BAIL security

InceptionLRT
Vaults

FINAL REPORT

April ‘2025

Disclaimer:

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	InceptionLRT - Vaults
Website	www.inceptionlrt.com
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/inceptionlrt/smart-contracts/tree/0297c93e65839b58d158e2f6fa8e85006c07c9d2/projects/vaults
Resolution 1	2nd Audit round – see below

Please be aware that this audit is conducted on the as-is codebase.

While some issues may be corresponding to upgradeability concerns, any potential issues which may arise from upgrading the codebase on-chain are not covered by this audit.

We recommend proper testing before upgrading as well as a migration consultation. This audit does not include any migration process.

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	7	4	1	2	
Medium	13	11		2	
Low	9	7	1	1	
Informational	13	8		5	
Governance					
Total	42	30	2	10	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium-level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless, the issue should be fixed immediately.
Informational	Effects are small and do not pose an immediate danger to the project or users.
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior.

3. Detection

InceptionAssetsHandler

InceptionAssetsHandler is an abstract contract that manages the ERC20 asset (Liquid Staking Token) used in the vault. It provides internal functions to safely transfer tokens between arbitrary addresses and the contract, with optional hooks for token-specific behaviors like deflationary transfers, and balance tracking.

Privileged Functions

- None

No issues found.

AdapterHandler

AdapterHandler inherits from InceptionAssetsHandler and serves as the primary coordination layer between the vault and external protocols. It manages delegation, undelegation, and withdrawal operations via a controlled registry of adapter contracts.
The contract maintains a dynamic set of approved adapters to which funds can be delegated. Its standard operational flow involves delegating user deposits to adapters, and when there are pending withdrawal requests in the WithdrawalQueue, it orchestrates undelegations from the relevant adapters. Once any adapter-specific withdrawal delay has passed, it facilitates the redemption of those funds back to the vault.
In scenarios where enough idle funds (free balance) are available in the vault, AdapterHandler can skip the undelegation process and fulfill user withdrawals directly, improving efficiency and reducing latency.
The contract also includes emergency paths, allowing for undelegation or asset reallocation between adapters, without the need of users' withdrawal requests.
Additionally, it continuously tracks flash loan capacity, free balance, and a configurable target capacity. These parameters enforce constraints around delegation and force-undelegation operations to maintain solvency and be as close to the targetCapacity as possible.

Appendix: Total Deposited

- The **Total Deposited** amount represents the amount deposited across all strategies. It is calculated in the following formula (this is the corrected formula, not the original one):

totalDeposited = getTotalDelegated() + totalAssets() + getTotalPendingWithdrawals() + getTotalEmergencyWithdrawals() - redeemReservedAmount - depositBonusAmount()

  - getTotalDelegated() - Returns the total amount deposited across all strategies
  - totalAssets() - Returns the current total balance of the vault of the underlying asset.
  - getTotalPendingWithdrawals() - Returns total pending withdrawals across all adapters
  - getTotalPendingEmergencyWithdrawals() - Returns total pending emergency withdrawals across all adapters
  - redeemReservedAmount() - returns the amount reserved for redemptions. These are assets that have been withdrawn, but have not been yet claimed by the users and are part of the InceptionVault's balance.
  - depositBonus - the amount reserved for the depositBonus

Appendix: Flash Capacity, Target Capacity and Free Balance

- The **Flash Capacity** of the vault represents the total amount of liquidity currently available for flash loans. It ensures that only unallocated assets are used, keeping sufficient reserves for withdrawals and deposit bonuses. The flash capacity is calculated using the following formula:

flashCapacity = totalAssets() - (redeemReservedAmount() + depositBonus)

  - totalAssets() - Returns the current total balance of the vault of the underlying asset.
  - redeemReservedAmount() - returns the amount reserved for redemptions
  - depositBonus - the amount reserved for the depositBonus

- The **Target Capacity** defines the desired level of assets that the vault should maintain as available liquidity for flash loans. The value is calculated using the following formula:

  _getTargetCapacity = (targetCapacity * getTotalDeposited()) / 100e8

  - targetCapacity - A configurable percentage (with 1e8 precision) set by the contract owner to indicate the target flash capacity relative to total deposits

- The **Free Balance** represents the available balance for new deposits. It is calculated using the following formula:

  freeBalance = flashCapacity - targetCapacity

  - Alternatively if the flashCapacity is less than the targetCapacity, the free balance will be 0.

Appendix: Delegation

Delegation is the core mechanism that puts user capital to work—transforming passive deposits into active positions that earn rewards.
In the context of Inception Vaults, delegation refers to the process of moving user-deposited assets from the vault's custody into external restaking protocols.

Appendix: Undelegate and Claim

Undelegation is the process of pulling assets back from external protocols into the vault, preparing them for user withdrawal. This happens when users request to exit or redeem their positions.
Once undelegation is triggered, the vault begins retrieving assets from various adapters and protocols where they were previously deployed. These assets go through a processing phase, depending on the underlying protocol's mechanics.

After the undelegation process is finalized, claiming is the next step. Claiming makes the withdrawn assets officially available for redemption by users. This step confirms that the funds have returned and are now safely in the vault's custody.

Appendix: Emergency Functions

emergencyUndelegate allows the operator to initiate an emergency undelegation from multiple adapters. This can be used for rebalancing purposes, when closing an adapter or other emergency situations.
emergencyClaim follows up by finalizing the withdrawal process from the affected adapters, allowing the retrieved assets to be reclaimed into the system.

Core Invariants:

INV 1: Can only delegate to registered adapters.
INV 2: Cannot delegate more than the freeBalance.
INV 3: Adapters must be contracts.
INV 4: Only the Operator can control delegations and withdraws

Privileged Functions

- setTargetFlashCapacity
- addAdapter
- removeAdapter

Issue_01	getTotalDeposited does not account for pendingEmergencyWithdrawals and the redeemReservedAmount
Severity	High
Description	Currently, the getTotalDeposited function returns the following: *return getTotalDelegated() + totalAssets() + getTotalPendingWithdrawals() - depositBonusAmount;* However there are two issues in this formula: 1) totalAssets() still includes the redeemReservedAmount. This is incorrect as these assets are reserved for redemption, these are assets that have been withdrawn, but have not been claimed by the users, so they shall not be included in the total deposits. 2) The current equation does not account for the pendingEmergencyWithdrawals. This will result in incorrectly calculating the targetCapacity and FreeBalance, leading to incorrect delegation and undelegation restrictions.
Recommendations	Consider adjusting the formula to properly reflecting all variables.
Comments / Resolution	Fixed - getTotalDeposited() now considers all inactive balances (pendingWithdrawals, emergencyWithdrawals, claimable), and discounts the reserved redeemable assets.

Issue_02	Operator can use emergencyClaim to finalize normal undelegations
Severity	**Medium**
Description	The emergencyClaim function allows the operator to finalize a normal undelegation flow. This is extremely dangerous as it would temporarily prevent this withdrawal epoch from being finalized on time. This would require the operator to call emergencyUndelegate again, or if the delegated assets are not enough, delegate more assets so they are sufficient to be able to initiate the withdrawal process for the protocol again, which would drastically increase users' waiting time.
Recommendations	The emergencyClaim function should be restricted to be used only after emergencyUndelegate. Alternatively, we recommend acknowledging this issue.
Comments / Resolution	Fixed - emergencyClaims are restricted for only emergencyUndelegates.

Issue_03	Missing functionality to migrate the depositBonus
Severity	**Medium**
Description	In the unlikely event of having to migrate assets from the vault, there is no functionality to retrieve the depositBonus. This may result in depositBonus being stuck in the contract.
Recommendations	Consider adding an onlyOwner function to retrieve the depositBonus in case of emergency.
Comments / Resolution	Fixed - Implemented a function to migrate depositBonus

Issue_04	removeAdapter does not check if there is any deposited amount in the adapter
Severity	**Informational**
Description	The removeAdapter function does not check if there are any delegated or pendingWithdrawals. This could cause issues, especially if there are any pendingWithdrawals as it will be impossible to finalize users' withdrawal requests.
Recommendations	The removeAdapter should ensure there are no pending withdrawals or deposited amounts in the adapter prior to removing it. Alternatively, we recommend acknowledging this issue as it can be fixed by adding the adapter back.
Comments / Resolution	Fixed - Implemented recommended mitigation to validate there are no deposits (delegated, pendingWithdrawals or claimable) on the adapter.

