbs(lockingController.balanceOf(alice), 1012, 1, "Error: alice's balance is not
    ↪  correct"); // +12
    assertApproxEqAbs(lockingController.balanceOf(bob), 2022, 1, "Error: bob's balance is not
    ↪  correct"); // +22

    _depositRewards(34);

    assertApproxEqAbs(lockingController.balanceOf(alice), 1024, 1, "Error: alice's balance is not
    ↪  correct"); // +12
    assertApproxEqAbs(lockingController.balanceOf(bob), 2044, 1, "Error: bob's balance is not
    ↪  correct"); // +22
}
```

Recommendation: Exercise caution when calling accrue. Changes to the protocol could result in DoS scenarios.

infiniFi: Will be prevented by initializing the share price, by minting each of the bucket tokens on deployment, therefore this will never be 0.

Spearbit: Acknowledged.

5.3.22 Inconsistent Voting Power Calculation Due to Late Multiplier Change

Severity: Low Risk

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

Description: The `vote` function in `AllocationVoting` calculates a user's voting power using the `rewardWeightForUnwindingEpochs` function from `LockingController`. However, the `setBucketMultiplier` function in `LockingController` allows the multiplier to be changed dynamically within the same epoch.

This can lead to an inconsistency where two users with identical positions and balances at the beginning of an epoch may receive different voting power if one votes before and the other after the multiplier is updated.

The next scenario is used to demonstrate the issue:

1. Alice and Bob both hold identical positions within the same bucket and epoch, initially having the same voting power.
2. Bob votes first and receives voting power Z.
3. The `setBucketMultiplier` function is called to increase the multiplier by $1.5\times$.
4. Alice votes afterward and receives $1.5Z$ voting power for the same epoch.
5. This creates an imbalance where two identical positions yield different voting power depending on when the vote was cast.

Although this scenario may not happen frequently, it introduces an inconsistency that can impact fairness in the voting system.

Recommendation: To ensure voting power remains consistent across an epoch, consider locking the multiplier value at the start of each epoch to prevent mid-epoch changes. If updates to the multiplier are necessary, storing a snapshot of the multiplier at the time of each user's vote would help ensure a fair application of changes.

Another approach could be to restrict voting immediately after a multiplier change, preventing discrepancies within the same epoch.

infiniFi: These values are not expected to ever change but if they do: We will be doing these changes at the epoch turn-over. Therefore it won't make much difference for users.

Spearbit: Acknowledged.

5.3.23 Lack of Duration Cap in `setRestrictionDuration` May Lead to Indefinite Fund Locking

Severity: Low Risk

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

Description: The `setRestrictionDuration` function in `MintController` allows the `restrictionDuration` to be set arbitrarily without an upper limit. Since this value is used to control transfer and redemption restrictions, setting an excessively high duration could result in user funds being locked indefinitely. Additionally, this value may be compared with a smaller primitive type when handling timestamps, which could introduce unintended behavior if large values are used.

Recommendation: Consider enforcing a reasonable maximum cap on `restrictionDuration` to prevent unintended indefinite locking of funds. This can be achieved by adding a validation check in `setRestrictionDuration`, for example:

```
require(_duration <= MAX_RESTRICTION_DURATION, "Duration exceeds allowed limit");
```

infiniFi: Fixed on both commits 913960a and 2d8b972.

Spearbit: Fix verified. infiniFi team removed the redeem restriction and modified the action restriction mechanism by a transfer restriction, correlated with the main fix of this issue.

5.3.24 Lack of Maximum Boundary for `minRedemptionAmount` May Lead to Fund Locking

Severity: Low Risk

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

Description: The `setMinRedemptionAmount` function in `RedeemController` allows setting a minimum redemption amount without an upper limit. While this can be useful for preventing spam or griefing attacks with small redemption requests, an excessively high minimum redemption amount could prevent users from redeeming their funds entirely if their balance is below the set threshold. If `minRedemptionAmount` is set too high, users with smaller balances may be effectively locked out of redemption, which could lead to undesirable user experience and accessibility issues.

Recommendation: Consider enforcing a reasonable maximum cap on `minRedemptionAmount` to prevent unintended restrictions on redemptions. This can be achieved by adding a validation check in `setMinRedemptionAmount`, such as:

```
require(_minRedemptionAmount <= MAX_REDEMPTION_AMOUNT, "Redemption amount exceeds allowed limit");
```

where `MAX_REDEMPTION_AMOUNT` is defined as a safe upper bound.

infiniFi: Not implementing this as this action is controlled by `Governor` role.

Spearbit: Acknowledged by infiniFi team.

5.3.25 Slippage Condition May Be Too Restrictive

Severity: Low Risk

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

Description: In both `SwapFarm` (line 108) and `PendleV2Farm` (line 137), the slippage check enforces that the received assets must be strictly greater than the minimum expected amount. While this ensures slippage protection, it may be unnecessarily restrictive. A transaction could revert even when the received amount matches the minimum threshold exactly, potentially rejecting swaps that meet the intended slippage tolerance.

Recommendation: Consider adjusting the slippage check to allow the received amount to be **greater than or equal to** the minimum expected amount instead of strictly greater. This would prevent unnecessary reverts in cases where the received amount is exactly at the slippage threshold, improving execution reliability while still maintaining protection. For example, instead of:

```
require(assetsReceived > minAssetsOut, SlippageTooHigh(minAssetsOut, assetsReceived));
```

Consider:

```
require(assetsReceived >= minAssetsOut, SlippageTooHigh(minAssetsOut, assetsReceived));
```

This minor adjustment improves the robustness of swap transactions without compromising security or slippage guarantees.

infiniFi: Fixed on commit 661f359.

Spearbit: Fix verified.

5.4 Gas Optimization

5.4.1 Avoid recomputing address hash in `getAddress`

Severity: Gas Optimization

Context: InfiniFiGatewayV1.sol#L42

Description: An enum or using a library as an enum (pattern used elsewhere) would save re-computing the hash each call.

infiniFi: Acknowledged. No fix implemented.

Spearbit: Acknowledged.

5.4.2 Make `sUSDe` Constant

Severity: Gas Optimization

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

Description: The contract currently defines `sUSDe` as a regular state variable instead of a constant. Since `sUSDe` is not expected to change after deployment, marking it as `constant` can reduce gas costs by avoiding unnecessary storage reads. Using `constant` for immutable values allows the Solidity compiler to inline the value, reducing storage access costs and improving execution efficiency.

Recommendation: Declare `sUSDe` as a `constant` to optimize gas usage. This change will reduce gas costs associated with retrieving `sUSDe` and improve contract efficiency.

infiniFi: Fixed in 7646b67.

Spearbit: Fix verified.

5.4.3 Redundant Transfers and Approvals on currency transfer through the gateway

Severity: Gas Optimization

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

Description: The `mint` function in `InfiniFiGatewayV1` performs redundant transfers and approvals that could be optimized to reduce gas costs. Currently, the user first approves the `gateway`, then the `gateway` transfers tokens to itself, approves another contract (`mintController` or `lockingController`), and then that contract transfers the tokens again.

This pattern creates unnecessary gas usage, as the tokens are moved multiple times before reaching their final destination. Since the `gateway` does not retain the funds and instead delegates handling to another contract, a direct transfer from the user to the respective contract (`mintController` or `lockingController`) would save gas.

Identified Redundant Transfers and Approvals:

- Lines 55, 66, 82: `usdc.approve(address(mintController), _amount);`.
- Line 93: `iusd.transferFrom(msg.sender, address(this), _amount);`.
- Line 94: `iusd.approve(address(lockingController), _amount);`.
- Lines 102, 112: `liusd.transferFrom(msg.sender, address(this), _shares);`.
- Lines 103, 113: `liusd.approve(address(lockingController), _shares);`.

This inefficiency is repeated in multiple functions, resulting in unnecessary storage updates and approvals.

Recommendation: Instead of transferring tokens to `InfiniFiGatewayV1` first and then approving another contract, consider allowing users to approve `mintController` and `lockingController` directly, so that the gateway only forwards the necessary call without intermediating token transfers. For example, instead of:

```
iusd.transferFrom(msg.sender, address(this), _amount);
iusd.approve(address(mintController), _amount);
```

Consider modifying the design to:

```
iusd.transferFrom(msg.sender, address(mintController), _amount);
```

Similarly, for locking:

```
liusd.transferFrom(msg.sender, address(lockingController), _shares);
```

This modification reduces unnecessary state changes, optimizing gas usage while maintaining the same security guarantees.

infiniFi: The reason for this is so approval happens only once per token on a single contract. This is purely for UX reasons to have people be able to interact with our system without having to send two transactions. Changing this remains an option still if the gas impact is too high.

Spearbit: Acknowleged by infiniFi team.

5.4.4 Save Gas By Doing Internal Call

Severity: Gas Optimization

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

Description: In the `liquidity` function of `AaveV3Farm`, the call to `this.assets()` triggers an external call to the contract itself, which is more expensive than an internal function call. Since `assets()` is already a public function within the same contract, it can be called directly without using `this.` to avoid unnecessary external execution overhead.

Recommendation: Instead of using:

```
uint256 totalAssets = this.assets();
```

Consider using an internal call:

```
uint256 totalAssets = assets();
```

This minor change will reduce gas costs by avoiding an external contract call and instead using a direct function reference within the same contract.

infiniFi:: Fixed on commit ID 1e100de.

Spearbit: Fix verified.

5.5 Informational

5.5.1 Precision loss in `_processProportionalRedeem` leads to dust in redemption queue

Severity: Informational

Context: BeforeRedeemHook.sol#L106

Description: In the `BeforeRedeemHook` when `_processProportionalRedeem` is called, the code calculates each farm's share of `_amount` by doing `uint256 assetsOut = _amount.mulDivDown(farmBalance, _totalAssets);`. Because `mulDivDown` floors the result, the summation across all farms can end up slightly less than `_amount`. As a result, a tiny residual amount is never redeemed and thus will be sent to the redemption queue. For example, if a user redeems `1000,000000` USDC in total, it might distribute only `999,999999` across farms, leaving 1 dust in the queue. This dust will be lost by the user as the gas costs for claiming them will be higher than the value actually retrieved by the user.

Recommendation: Consider using a single leftover assignment logic for the last farm in `_processProportionalRedeem`. One simple fix is to compute each farm's share using floor, track a running total, and then assign any leftover difference between the sum and `_amount` to the final farm. This ensures the full `_amount` is proportionally distributed without sending residual to the redemption queue.

infiniFi: Fixed in 9e760c3.

Spearbit: Fix verified.

5.5.2 Epoch vote roll-over logic limitations

Severity: Informational

Context: AllocationVoting.sol#L129-L140

Description: In `AllocationVoting`, the `_getFarmWeight` function returns 0 if the farm's last vote was older than one epoch and no new vote has occurred this epoch. This is tied to the “roll over" approach, where `data.currentWeight` becomes `data.nextWeight` only after a new vote in the next epoch. As a result:

1. By minting/redeeming in the first second of a new epoch (before anyone has voted) you can still operate on the old epoch weights. (No major impact though as the same user could have done exactly this in the previous epoch).
2. If the contract is paused and no one votes for a full epoch in a farm, all farm weights will revert to 0 once the contract is unpaused and the first vote in the new epoch is stored.

Recommendation: Clearly document how farm weights are managed across epochs, including the conditions under which weights revert to 0. This should cover:

- The reliance on new votes to roll over weights from data.nextWeight to data.currentWeight.
- The impact of epoch transitions, especially when no votes occur.
- The effect of contract pauses, where weights reset to 0 after an epoch of inactivity.

infiniFi: Fixed in 7cc72f5.

Spearbit: Fix verified.

5.5.3 Accrue calls could be sandwiched if a farm yields a sudden lump sum profit

Severity: Informational

Context: YieldSharing.sol#L129

Description: The `accrue` function in `YieldSharing` calculates `unaccruedYield` and distributes profits to stakers and lock holders. If one of the integrated farms someday supports a “single-block harvest" or sudden lump-sum distribution (e.g., a function that instantly increases the farm's yield by a large amount), an attacker could front-run that harvest by minting and locking new `ReceiptTokens`, then immediately call `accrue`, capturing a disproportionate share of the windfall. The attacker can then unwind or redeem shortly after to lock in a guaranteed profit. Although the current farms (AaveV3, ERC4626, PendleV2, SwapFarm) do not exhibit one-shot lumps, adding a new farm with an instant distribution event would open this exploit possibility.