Issue_05	Unnecessary check in removeAdapter
Severity	**Informational**

Description	When an adapter is added the following validations are made:
	
	function addAdapter(address adapter) external onlyOwner {
	    if (!Address.isContract(adapter)) revert NotContract();
	    if (_adapters.contains(adapter)) revert AdapterAlreadyAdded();
	
	The isContract check ensures that the adapter has a non-zero bytecode. However, when an adapter is removed, the same validation is performed:
	
	function removeAdapter(address adapter) external onlyOwner {
	  >>> if (!Address.isContract(adapter)) revert NotContract();
	    if (!_adapters.contains(adapter)) revert AdapterNotFound();
	
	This check is redundant, because it was already performed in addAdapter.

Recommendations	Remove the *isContract()* check from the removeAdapter()
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_06	Emergency functions cannot be used during paused state
Severity	**Informational**
Description	The AdapterHandler implements emergencyUndelegate and emergencyClaim, however these functions are also restricted with a whenNotPaused modifier, so they cannot be accessed in emergency situations.
Recommendations	Determine if this is the intended behavior and remove the whenNotPaused modifiers from the emergency functions if necessary.
Comments / Resolution	Acknowledged.

InceptionVault_S

InceptionVault_S is the core user-facing vault in the Inception protocol. It inherits from AdapterHandler and exposes the full ERC4626 interface for interacting with the vault. Users deposit liquid staking tokens (LSTs) and receive Inception tokens in return based on the ratio reported by the ratioFeed.

The vault supports two withdrawal paths:

- **Standard withdrawal** – triggers a delayed redemption through the WithdrawalQueue, after funds have been undelegated and received by the vault, users will call redeem to receive their LSTs back.
- **Flash withdrawal** – enables instant exit but incurs a dynamic fee.

The contract integrates with the InceptionLibrary to calculate utilization-based deposit bonuses and flash withdrawal fees. These mechanisms help maintain optimal vault liquidity and incentivize balanced usage.

Appendix: Deposit and Mint

The deposit and mint functions allow users to receive shares in exchange for their assets. Unlike standard ERC-4626 vaults, this implementation utilizes a ratio() provided by the ratioFeed to determine the exchange rate. Additionally, the InceptionVault includes a deposit bonus mechanism designed to incentivize deposits when the vault's flash capacity falls below the targetCapacity. This bonus is applied by overestimating the value of the user's supplied assets, resulting in the minting of additional shares.

Appendix: Ratio Feed

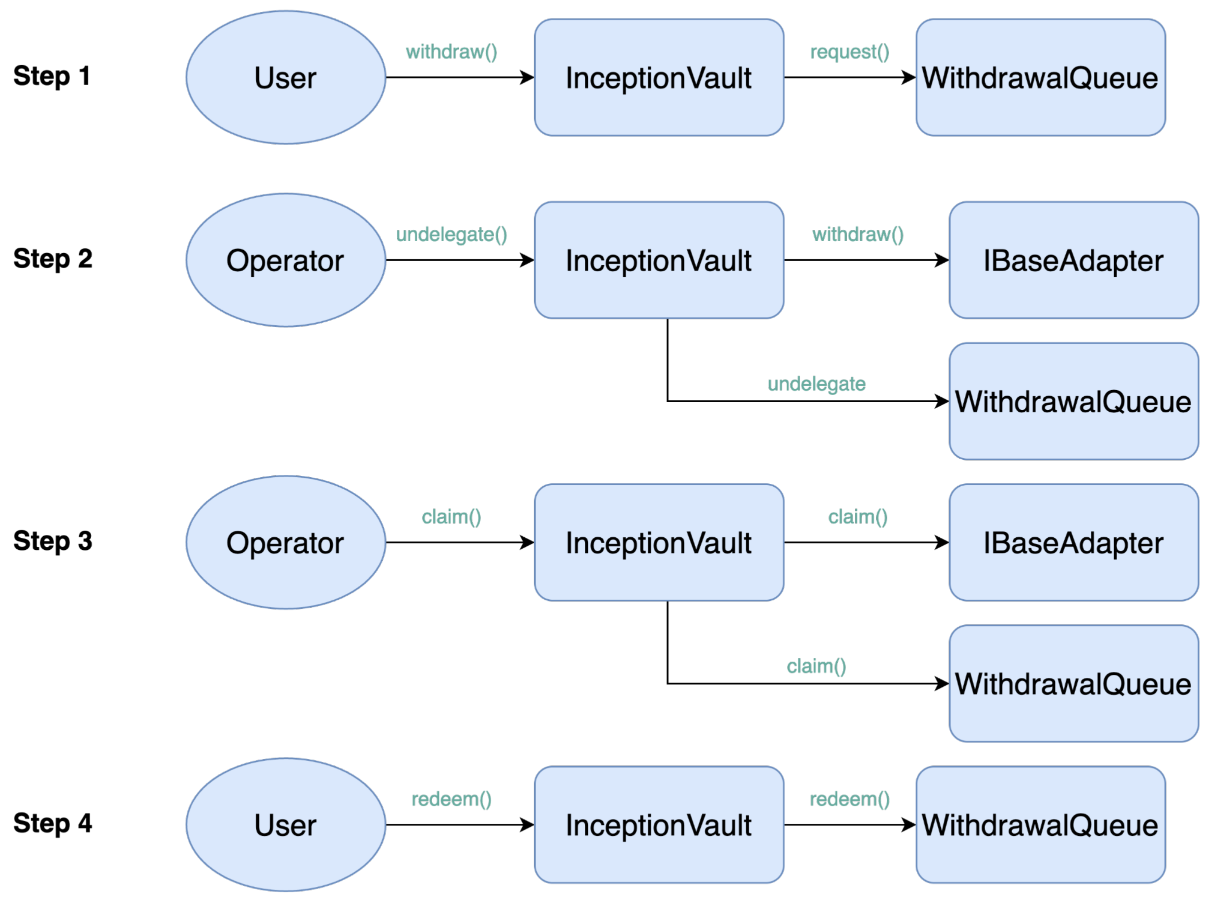
The ratioFeed contract is the protocol's own data feed for fetching the most recent ratio from the getRatioFor() function. The ratio is going to be updated by protocol's own backend server or on every slashing event. The getRatioFor() function will return the ratio in 1e18 precision as a result of the following formula:

Ratio = (InceptionTokenSupply + totalSharesToWithdraw) / (totalAssets() + getTotalDelegated() + getTotalPendingEmergencyWithdrawals() + depositBonusAmount - redeemReservedAmount);

- InceptionTokenSupply - minted shares
- totalSharesToWithdraw - requested shares to withdraw from users
- totalAssets() - asset balance of vault
- getTotalDelegated() - how many asset tokens are delegated to external protocols
- getTotalPendingEmergencyWithdrawals() - amount of queued emergency withdrawals assets from protocols
- redeemReservedAmount - amount of claimed assets available for users' redemptions

Appendix: Normal Withdrawals

The withdrawal process in the InceptionVault follows a unique multi-step flow before it is finalized:



1) **Request Creation**: A user initiates a withdrawal by calling the withdraw function. This creates a request in the WithdrawalQueue, which is grouped into an epoch along with other pending requests.

2) **Undelegation**: Every two days, the operator triggers the undelegate() function. This, in turn, calls IBaseAdapter::withdraw to initiate the withdrawal process from the external protocol. Multiple adapters may be utilized during this step. The WithdrawalQueue::undelegate function also starts a new epoch and records the shares and undelegated amounts associated with the previous one.

3) **Finalization**: After the required withdrawal delay has passed, the operator calls claim(). This finalizes the withdrawal by invoking IBaseAdapter::claim on each adapter and marks the corresponding withdrawal epoch as complete.

4) **Redemption**: Once the above steps are completed, users can call redeem() at any time to receive their assets.

Appendix: Flash Withdrawals

A **Flash Withdrawal** allows users to bypass the standard withdrawal delay by calling the flashWithdraw() function. While this provides immediate access to assets, it incurs a fee—known as the flashWithdrawalFee. This fee is split between the protocol and the deposit bonus, helping to incentivize future deposits. Flash Withdrawals can only be executed when there is sufficient flashCapacity available to cover the requested amount.