Recommendation: Be aware of this limitation and keep using farms that accrue yield continuously within the InfiniFi protocol.

infiniFi: Acknowledged as a known issue. All farms will be made in a such way that dumping interest won't be possible.

5.5.4 Hardcoded slippage tolerance in `PendleV2Farm` restricts operational flexibility

Severity: Informational

Context: PendleV2Farm.sol#L54

Description: The `PendleV2Farm` contract uses a constant `_MAX_SLIPPAGE = 0.995e18` (representing a 0.5% slippage ceiling) for both `wrapAssetToPt` and `unwrapPtToAsset` calls. If total swap fees and price impact exceed 0.5%, these functions revert. Although 0.5% might be adequate most of the time, a single hardcoded tolerance can break large trades or swaps in illiquid pools, especially when Pendle's AMM imposes roughly 0.3% in fees plus additional spot price deviations. An overly strict slippage constant forces the `FarmSwapCaller` to split large swaps into many small ones, raising gas costs and operational complexity. Because `_MAX_SLIPPAGE` is a compile-time constant, it cannot be adjusted if liquidity conditions change or protocol fees increase.

Recommendation: Replace `_MAX_SLIPPAGE` with a state variable that governance can modify in response to changing market conditions and fee levels. By making slippage tolerance dynamically adjustable, the farm can accommodate temporary spikes in fees or low liquidity without forcing repeated small transactions.

infiniFi: Fixed in 59f2628.

Spearbit: Fix verified.

5.5.5 Limited flexibility in voting mechanism due to absolute token amounts

Severity: Informational

Context: AllocationVoting.sol#L179

Description: In the `AllocationVoting` contract, the `vote` function enforces a strict requirement through the line:

```
require(weightAllocated == _userWeight || weightAllocated == 0, InvalidWeights(_userWeight,
↪  weightAllocated));
```

This check mandates that a user's total allocated voting weight, represented by their liUSD token holdings(`_userWeight`), must either be fully distributed across one or more farms or left entirely unallocated (zero). The design relies on users specifying their voting preferences in absolute token amounts, meaning a user with 100 liUSD must allocate exactly 100 units across their chosen farms or abstain completely. This rigid structure limits how users can express their governance preferences, as they lack the ability to allocate only a portion of their voting power while leaving the rest unused. For example, a user cannot allocate 70 liUSD to one farm and retain 30 liUSD unallocated, as the contract demands full commitment or none at all. This inflexibility can force users into making voting decisions that do not fully align with their intentions. Additionally, requiring precise sums of absolute token amounts increases the complexity for users, who must ensure their allocations exactly match their total voting power, which may change due to external factors like token transfers.

Recommendation: To improve the flexibility and usability of the voting mechanism in the `AllocationVoting` contract, the system should transition from using absolute token amounts to a percentage-based voting approach. Under this revised design, users would specify their voting preferences as percentages of their total voting power, with the sum of all allocations required to equal 100%, represented as `1e18` in 18-decimal precision. For instance, a user could allocate 50% (`5e17`) to one farm and 50% (`5e17`) to another, or 100% (`1e18`) to a single farm, offering a more natural way to distribute influence. The contract would then calculate the actual weight allocated to each farm by applying these percentages to the user's current `_userWeight` at the time of voting. To enforce correctness, the contract should include a check ensuring that the total percentage allocated across all votes equals exactly `1e18` or zero, replacing the current absolute weight comparison. This percentage-based method would simplify the voting process for users, eliminate the need for manual adjustments to match absolute token amounts and provide a more adaptable framework that aligns with common governance practices.

infiniFi: Acknowledged.

Spearbit: Do note that the current griefing vector where 1 user mints 1 liUSD to the user voting (front-running his `vote` call) causing a DoS is still present. The `vote` transaction would revert in the following line in the `AllocationVoting._storeUserVotes` function:

```
require(weightAllocated == _userWeight || weightAllocated == 0, InvalidWeights(_userWeight,
↪  weightAllocated));
```

5.5.6 Lack of upgradeability across core contracts

Severity: Informational

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

Description: While certain components (e.g. the `InfiniFiGatewayV1`) will be behind an upgradeable proxy, many core infiniFi contracts such as the `UnwindingModule`, the `LockingController` and others are immutable. In the event of a serious logic bug or precision error the protocol could face indefinite asset lockups or inaccurate reward/penalty distribution. Relying on redeployment or manual data migration for large contracts with extensive storage is cumbersome and risky. Moreover, users already place trust in infiniFi governance, since privileged roles can alter core parameters or forcibly reallocate funds. Making all contracts upgradeable adds no further centralization risk beyond what already exists, but it grants the flexibility to swiftly address critical vulnerabilities or unexpected integration issues (e.g., with Aave, Pendle or other external yield sources).

Recommendation: Adopt an upgradeable pattern for all the contracts, so governance can rapidly deploy fixes for urgent logic or arithmetic flaws.

infiniFi: Acknowledged.

Spearbit: Acknowledged.

5.5.7 Incorrect user variable in `PositionRemoved` event

Severity: Informational

Context: LockingController.sol#L277

Description and Recommendation: In the `increaseUnwindingEpochs()` function of `LockedController`, the `PositionRemoved` event incorrectly emits `msg.sender` as the second parameter. Since `msg.sender` is always the gateway in this context, it should instead emit `_recipient`, which represents the actual function caller. The correction should be:

```
- emit PositionRemoved(block.timestamp, msg.sender, receiptTokens, _oldUnwindingEpochs);
+ emit PositionRemoved(block.timestamp, _recipient, receiptTokens, _oldUnwindingEpochs);
```

infiniFi: Fixed in 6e65d8a.

Spearbit: Fix verified.

5.5.8 Not all mint related functions return `receiptTokens` minted

Severity: Informational

Context: InfiniFiGatewayV1.sol#L75, InfiniFiGatewayV1.sol#L89

Description: `InfiniFiGatewayV1.mint` and `InfiniFiGatewayV1.mintAndStake` return the number of `receiptTokens` minted. Conversely, `InfiniFiGatewayV1.mintAndLock` does not.

Recommendation: Consider if the other token handling (mint/burn) functions in the gateway would benefit from a relevant return value.

infiniFi: Fixed in 52e9c19.

Spearbit: Fix verified.

5.5.9 Consider separating pause role from unpause role

Severity: Informational

Context: CoreControlled.sol#L59

Description: The protocol may want to separate out the pause and unpause role. A separate pauser allows introduction of monitoring triggered pausing leaving unpause to a more secure account.

infiniFi: Fixed in edc7b2e.

Spearbit: Fix verified.

5.5.10 Constructor function calls are `noops`

Severity: Informational

Context: Timelock.sol#L18-L19

Description: Due to having overridden `_setRoleAdmin` and `_revokeRole` to be `noops` the two functions calls in the constructor have no effect.

infiniFi: Fixed in c239314b.

Spearbit: Fix verified.

5.5.11 Limit total number of farms to eliminate DoS possibility

Severity: Informational

Context: FarmRegistry.sol#L87

Description: Be caution of array length related to farms as other contracts rely on reading the entire array, e.g. `getTypeFarms` called from `AllocationVoting`. Adding farms is governance controlled reducing dos possibility and making this information only.

infiniFi: Acknowledged, won't fix.

Spearbit: Acknowledged.

5.5.12 Consider reverting instead of `noop` on `MintController._deposit`

Severity: Informational

Context: MintController.sol#L106

Description: Reverting is recommended as a deposit would seem to be an error from calling `FarmRebalancer.singleMovement`.

infiniFi: Acknowledged. No fix implemented.

Spearbit: Acknowledged.

5.5.13 Typo in comment

Severity: Informational

Context: AllocationVoting.sol#L46

Description: Replace `comitted` with `committed`.

infiniFi: Fixed in 268c676.

Spearbit: Fix verified.

5.5.14 Add oracle price validation

Severity: Informational

Context: Accounting.sol#L29

Description/Recommendation: `Accounting.price` should defend against unexpected values. Above or below a particular threshold (e.g. 0) would suggest something is broken on the oracle side. A revert would prevent damage and allow the admin time to set a new oracle if needed.

infiniFi: Acknowledged. No fix implemented.

Spearbit: Fix verified.

5.5.15 Emit events on state changing functions

Severity: Informational

Context: Accounting.sol#L33, Accounting.sol#L38, FixedPriceOracle.sol#L15, YieldSharing.sol#L82, YieldSharing.sol#L87, YieldSharing.sol#L101, YieldSharing.sol#L106, MintController.sol#L46, MintController.sol#L52, MintController.sol#L57, RedeemController.sol#L46, RedeemController.sol#L52, InfiniFiGatewayV1.sol#L37, LockingController.sol#L75, LockingController.sol#L88, UnwindingModule.sol#L136, UnwindingModule.sol#L172, UnwindingModule.sol#L207

Description: Add events for the functions noted.

infiniFi: Fixed in PR 104.

Spearbit: Fix verified.

5.5.16 Ensure`GOVERNOR` is moved to a timelocked multisig

Severity: Informational

Context: InfiniFiCore.sol#L17

Description: Project confirmed deploy will use an EOA, then assign the role to a secure account afterwards. This step will be important to confirm complete as part of the deploy process.

Alternatively, an on chain deployment could eliminate the reliance on an EOA for deploy and config related permissions. See Tea Token's `TokenDeploy.sol`.

infiniFi: This role will be revoked in the deployment script.

Spearbit: Acknowledged.

5.5.17 `assetFarmTypes` does not conform to variable naming convention in `FarmRegistry`

Severity: Informational

Context: FarmRegistry.sol#L28

Description/Recommendation: The `assetFarmTypes` variable name doesn't match the convention used in this contract. e.g. the getter is `getAssetTypeFarms`. Renaming to `assetTypeFarms` would be consistent.

infiniFi: Fixed in commit 8eebe5cf.

Spearbit: Fix verified.

5.5.18 Consider adding `assetToken` validation in `AaveV3Farm.constructor`

Severity: Informational

Context: AaveV3Farm.sol#L21

Description/Recommendation: Could add similar `assetToken` verification to what the `ERC4626Farn` does in its `constructor`.

infiniFi: Acknowledged. We think we can safely go without it. Adding asset token will correspond to adding new set of controllers, updating registry, etc. so auditing individual asset tokens will always take place.

Spearbit: Acknowledged.

5.5.19 `getAssetVoteWeights` Does Not Check for Disabled Assets

Severity: Informational

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

Description: The `getAssetVoteWeights` function in `AllocationVoting` does not verify whether the asset is enabled in the `FarmRegistry` contract before proceeding with calculations. This means that if an asset has been disabled but `getAssetVoteWeights` is called from `AfterMintHook` or `BeforeRedeemHook`, it could still attempt to process the asset and find an optimal farm.

Although this scenario seems unlikely, as the system appears to primarily work with USDC, it is worth considering whether disabled assets could still be processed incorrectly. If this scenario is possible, it could result in unnecessary computations or unintended behavior when interacting with farms of disabled assets.

Recommendation: If this scenario can occur, consider checking whether the asset is enabled in `FarmRegistry` before performing any further calculations. If the asset is disabled, the function could return empty values early to avoid unnecessary processing. This can be achieved by adding a check such as:

```
if (!FarmRegistry(farmRegistry).isAssetEnabled(_asset)) {
    return (new addressw uint256 );
```

This would ensure that disabled assets do not interfere with downstream logic in `AfterMintHook` or `BeforeRedeemHook`, improving the robustness of the system while avoiding unnecessary calculations.

infiniFi: This is not an issue on the voting part. This is responsibility of the caller to verify this. Will add appropriate consumers of this method to verify if the assets are enabled as an extra step. This is the responsibility of controllers as they accept deposits of certain assets. Since voting results generally are very internal, this is not an issue of any significance, since any time an asset is disabled, corresponding controllers will be paused as well.

Fixed the controller side on `BeforeRedeemHook.sol` and `AfterMintHook.sol` on 56581b2.

Spearbit: As indicated by the infiniFi team, the responsibility relies on the caller. Thus, the hooks added this check. Fix verified.

5.5.20 `revokeRole` and `renounceRole` Not Overridden to Ensure Governor Role Persistence

Severity: Informational

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

Description: The `InfiniFiCore` contract inherits from `AccessControl` but does not override the `revokeRole` and `renounceRole` functions. Since `InfiniFiCore` is a key contract for access control across the protocol, the absence of these overrides may allow scenarios where the last `GOVERNOR` role is unintentionally revoked or renounced, leaving the contract without an administrator.

Ensuring that at least one `GOVERNOR` role exists at all times is important to maintain governance control. If no `GOVERNOR` is present, administrative actions, including role management, may become impossible without external intervention.

Recommendation: Consider overriding `revokeRole` and `renounceRole` to prevent the last `GOVERNOR` from being removed unintentionally. This would safeguard against losing governance control while maintaining flexibility for role management.

infiniFi: This is intentional in order to allow protocol to become fully decentralized at some point. It is a part of the roadmap and we hope to be able to renounce the Governor role at some point.

Spearbit: Acknowledged by infiniFi team. This can be a potential point for user trust and protocol decentralization.

5.5.21 Oracle System Is Not Optimized for Non-Stablecoin Assets

Severity: Informational

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

Description: The protocol is originally designed to work with USDC and potentially other stablecoins, but the current oracle system may not support volatile assets effectively. The `FixedPriceOracle` relies on a manually set price, which is suitable for stablecoins but would require frequent updates if used with assets like WETH or other fluctuating tokens.

The `Accounting` contract interacts with the oracle, indicating that the system depends on oracle-reported values for calculations. However, the `RedemptionPool` contract does not currently utilize an oracle, and there are no strong indications that the system is actively prepared to handle volatile assets. While some parts of the protocol may reference future multi-asset support, no clear mechanisms exist to update asset prices dynamically for volatile tokens.

Recommendation: If the protocol intends to support assets beyond stablecoins, consider implementing a more dynamic oracle system that can fetch real-time prices, such as Chainlink or Uniswap TWAPs. Additionally, reviewing how `Accounting` and `RedemptionPool` interact with pricing mechanisms could help prevent unintended behaviors when handling non-stable assets.

infiniFi: Volatility is not a concern for our system as it does not rely on conversions between multiple assets that are different in value. If we need to support ETH, we will deploy a new instance of infiniFi and not try to have it together with the USDC.

Spearbit: Acknowledged by infiniFi team.

5.5.22 System Relies on USDC Peg Stability and May Not Handle Severe Depegging

Severity: Informational

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