Core Invariants:

INV 1: Users cannot deposit less than the depositMinAmount.
INV 2: depositBonus can only be received when depositing below the targetCapacity.
INV 3: All asset/share conversions within the vault rely on the current ratio returned by the ratioFeed contract.
INV 4: Every deposit must result in minting a non-zero amount of Inception tokens (shares).
INV 5: flashWithdraw() is only permitted when the requested amount is within the current flashCapacity.
INV 6: flashWithdraw() will charge a higher fee the further the vault utilization is below the optimal capacity.

Privileged Functions

- setDepositBonusParams
- setFlashWithdrawFeeParams
- setProtocolFee
- setTreasuryAddress
- setRatioFeed
- setOperator
- setWithdrawMinAmount
- setDepositMinAmount
- setFlashMinAmount
- setName
- setWithdrawalQueue
- pause
- unpause

Issue_07	An attacker can front-run setWithdrawalQueue and steal funds from the vault
Severity	**High**
Description	When the withdrawalQueue is changed, setWithdrawalQueue can be called, to update it with its new address. However, this is not safe, as an attacker could take advantage of the legacy settings in the withdrawalQueue and front-run the setWithdrawalQueue with a call to the redeem function. Consider the following scenario: 1) In T1, owner initializes the withdrawalQueue with legacy withdrawals including the attacker. In the same transaction the owner calls setWithdrawalQueue. 2) The attacker sees the owner's T1 in the mempool, so he front-runs it with T0, which calls the redeem function. 3) Now, since the new withdrawalQueue also includes the attackers legacy amount, the attacker will be able to redeem again.
Recommendations	We recommend pausing the protocol before initializing the withdrawalQueue contract. To ensure a safe update, it is also advisable to include a check in the setWithdrawalQueue function that verifies the protocol is in a paused state.

Comments / Resolution	Acknowledged. The Inception team will take the following steps to ensure secure withdrawalQueue migration: 1) The operator will call SymbioticHandler::updateEpoch and complete all currently pending withdrawals. 2)The owner will pause the vault and record all unclaimed withdrawals, if there is any unprocessed withdrawal, it will be caught during the recording phase and processed again, by unpausing and invoking the updateEpoch() function.

Issue_08	Users could front-run the ratioFeed slashing update
Severity	**Low**
Description	The ratio() function returns the most recent ratio that was uploaded by the protocol's trusted server. However, a malicious user could front-run the ratio update, and flashWithdraw their shares. This creates an unfair advantage for proactive users, leaving others to bear the entire loss.
Recommendations	The protocol could check for slashing in the flashWithdraw function, but this would require excessive refactoring. We recommend acknowledging this issue.
Comments / Resolution	Fixed - ratio calculations are now performed directly onchain

Issue_9	previewMint does not account for the depositBonus
Severity	Low
Description	The previewMint will return how many assets are required to mint the corresponding number of shares. However, it does not take the depositBonus into consideration, which will cause it to return more assets. Per the EIP-4626, previewMint "MUST return as close to and no fewer than the exact amount of assets that would be deposited in a mint"
Recommendations	The previewMint function should take the deposit bonus into consideration when calculating the required assets. Alternatively, we recommend acknowledging this issue, because fixing it would require introduction of unnecessary complexity.
Comments / Resolution	Acknowledged.

Issue_10	maxMint does not consider paused state
Severity	Low
Description	The maxMint function does not not account if the state of the protocol is paused. Per the EIP-4626, maxMint: "MUST factor in both global and user-specific limits, like if mints are entirely disabled (even temporarily) it MUST return 0."
Recommendations	If the contract is in paused state, return 0.
Comments / Resolution	Fixed - Implemented recommended mitigation.

Issue_11	maxDeposit should not rely on balanceOf()
Severity	**Low**

Description	The maxMint function returns the following:
	
	function maxDeposit(address receiver) public view returns (uint256) {
	return !paused() ? _asset.balanceOf(receiver) : 0;
	}
	
	However as per EIP-4626:
	maxDeposit "MUST NOT rely on balanceOf of asset".

Recommendations	Per EIP-4626 it should return 2 ** 256 - 1.
Comments / Resolution	Fixed - Implemented recommended mitigation.

Issue_12	Lack of Slippage Protection in the deposit() function
Severity	**Low**
Description	The InceptionVault_S contract is designed to handle asset deposits and withdrawals while minting Inception tokens based on a predefined ratio. It includes functions such as deposit(), mint(), which work together to transfer assets from users to the vault and issue tokens in return. However, the deposit() function lacks slippage protection. Currently, users can deposit assets without safeguards against slippage. This means the amount of assets a user intends to deposit may differ from the amount actually received due to market fluctuations or sandwich attacks. While the mint() function implements slippage protection, the absence of similar measures in the deposit() function creates an inconsistent user experience and exposes users to potential losses.

Recommendations	Adding a minOut parameter would make the vault non-compliant with EIP-4626. To maintain clarity, it's essential to document this behavior clearly and advise users to use flashWithdraw() directly instead of redeem().
Comments / Resolution	Fixed - Created new functions with the option to set a minOut for slippage protection.

Issue_13	previewRedeem does not comply with EIP-4626
Severity	**Informational**
Description	EIP states that previewRedeem may revert by the same reason that would cause \`redeem()\` to revert. However, in order to prevent a revert when *amount>capacity*, it caps the capacity to the amont: *uint256 flash = amount <= capacity ? capacity : amount;*
Recommendations	Revert when amount > capacity
Comments / Resolution	Acknowledged - Known issue if reverting for external integrators.

Issue_14	setDepositBonusParams and setFlashWithdrawFeeParams do not validate the curve params
Severity	**Informational**

Description	The setDepositBonusParams and setFlashWithdrawFeeParams will change key variables for the calculation of the deposit bonus and the flash withdrawal fee.
	However, these params are not sufficiently validated:
	`function setDepositBonusParams(uint64 newMaxBonusRate, uint64 newOptimalBonusRate, uint64 newDepositUtilizationKink)`
	`    external`
	`    onlyOwner`
	`  {`
	`    if (newMaxBonusRate > MAX_PERCENT) {`
	`      revert ParameterExceedsLimits(newMaxBonusRate);`
	`    }`
	`    if (newOptimalBonusRate > MAX_PERCENT) {`
	`      revert ParameterExceedsLimits(newOptimalBonusRate);`
	`    }`
	`    if (newDepositUtilizationKink > MAX_PERCENT) {`
	`      revert ParameterExceedsLimits(newDepositUtilizationKink);`
	`    }`
	`    if (newOptimalBonusRate > newMaxBonusRate) revert InconsistentData();`
	
	`    maxBonusRate = newMaxBonusRate;`
	`    optimalBonusRate = newOptimalBonusRate;`
	`    depositUtilizationKink = newDepositUtilizationKink;`
	
	Passing a wrong rate or a kink could result in a choppy or discontinuous curve, allowing for fee and deposit bonus manipulations.

Recommendations	We recommend enforcing some validations on the inputs of the setDepositBonusParams and setFlashWithdrawFeeParams , to ensure that the curve is continuous at capacity == optimalCapacity. Alternatively if these params are not going to be changed we

	recommend removing the setter functions or acknowledging the issue.
Comments / Resolution	Acknowledged.

InVault_S_E2

The InVault_S_E2 is an extension of InceptionVault_S that overrides the _getAssetWithdrawAmount and _getAssetReceivedAmount functions to properly support rebase tokens that transfer 1-2 wei less.

Privileged Functions

- No privileged functions

Issue_15	The overridden _getAssetWithdrawAmount and _getAssetReceivedAmount functions are unused
Severity	**Informational**

Description	The InVault_S_E2 overrides the InceptionAssetsHandler's _getAssetWithdrawAmount and _getAssetReceivedAmount:
	
	function _getAssetWithdrawAmount(uint256 amount) internal pure override returns (uint256) {
	    return amount + 2;
	  }
	
	  function _getAssetReceivedAmount(uint256 amount) internal pure override returns (uint256) {
	    return amount - 2;
	  }
	
	However those functions remain unused by the InceptionVault.

Recommendations	Consider implementing or removing these functions.
Comments / Resolution	Fixed - InVault_S_E2 contract has been removed.

Convert

The Convert library is used by the InceptionVault to perform Math operations and conversions. It features the following functions:

- saturatingMultiply() - a multiplication function using the simple formula:
  - c = a * b
  - c / a != b - A result check, which returns uint256.max in case of failures
- saturatingAdd() - a function for simple additions featuring the following:
  - c = a + b
  - c < a - if this is true, than this means there is a failure in the calculation so it returns uint256.max
- multiplyAndDivideFloor() – A utility function that performs multiplication followed by division with rounding down. It internally uses saturatingMultiply() and saturatingAdd() to handle potential rounding safely.
  - *saturatingAdd(saturatingMultiply(a / c, b), ((a % c) * b) / c)*
- multiplyAndDivideCeil() – Similar to the previous function, but performs rounding up after division. Useful when rounding down would result in underestimation.
  - *saturatingAdd(saturatingMultiply(a / c, b), ((a % c) * b + (c - 1)) / c)*

No issues found.

InceptionLibrary

The InceptionLibrary is a utility library responsible for dynamically calculating deposit bonuses and flash withdrawal fees in the InceptionVault_S. These calculations are based on the current utilization of available capacity, movement toward or away from a predefined optimal range (kink), and the size of the transaction being processed. It ensures that incentives and penalties scale proportionally based on system state, enabling dynamic fee structures to remain in the target capacity range within the vault.

Appendix: Deposit Bonus

The depositBonus is distributed in two areas - [0; optimalCapacity] and [optimalCapacity; targetCapacity].

If the current flash capacity is less than the optimal threshold, the bonus is calculated on the replenished portion of the deposit, defined as:

$$replenished \ = \ Min(amount, optimalCapacity - capacity)$$

This defines how much of the deposit helps restore the vault to its optimal state.

Next, the **bonusSlope** is calculated, which determines how aggressively the bonus tapers off as the vault approaches optimal utilization:

$$bonusSlope = \frac{maxDepositBonusRate - optimalBonusRate}{optimalCapacity \div targetCapacity}$$

This slope models a linear decrease in bonus as capacity increases from zero to optimal. Then, a **subtractor** is introduced to modulate the bonus based on the average capacity reached by the deposit:

$$subtractor \ = \frac{bonusSlope * (capacity + replenished \ \div \ 2)}{targetCapacity}$$

After that the bonus and the bonusPercent are calculated:

$$bonusPercent \ = maxDepositBonusRate \ - \ subtractor$$

$$bonus \ = \ (replenished * bonusPercent) \ / \ 100$$

After applying the depositBonus, the remaining amount and capacity are updated accordingly:

capacity += replenished

amount -= replenished

Any remaining deposit amount (after optimal capacity is reached) is considered for a **fixed bonus**, calculated using the optimal bonus rate:

$$replenished = Min(amount, targetCapacity - capacity)$$

$$bonus += (replenished * optimalBonusRate) / 100e8$$

Appendix: Flash Withdrawal Fee

The **flash withdrawal fee** is designed to discourage over-draining the vault's available capacity, especially when it's already low. The fee is dynamically calculated based on how much a withdrawal affects the flash capacity, using a two-tier model depending on the current utilization level:

1) When capacity is above the optimalCapacity:
   If the vault has more available flash capacity than the optimal threshold, a flat optimal fee rate is applied to the portion of the withdrawal that brings the capacity closer to optimal:

$$replenished = Min(amount, capacity - optimalCapacity)$$

$$fee += (replenished * optimalFeeRate) / MAX\_PERCENT$$

After applying the fee to this part, the remaining amount and capacity are updated accordingly:

$$amount -= replenished$$

$$capacity -= replenished$$

2) When capacity drops **below the optimalCapacity:**

Any remaining amount that reduces the vault's capacity further below optimal is subject to a dynamic fee, which increases as the vault gets more depleted.
First, a fee slope is calculated to scale the fee rate linearly:

$$feeSlope = \frac{(maxFeeRate - optimalFeeRate) * 1e18}{(optimalCapacity * 1e18 \, / \, targetCapacity)}$$

This defines how sharply the fee increases as the available capacity falls.

Then a subtractor is calculated, which dynamically increases the effective fee the closer the vault is to empty:

$$subtractor = \frac{bonusSlope * (capacity - replenished \div 2)}{targetCapacity}$$

The resulting **fee rate** applied to this portion of the withdrawal is:

$$feeRate = maxFeeRate - subtractor$$

$$fee \mathrel{+}= (amount * feeRate) \, / \, MAX\_PERCENT$$

If the calculated fee is too small (due to rounding), the logic ensures it's never zero by incrementing it.

Privileged Functions

- None.

Issue_16	Users can limit their fee exposure by splitting their withdrawal into smaller portions, paying less fee to the protocol
Severity	**High**
Description	The subtractor calculation uses the midpoint between the starting and the finishing capacity for the transaction. This is used to increase the fee when the capacity goes below the optimal capacity proportionally: *uint256 subtractor = (feeSlope * (capacity - amount / 2)) / targetCapacity;* The subtractor can be exploited by any user by splitting their withdrawal into small portions to keep the subtractor as high as possible, which will substantially decrease their fee up to 10-15%, which is a direct loss of revenue for the protocol.
Recommendations	In order to prevent this attack, consider introducing a withdrawal delay.
Comments / Resolution	Acknowledged.

Issue_17	Users can extract more depositBonus by splitting their deposit into smaller amounts
Severity	**Medium**
Description	The subtractor calculation uses the midpoint between the starting and the finishing capacity for the transaction. This is used to proportionally decrease user's deposit as the capacity gets closer to the optimalCapacity. *uint256 subtractor = (bonusSlope * (capacity + replenished / 2)) / targetCapacity;* This can be exploited by any user by splitting their deposit into small portions to keep the subtractor as low as possible, which will substantially increase their bonus.
Recommendations	In order to prevent this attack, consider introducing a deposit delay.
Comments / Resolution	Acknowledged.

Issue_18	Users can sweep a portion of the deposit bonus for free
Severity	**Informational**
Description	The deposit bonus is granted to users upon depositing. This can be gamed by users to extract a big portion of the deposit bonus for free. Consider the following scenario: 1) The capacity is below the optimal capacity 2) A user takes a flash loan and makes a deposit that will bring the capacity up to the targetCapacity, in order to extract the maximum deposit bonus possible. 3) After that the user slaps the received Inception tokens at a DEX to return the flash loan and keep the profit
Recommendations	We recommend acknowledging this issue.
Comments / Resolution	Acknowledged.

WithdrawalQueue

The WithdrawalQueue contract is a core component that manages user withdrawal requests in a batched and epoch-based manner. It allows the connected vault to enqueue user withdrawal requests through the request function, assigning them to the current epoch. Each request is tracked by user and epoch, enabling structured and efficient withdrawal processing.
The contract maintains internal accounting for total requested shares, claimed amounts, and pending redemptions. Once an epoch's funds are undelegated and claimed from external strategies (via adapters), the contract transitions the epoch to a redeemable state. Users can then redeem their share of the claimed assets using the redeem function, which are proportionally distributed based on individual share allocations.

Appendix: Backwards Compatibility

The WithdrawalQueue contract includes built-in support for backward compatibility through the _initLegacyWithdrawals function, enabling seamless migration of withdrawal requests from a previous system or vault version.
This mechanism ensures that users who had pending withdrawal requests before **the queue system upgrade** do not lose their positions or balances.

Appendix: Adapter Coordination

The undelegate() and claim() flows ensure all adapter–vault pairs involved in an epoch are processed only once. Per-adapter tracking is stored in adapterUndelegated, and counters (adaptersUndelegatedCounter, adaptersClaimedCounter) are used to gate finalization logic, requiring all adapters to be processed. Epochs can only be marked as redeemable after all undelegation and claim operations are completed.

Appendix: Epoch and User Epoch Management

The withdrawal queue operates on an **epoch-based system**, where each epoch groups user withdrawal requests into a collective batch for processing and undelegation. This system ensures efficient management of undelegation and claiming operations from multiple adapters.

- **Epoch Tracking**:
  Each withdrawal epoch maintains state related to the requested shares, undelegated amounts, and claimed assets. The currentEpoch value increments after the start of each undelegation round, marking the transition to a new batch of withdrawal requests.