Description: The protocol assumes that USDC remains stable and includes a buffer mechanism in the `YieldSharing` contract to absorb minor fluctuations. This provides some level of resilience against small deviations from the peg. However, in the event of a severe or prolonged depeg, the current system may not be fully equipped to handle the situation effectively.

While the `YieldSharing` contract does contain mechanisms related to depegging, it is unclear whether these would be sufficient for extreme cases where USDC loses a significant portion of its value. If such an event were to occur, it could impact the overall stability of the protocol and the security of deposited funds.

Recommendation: Acknowledging this design consideration is important. If further mitigation is desired, additional safeguards could be explored, such as automated depeg detection mechanisms, dynamic risk adjustments, or integration with decentralized price feeds to react to significant price changes more effectively.

infiniFi: The system does not care about the current value of the `USDC`. It only tracks the conversion ratio between `iUSD` and `USDC` which changes only under most extreme conditions. For the same reason we are able to use this system to plug in ETH into it. As we do not care about the price movement of ETH, we would only care about the difference between ETH deposited and ETH earned.

Spearbit: As explained by infiniFi team, `iUSD` does not aim to peg a stablecoin value. And thus all its derivatives are susceptible to price fluctuation due to underlying depeg. This is a system design constraint.

5.5.23 Centralization Risk Due to Governor-Controlled Arbitrary Calls

Severity: Informational

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

Description: Most contracts in the protocol inherit from `Farm`, which itself inherits from `CoreControlled`. The `CoreControlled` contract contains a function that allows the `GOVERNOR` role to call any contract. While this is an intentional design choice to provide flexibility and emergency control, it also introduces a centralization risk.

If the Governor role or the Role Access Control (RAC) system were to be compromised, an attacker could use this functionality to execute arbitrary calls, potentially leading to unauthorized fund transfers, modifications of critical protocol parameters, or disruptions in protocol operations.

Additionally, the `Accounting` contract, which is integral to the protocol's calculations, inherits from `Farm`, meaning that governance actions could influence how funds and rewards are managed.

Recommendation: This is an inherent trade-off between control and decentralization.

infiniFi: Intentional system design. Can be prevented by renouncing `Governor` role in the future.

Spearbit: Acknowledged by infiniFi team.

5.5.24 Risk of type casting overflow in `RedemptionQueue`'s `uint96` request amount

Severity: Informational

Context: RedemptionPool.sol#L72

Description: In the redemption queue logic (e.g., `RedemptionQueue.RedemptionRequest`), the contract stores each request's amount of receipt tokens in a `uint96`. This allows for a maximum value of roughly `7.92e28` ($2^{96}-1$). Under normal circumstances where 1 receipt token starts at 1 USD, this is more than enough. However, even if very unlikely, receipt token price can drop significantly due to slashes and this could lead to scenarios where 1 unit of an asset is worth more than 2^{96} in receipt tokens. At that point, the request amount cannot be accurately recorded in a `uint96` without overflow or forced truncation. Any request beyond this boundary might revert or silently corrupt the redemption data, preventing large redemptions or creating partial leftover conditions that never settle.

Recommendation: Use a bigger storage type such as `uint256` for storing redemption request amounts. While it takes slightly more gas, it ensures safe handling of extreme price scenarios, especially if the protocol grows to support highly volatile assets or experiences major slashes.

infiniFi: Partially fixed in 214c69e by adding a check to enforce we are not redeeming more than `type(uint96).max` to avoid casting losses.

Spearbit: Be aware that the current partial fix does not really change anything as if the receipt token amount ever exceed `type(uint96).max` a DoS would still occur.