- **User Epoch Mapping**:
  Every user address is mapped to a list of epoch indices (userEpoch) in which they have active withdrawal requests. This mapping allows the contract to efficiently determine when a user is eligible to redeem assets, without scanning the full epoch history.
- **Epoch Advancement**:
  The epoch advances only after the undelegation, or when forceUndelegateAndClaim() is explicitly called. This ensures that all user withdrawal requests within the current epoch are fully handled before enabling redemptions.
- **Efficient Redemption**:
  On redemption, the contract iterates through the user's epoch list and aggregates all redeemable amounts. Once processed, the corresponding entries are cleared to avoid re-processing and reduce gas usage in future calls.

Core Invariants:

INV 1: Only the vault is authorized to call sensitive queue functions (request, undelegate, claim, forceUndelegateAndClaim, redeem).
INV 2: Users cannot request a withdrawal with zero shares or to the zero address.
INV 3: An undelegation can only be processed for the current epoch
INV 4: An adapter-vault pair can be undelegated only once per epoch.
INV 5: A claim can only occur after the corresponding undelegation has been recorded.
INV 6: A claim amount cannot exceed the undelegated amount for that adapter-vault pair.
INV 7:An epoch becomes redeemable only when all expected claims for undelegated adapters are completed (adaptersClaimedCounter == adaptersUndelegatedCounter).
INV 8: currentEpoch is incremented on every undelegation

Privileged Functions

- None.

Issue_19	Slashing is not properly socialized
Severity	**High**
Description	When a user initiates a withdrawal from the InceptionVault, a request is made in the WithdrawalQueue. In the request function, totalRequestedShares and userShares are increased by the shares being withdrawn: *withdrawal.userShares[receiver] += shares;* *withdrawal.totalRequestedShares += shares;* After that, in order to process this withdrawal, the operator will call undelegate, passing an array of adapters to undelegate from, or alternatively supplying 0 adapters, indicating a direct use of the vault's freeBalance. For the finalization of the withdrawals, when the withdrawn amount becomes claimable, the operator calls claim, which fetches the claimedAmounts from each adapter: *claimedAmounts[i] = _claim(adapters[i], _data[i], false);* After that, these claimed amounts are used in the WithdrawalQueue to determine the totalClaimedAmount: *withdrawal.totalClaimedAmount += claimedAmount;* Then, the epoch becomes redeemable and users call InceptionVault_S::redeem(), which will calculate the assets using the following formula in _getRedeemAmount: *return withdrawal.totalClaimedAmount.mulDiv(* *withdrawal.userShares[receiver],* *withdrawal.totalRequestedShares, Math.Rounding.Down* *);* However this share conversion is incorrect, as it will not properly distribute slashing among all token holders. Consider the following scenario:

	1) At totalDeposited = 2000e18, adapterA.getTotalDelegated = 1000e18(adapterA is InceptionEigenAdapter), totalAssets = 1000e18, targetCapacity = 1000e18, totalShares = 2000e18 (1:1 ratio with the underlying assets) 2) A user calls withdraw with 200e18 shares which equals 200e18 assets. 3) The operator calls undelegate from adapterA, passing 200 assets. 4) During the withdraw delay at the delegationManager, the assets that are being withdrawn are still subject to slashing as it is stated in the natspec. For this example, a slashing of 50% will occur. This means that the ratio is now: ratio = (totalShares *1e18) / totalDeposited = (2000e18*1e18) /1500e18 = 1.33e18 (Here the totalDeposited dropped to 1500, which is 500 assets less due to the slashing). 5) User calls redeem which uses the aforementioned formula in the _getRedeemAmount: assets = (totalClaimedAmount * userShares) / totalRequestedShares = 100e18 (Here since the user is the only withdrawer he takes the whole totalClaimed, which is 100e18, instead of 200e18 due to the slashing) This means that the user redeemed at a rate of 2e18, instead of the rate of 1.33e18 by which he would get 150e18 tokens instead of the 100e18 he eventually received. Not socializing the slashing loss in such cases is unfair to users as the operator is responsible for choosing the adapters for the withdrawal epoch. Furthermore, users have no slippage protection, so they cannot do anything to prevent this unpleasant scenario, in which withdrawing users suffer the effect of the slashing more severely.
Recommendations	We recommend taking a snapshot of the actual ratio in the _claim function and using it to calculate the redeemed assets.

Comments / Resolution	Partially Fixed - Refactoring accounts for most of the cases when slashing can occur, but failed to consider an edge case involving the Mellow Adapters.

Issue_20	Normal undelegation flow is not optimized for certain cases
Severity	**Medium**
Description	In the _afterUndelegate function, the totalUndelegated amount has the following restrictions: *function _afterUndelegate(WithdrawalEpoch storage withdrawal) internal {* *uint256 requested = IERC4626(vaultOwner).convertToAssets(withdrawal.totalRequestedShares);* *uint256 totalUndelegated = withdrawal.totalUndelegatedAmount + withdrawal.totalClaimedAmount;* *require(* *requested >= totalUndelegated* *? requested - totalUndelegated <= MAX_CONVERT_THRESHOLD* *: totalUndelegated - requested <= MAX_CONVERT_THRESHOLD,* *UndelegateNotCompleted()* *);* This ensures that the undelegated assets are relevant to the ratio supplied by the ratioFeed. However, there will be cases in which the normal withdrawal flow will not work properly Consider the following scenario: 1) At totalDeposited = 2000e18, adapterA.getTotalDelegated = 1000e18(adapterA is InceptionEigenAdapter), totalAssets = 1000e18,

	targetCapacity = 1000e18(50%), totalShares = 2000e18 (1:1 ratio with the underlying assets) 2) A whale investor decides to withdraw 1001e18 shares. Due to the aforementioned restriction it will be impossible to start a normal undelegation. Furthermore, the operator will also be unable to do a undelegate with 0 adapters(forceUndelegateAndClaim). In this edge case, the operator will have to perform a series of operations to handle this: 1) emergencyUndelegate(1000e18) 2) emergencyClaim() 3) Still, in our case the FreeBalance is still not enough, the owner will need to call setTargetFlashCapacity() and change the targetFlashCapacity to 51%, so that the following check is passed: *if (getFreeBalance() < requestedAmount) revert InsufficientFreeBalance();* 4) undelegate() with 0 adapters to force a \`undelegaetAndClaim()\`
Recommendations	Refactor _undelegateAndClaim() as follows: *if (getFlashCapacity() < requestedAmount) revert InsufficientFreeBalance();*
Comments / Resolution	Fixed by applying the recommendation.

Issue_21	Users may accidentally loop too much and cause DOS
Severity	**Medium**
Description	WithdrawaQueue::redeem does a while loop on all user withdraw epochs. This can be dangerous as users may accidentally push a lot of epochs to their ` userEpoch` map causing a DOS error when they try to redeem. On top of that other users can also push `userEpoch` to any other user, by making a request for them, making the issue even more likely. For cases where some users haven't used the protocol in a while and come to a ton of little withdrawal requests made for them. Note that the chances of this bug are low, however its impact is significant as it blocks all withdrawals for that user.
Recommendations	Add the option for users to input epochs, in order to prevent this vulnerability from occurring.
Comments / Resolution	Resolved by following the recommendation.

Issue_22	_initLegacyWithdrawals shall not use initializer modifier
Severity	**Medium**
Description	The _initLegacyWithdrawals is to be called by the initialize function. However, the _initLegacyWithdrawals uses initializer modifier instead of onlyInitializing. According to the OpenZeppelin documentation, the onlyInitializing modifier should be used to allow initialization in both functions. The onlyInitializing modifier ensures that when the initialize function is called, any calls in its call chain can still complete their own initialization.
Recommendations	Consider using the onlyInitializing modifier instead of initializer.
Comments / Resolution	Resolved by following the recommendation.

Issue_23	isRedeemable will return wrong withdrawalIndexes
Severity	**Low**

Description	isRedeemable() should return the epoch indexes for the specified claimer for the withdrawalRequests that are ready to be redeemable.
	However, withdrawalIndexes data is filled up using the counter of the for loop, instead of filling it up with the actual epoch that is redeemable:
	
	for (uint256 i = 0; i < epochs.length; i++) {
	WithdrawalEpoch storage withdrawal = withdrawals[epochs[i]];
	if (!withdrawal.ableRedeem || withdrawal.userShares[claimer] == 0) {
	continue;
	}
	
	able = true;
	>>> withdrawalIndexes[index] = i;

Recommendations	Change *withdrawalIndexes[index] = i;* with *withdrawalIndexes[index] = epochs[i]*
Comments / Resolution	Resolved, now the function comments have been updated to properly describe the returned value.

InceptionToken

The InceptionToken is an ERC20Upgradeable token that acts as the Liquid Restaking Token (LRT) of the protocol. It represents proportional ownership in a specific vault and is minted or burned in response to user deposits and withdrawals. Each vault has a dedicated InceptionToken, enforcing a strict 1:1 relationship between tokens and vaults to maintain clear accounting boundaries.
The token includes pausable functionality, allowing transfers to be temporarily disabled by the owner.
Core Invariants:

INV 1: During paused state, transfers, minting and burning will revert
INV 2: Only the owner can pause and unpause
INV 3: The Inception Token (LRT) balance represents a user's proportional ownership of the underlying LST assets held by the vault.

Privileged Functions

- setVault
- pause
- unpause

No issues found.

IBaseAdapter

The IBaseAdapter is an abstract upgradeable contract that forms the foundation for adapter implementations within the InceptionLRT protocol. It standardizes shared logic, access control, and interface compatibility across different protocol integrations. The contract manages key roles, including the _trusteeManager (controlled by the governance), and the _inceptionVault (the primary vault interacting with the adapter).
It enforces strict access control through the onlyTrustee modifier, which allows only the vault or trustee manager to execute sensitive functions.

Privileged Functions

- setInceptionVault
- setTrusteeManager
- pause
- unpause

Issue_24	IBaseAdapter should add a storage gap
Severity	**Medium**
Description	The IBaseAdapter is a base contract to be inherited by the adapters of the Inception protocol. However, it does not implement a storage gap which could be problematic in case of future upgrades.
Recommendations	Consider adding a storage gap to the IBaseAdapter.
Comments / Resolution	Acknowledged.

Issue_25	__IBaseAdapter_init shall not use initializer modifier
Severity	**Medium**
Description	The IBaseAdapter is to be inherited by all the adapters of the Inception vault. However, the __IBaseAdapter_init uses initializer modifier instead of onlyInitializing. The problem here is that both the parent contract and the child contract are using the initializer modifier, which limits initialization to only one call. According to the OpenZeppelin documentation, the onlyInitializing modifier should be used to allow initialization in both the parent and child contracts. The onlyInitializing modifier ensures that when the initialize function is called, any contracts in its inheritance chain can still complete their own initialization.
Recommendations	Use onlyInitializing modifier instead of initializer.
Comments / Resolution	Fixed - Implemented recommended mitigation.

InceptionEigenAdapter

The InceptionEigenAdapter is a custom adapter contract designed to integrate with the EigenLayer protocol, enabling both delegation and strategy interactions on behalf of the Inception Vault. It supports two primary actions: delegating to an operator via EigenLayer's DelegationManager for AVS participation, and depositing into a strategy through the StrategyManager.
In addition to deposits and delegation, the adapter handles withdrawal flows by queuing exit requests from strategies, either normally or in emergency mode. These withdrawals are later claimed and transferred back to the vault.
The adapter also tracks the currently deposited amounts, as well as pending withdrawals and emergency withdrawals.

Appendix: Strategy

In EigenLayer, a strategy represents a yield-generating vault where users' restaked assets (such as stETH) are deposited. These strategies are managed by the **StrategyManager** contract and act as the underlying destinations for capital restaked through EigenLayer.
When a user (or adapter contract) deposits into a strategy, the asset (e.g., stETH) is converted into **strategy shares**, which track the depositor's proportional ownership.
Users cannot interact with these strategies directly; instead, interaction happens through contracts like **StrategyManager** or via delegation flows through **DelegationManager**, ensuring controlled access and slashing protection.

Appendix: StrategyManager

The StrategyManager plays a central role in the EigenLayer architecture by facilitating the secure deposit and withdrawal of tokens into various restaking strategies. Within the InceptionEigenAdapter, it acts as the primary interface for managing asset interactions with the selected IStrategy contract.
When the adapter receives a delegation request where the operator address is zero and a non-zero token amount is specified, it interprets this as a deposit into the strategy rather than a delegation to an EigenLayer operator. In this case, the adapter first transfers the specified amount of the underlying asset from the vault (caller) into itself. It then uses the StrategyManager to deposit those tokens into the designated strategy by calling depositIntoStrategy. This method handles the minting of strategy shares in return for the deposited assets, and the adapter uses sharesToUnderlying to translate those shares back into an equivalent token amount for accounting purposes.

On the withdrawal side, when a user initiates a withdrawal, the adapter again leverages the StrategyManager, this time to queue a withdrawal through the DelegationManager. It converts the amount to be withdrawn into the appropriate number of strategy shares via underlyingToShares and then prepares a QueuedWithdrawalParams object. This is submitted to EigenLayer, which begins the withdrawal process. Once the withdrawal is mature, the adapter finalizes it by calling completeQueuedWithdrawal, which transfers the underlying tokens back to the adapter for forwarding to the vault.

Appendix: DelegationManager

The DelegationManager is a key EigenLayer component that manages validator delegation and withdrawal lifecycle. In the InceptionEigenAdapter, it's used to delegate restaked assets to operators via the delegateTo() call, enabling participation in AVSs and earning additional yield.
For withdrawals, the adapter queues requests using queueWithdrawals() and later completes them with completeQueuedWithdrawal(), allowing secure asset recovery. This integration ensures proper delegation flow, slashing protection, and compatibility with EigenLayer's security mechanisms. While in the queue, the requested shares are removed from the staker's balance, as well as from their operator's delegated share balance (if applicable). Also, while in the queue, deposit shares are still subject to slashing.

Appendix: RewardsCoordinator

The RewardsCoordinator facilitates the claiming of rewards distributed by an AVS (Autonomous Verifiable Service). The flow is as follows: the AVS submits rewards to the RewardsCoordinator, after which a trusted off-chain rewardsUpdater calculates allocations and submits the resulting DistributionRoots on-chain. Once a root is active, the assigned claimer—configured by the InceptionEigenAdapter—calls processClaim, providing a valid Merkle proof against the posted DistributionRoot.

Privileged Functions

- setRewardsCoordinator

Issue_26	Missing functionality to undelegate from an operator and redelegate
Severity	**Medium**
Description	In the delegate function, the _delegationManager.delegateTo() is called to delegate the adapter's strategy deposits to a specific operator: *_delegationManager.delegateTo(operator, approverSignatureAndExpiry, approverSalt);* However it does not implement any functions to change the operator (via _delegationManager.redelegate()) or undelegate from an operator (via _delegationManager.undelegate()). Furthermore, the trustee cannot change the delegator by calling delegateTo due to the following check here: *function delegateTo(* *address operator,* *SignatureWithExpiry memory approverSignatureAndExpiry,* *bytes32 approverSalt* *) public nonReentrant {* *require(!isDelegated(msg.sender), ActivelyDelegated());*
Recommendations	Consider implementing undelegate() and redelegate(). It must be clear that the introduction of these new functionalities implements additional complexity and state transitions which can be prone to errors. Thus, we recommend alternatively to acknowledge this issue.
Comments / Resolution	Fixed - Added functions to undelegate and redelegate on the DelegationManager.

Issue_27	Missing functionality to distribute rewards from RewardsCoordinator
Severity	**Medium**

Description	In the InceptionEignenAdapter the setRewardsCoordinator function will set a claimer for the configured RewardsCoordinator:
	function _setRewardsCoordinator(address newRewardsCoordinator, address claimer) internal {
	
	IRewardsCoordinator(newRewardsCoordinator).setClaimerFor(claimer);
	
	However, there is no functionality for the claimer to allocate these rewards without causing a stepwise jump in the ratio.

Recommendations	Consider implementing an addReward function.
Comments / Resolution	Fixed - Added new claimRewards() to claim rewards from the rewardsCoordinator.

Issue_28	Some strategies have a deposit limit
Severity	**Informational**
Description	The InceptionEigenAdapter contract manages delegation and withdrawal requests within the EigenLayer protocol, interacting with components such as DelegationManager, StrategyManager to facilitate asset delegation and withdrawals. The delegate() function is responsible for depositing funds into a strategy. However, some strategies impose deposit limits (e.g. mellow wsteth), meaning that once a limit is reached, the operator cannot delegate additional assets. This restriction can result in funds remaining idle in the vault, generating no rewards for users.
Recommendations	Consider acknowledging this issue.

Comments / Resolution	Fixed - new function beforeDepositAssetIntoStrateg() reverts if deposit would make strategy's balance to go beyond maxTotalDeposits

Issue_29	Direct donations to InceptionEigenAdapter will reduce the delegatable amount.
Severity	**Informational**
Description	claimableAmount() is used in the inactiveBalance(), which will be returned in the getTotalPendingWithdrawals(), causing the getTotalDeposited to increase, which results in an increase of the targetCapacity preventing further delegations. This claimableAmount cannot be claimed due to missing functionality, so any amount donated to the InceptionEigenAdaper will reduce the delegatable amount.
Recommendations	Consider implementing a way to claim the claimableAmount.
Comments / Resolution	Fixed - Implemented recommended mitigation

InceptionEigenAdapterWrap

InceptionEigenAdapterWrap is a specialized variant of the InceptionEigenAdapter, tailored for use with Lido's **wstETH** (wrapped staked ETH). It introduces wrapping and unwrapping logic to seamlessly handle conversions between **wstETH** and **stETH**, allowing integration with EigenLayer's strategy-based delegation system while maintaining compatibility with wrapped assets.

Appendix: Lido's WstEth

WstETH is a wrapper contract for Lido's stETH token. It provides a non-rebasing ERC20 representation of stETH called **wstETH**, where balances remain constant and represent a share of the total stETH supply. Users can wrap stETH into wstETH, unwrap it back to stETH, or stake ETH directly into the contract to receive wstETH in return.

The adapter integrates with the wstETH contract by unwrapping tokens before delegating assets to EigenLayer, and wrapping them back into wstETH immediately upon claiming, ensuring compatibility with both systems.

Privileged Functions

- setRewardsCoordinator

Issue_30	Negative rebase of stEth may temporarily prevent claiming
Severity	**Medium**
Description	In the WithdrawalQueue::_claim function there is the following check, which requires the claimed amount to be less than the undelegated amount: *require(withdrawal.adapterUndelegated[adapter][vault] >= claimedAmount, ClaimedExceedUndelegated());* In the rare case of a negative stEth rebase, the InceptionEigenAdapter might temporarily return more wstEth, than the undelegated amount in the conversion here: *function claim(bytes[] calldata _data, bool emergency)* *external* *override* *onlyTrustee* *whenNotPaused* *returns (uint256)* *{* ... *return wrappedAsset().getWstETHByStETH(withdrawnAmount);* This will cause the aforementioned condition in the WithdrawalQueue::_claim function to revert the claiming, because the undelegated amount will be less than the claimed.
Recommendations	Consider removing the following check in WithdrawalQueue::_claim: *require(withdrawal.adapterUndelegated[adapter][vault] >= claimedAmount, ClaimedExceedUndelegated());*
Comments / Resolution	Fixed - Implemented recommended mitigation

Issue_31	Unnecessary approval to strategyManager
Severity	**Informational**
Description	In the intialize function the _asset is approved to the strategyManager: *function initialize(* *...* *) public initializer {* *...* *// approve spending by strategyManager* *_asset.safeApprove(strategyManager, type(uint256).max);* This approval is unnecessary, as the InceptionEigenAdapter wrap will not deposit wstEth, into the strategy. Instead, it will only deposit stEth.
Recommendations	Consider removing the redundant approval
Comments / Resolution	Acknowledged.

IMellowAdapter

The IMellowAdapter contract acts as a bridge between the InceptionVault and multiple Mellow Vaults, enabling delegation and withdrawal of assets. It allows the Inception vault to deposit funds into one or more Mellow Vaults, either manually or automatically, based on predefined allocation weights. Withdrawals are handled asynchronously via separate claimer contracts, which facilitate the secure transfer of undelegated funds back to the vault.

Appendix: EthWrapper

The EthWrapper contract is a utility that standardizes deposits into protocols by converting various ETH-based tokens into a single unified asset: **wstETH** (Wrapped Staked Ether). It accepts deposits in **ETH**, **WETH**, **stETH**, or **wstETH**, and wraps them accordingly into wstETH before forwarding them to a Mellow vault.

Appendix: Auto Delegation

Auto delegation in the IMellowAdapter enables seamless distribution of a deposit across multiple Mellow vaults based on their configured allocation weights. Instead of specifying a single target vault, the adapter can automatically split the deposited amount proportionally according to each vault's share in the allocations mapping.

Appendix: Mellow Symbiotic Vault

The MellowSymbioticVault is an ERC4626-compliant vault implementation that integrates with a Symbiotic Vault and Collateral system to optimize asset utilization and yield generation.
Upon deposit, the vault automatically routes assets into either collateral or the Symbiotic Vault, based on available capacity and predefined limits.
Withdrawals are served from local balance first, then collateral, and finally from the Symbiotic Vault via the withdrawalQueue if needed.

Core Invariants:

INV 1: Only vaults added by the owner can be used.

Privileged Functions

- addMellowVault
- changeAllocation
- setEthWrapper

Issue_32	A direct donation will DOS the claim on IMellowAdapter
Severity	**High**
Description	In the claim() function, the IMellowAdapter calls the claimer to finalize the claiming in the Mellow vault: *function claim(bytes[] calldata _data, bool emergency)* *external* *override* *onlyTrustee* *whenNotPaused* *returns (uint256)* *{* *...* *MellowAdapterClaimer(claimer).claim(_mellowVault, address(this), type(uint256).max);* *uint256 amount = _asset.balanceOf(address(this));* *if (amount == 0) revert ValueZero();* *_asset.safeTransfer(_inceptionVault, amount);* *return amount;* *}* Using *balanceOf(address(this))* can be exploited by an attacker by directly sending some amount to the IMellowAdapter, causing the following check in WithdrawalQueue::_claim to revert: *require(withdrawal.adapterUndelegated[adapter][vault] >= claimedAmount, ClaimedExceedUndelegated());*
Recommendations	Consider using the output amount from the claim call, instead of the balance of the adapter.
Comments / Resolution	Fixed - Implemented recommended mitigation.

Issue_33	All preview functions in the IMellowAdapter will return amount in wstEth
Severity	High

Description	The IMellowAdapter uses ethWrapper, which will convert any inputted asset to wstEth and deposit it into the Mellow vault. For example, if stEth is used as underlying asset by the InceptionVault, this will convert the stEth into wstEth and deposit it into the vault. However, all preview functions in the IMellowAdapter will return the quired amounts in wstEth.
	For instance, getTotalDeposited will call mellowVault.previewRedeem() which will return the assets in wstEth instead of stEth as expected:
	
	function getTotalDeposited() public view override returns (uint256) {
	    uint256 total;
	    for (uint256 i = 0; i < mellowVaults.length; i++) {
	      uint256 balance = mellowVaults[i].balanceOf(address(this));
	      if (balance > 0) {
	        total +=
	IERC4626(address(mellowVaults[i])).previewRedeem(balance);
	      }
	    }
	
	This will cause incorrect values used by the Inception Vault from the following functions: getTotalDeposited(), inactiveBalance(), inactiveBalanceEmergency(), delegate(). Also withdrawing and claiming will be impossible, due _asset being different from wstEth.

Recommendations	Consider implementing conversion from wstEth to _asset.
Comments / Resolution	Fixed - Adapter has been renamed to reflect it is only compatible with wstETH; For stETH or WETH, a separate adapter will be implemented.

Issue_34	pendingClaimers could cause OOG issues
Severity	**Medium**
Description	When withdrawing (undelegation) on the MellowAdapter, the MellowVault could have enough liquidity to process the full withdrawal. This means `MellowAdapter.withdraw()` would return (0, claimedAmount), marking `WithdrawalQueue._afterUndelegate()` the epoch as redeemable. This will result in the epoch skipping the claim() process, where the claimer is removed from the pendingClaimers, causing the pendingClaimers to grow indefinitely. As a result, pendingWithdrawalAmount and the claimableWithdrawalAmount could run into out-of-gas error, causing a DOS of the protocol.
Recommendations	During the withdrawal on the MellowAdapter, if all the undelegated amount was instantly withdrawn from the MellowVault (remaining amount to withdraw returned as 0), call `_removePendingClaimer()` to remove the claimer from pendingClaimers.
Comments / Resolution	Fixed - If all request amount to withdraw is fully withdrawn, the claimer is removed from pendingClaimers in the same call.

Issue_35	Farm rewards cannot be claimed
Severity	Low
Description	In the Mellow documentation it is stated that some rewards will be transferred to farms, which implement the IStakerRewards interface. However the IMellowAdapter does not have any way to call the claimRewards function. As a result these rewards cannot be claimed.
Recommendations	Consider implementing a way to claim these rewards.
Comments / Resolution	Fixed.

Issue_36	claim() should perform claimer validation
Severity	Low
Description	The claim() function allows the Operator to pass any data. In the IMellowAdapter this data is decoded into the _mellowVault and the claimer: *MellowAdapterClaimer(claimer).claim(_mellowVault, address(this), type(uint256).max);* However, the vault and the claimer are not verified, which could result in using the wrong claimer, causing less assets to be claimed for the epoch.
Recommendations	Consider validating the claimer and the vault using the claimerVaults.
Comments / Resolution	Fixed - Implemented validations to verify claimer and mellow vault.

Issue_37	Inability to remove Mellow vaults
Severity	**Low**
Description	Currently we can only add Mellow vaults to the IMellowAdapter, and each vault is looped over when we do any checks (`_beforeDelegate`), amount extractions (`getTotalDeposited`), in `_delegateAuto` and many other places. However with time some vaults will become obsolete, as either they will be no longer maintained by Mellow, hacked, abandoned or with just too low APY to be useful. Since we have no way to remove such vaults, we would be forced to loop over them every time in almost all of the functions. This will cause unnecessary gas consumption and possibly DOS if there are too many vaults.
Recommendations	Consider adding a function to remove vaults, but make sure we have withdrawn from the first.
Comments / Resolution	Partially Fixed - Implemented recommended mitigation.

Issue_38	inactiveBalance does not match the natspec comments
Severity	**Informational**
Description	In the natspec it is stated that the inactiveBalance() will return: “Sum of pending withdrawals, claimable withdrawals, and claimable amount” However in reality it returns: *return pendingWithdrawalAmount() + claimableWithdrawalAmount();*
Recommendations	Remove claimable amount from natspec.
Comments / Resolution	Fixed - Added a call to claimableAmount()

Issue_39	pendingWithdrawalAmount() for a single MellowVault doesn’t include the emergencyWithdrawals
Severity	Informational

Description	\`pendingWithdrawalAmount()\` only gets the pending assets of the pendingClaimers, but the MellowAdapter can have emergencyWithdrawals on that Vault, and those are not included:
	`function pendingWithdrawalAmount(`
	`        address _mellowVault`
	`    ) external view returns (uint256 total) {`
	`        for (uint256 i = 0; i < pendingClaimers.length(); i++) {`
	`            total +=`
	`IMellowSymbioticVault(_mellowVault).pendingAssetsOf(pendingClaimers.at(i));`
	`        }`
	Here, the pending assets of the emergency withdrawals will not be included, because emergency claimers are not stored in pendingClaimers.

Recommendations	Add the pending assets of the emergency claimer too.
Comments / Resolution	Fixed - Added a flag to get pendingWithdrawals or emergencyPendingWithdrawals.

MellowAdapterClaimer

The MellowAdapterClaimer is a helper contract designed to facilitate asset claiming operations through the IMellowAdapter and the MellowSymbioticVault. It works as a controlled intermediary to safely claim tokens on behalf of the adapter.

Appendix: Integration with IMellowAdapter

The IMellowAdapter deploys instances of MellowAdapterClaimer in a one-to-many relationship, where each withdrawal operation is assigned a dedicated claimer contract.

Core Invariants:

INV 1: Only the IMellowAdapter can call claim().

Privileged Functions

- None.

No issues found.

ISymbioticAdapter

The ISymbioticAdapter is a contract that enables delegation and withdrawal of assets into supported Symbiotic vaults. It can only be interacted with by the InceptionVault or authorized entities. Upon delegation, it transfers the specified asset from the caller and deposits it into a selected vault, provided the vault has been registered and uses the correct collateral.
Withdrawals are handled in two steps: first, the adapter calls the withdraw() function on the vault, which burns active shares and mints withdrawal shares for a future epoch. It tracks this withdrawal epoch per vault. Later, the claim() function can be called to finalize the withdrawal and transfer the funds to the InceptionVault. Claims are handled via dynamically deployed or reused claimer contracts, which act as temporary receivers of funds.
The adapter also supports emergency withdrawal and claim paths, using a dedicated emergency claimer. It maintains a list of all supported vaults and exposes functions to get the total deposited, pending, and inactive balances across all vaults. Only the owner can add or remove vaults from the adapter.

Appendix: Symbiotic Vault

A Symbiotic Vault is the foundational component of the Symbiotic protocol that manages staked assets, coordinates delegation to operators, and facilitates secure participation in network security. It acts as an accounting system where stakers deposit ERC-20 tokens, which are internally tracked and represented through shares. These deposits remain within the vault at all times, ensuring that no tokens are transferred out during delegation. Instead, delegation operates on a purely accounting basis.
Vaults serve as the bridge between stakers, networks, and operators. Once assets are deposited, vault curators or delegation logic determine how the stake is allocated to specific operators. These operators then use the delegated stake to perform duties for various networks—such as validating or securing infrastructure—without ever taking custody of the underlying capital.
Withdrawals from the vault are not instant. Instead, they are processed through an epoch-based queue system, where requests are tracked and become claimable after the designated waiting period. This ensures orderly exit mechanics and allows the vault to manage liquidity and delegation stability. In parallel, vaults are also equipped to handle slashing events, where networks can penalize misbehaving operators.

Core Invariants:

INV 1: Only vaults with matching collateral can be added.
INV 2: Only vaults present in _symbioticVaults are allowed for deposits, withdrawals, and claims.
INV 3: Cannot make a withdrawal from a vault until the previous one was claimed.

Privileged Functions

- addVault
- removeVault

Issue_40	Requesting more than one withdrawals in the same epoch will cause loss of funds
Severity	**High**

Description	In the withdraw function, requesting more than one withdraw for the same epoch is allowed:
	
	if (withdrawals[vaultAddress] != vault.currentEpoch() + 1 && withdrawals[vaultAddress] > 0) {
	revert WithdrawalInProgress();
	}
	
	However, in case of multiple withdrawal, after the first claim is completed, the withdrawals mapping of the corresponding vaultAddress will be deleted:
	
	delete withdrawals[vaultAddress];
	
	This will cause all other claims from the same epoch to revert here:
	
	if (withdrawals[vaultAddress] == 0) revert NothingToClaim();
	
	As a result, the operator will be unable to claim any additional withdrawals from the same epoch.

	Consider the following scenario: 1)In withdrawalQueueEpoch = 1 and SymbioticEpoch = 10, the operator calls undelegate, which will invoke the ISymbioticAdapter::withdraw function 2) In withdrawalQueueEpoch = 2 and SymbioticEpoch = 10 (the same epoch), the operator initiates undelegate again, increasing the withdrawal amount for this SymbioticEpoch. 3) After the epoch becomes claimable, the operator calls claim for withdrawalQueueEpoch = 1. The claiming succeeds, however the withdrawals mapping is deleted. 4) Now the operator tries to call claim for withdrawalQueueEpoch = 2, but the claiming revert with NothingToClaim(), due to the deleted mapping in withdrawalQueueEpoch = 1.
Recommendations	Enforce a restriction to prevent withdrawing, when the *withdrawals[vaultAddress] != 0.* Alternatively, refactor the withdraw and claim functions to support multiple withdrawal requests during the same SymbioticEpoch.
Comments / Resolution	Fixed - Refactored withdrawal and claiming functionalities, now the epoch is associated to the vault && assigned claimer.

Issue_41	Farm rewards cannot be claimed
Severity	**Medium**
Description	In the Symbiotic documentation,it is stated that some rewards will be transferred to farms, which could implement the Default Staker Rewards interface. However the ISymbioticAdapter does not have any way to call the claimRewards function. As a result these rewards cannot be claimed.
Recommendations	Consider implementing a way to claim these rewards.
Comments / Resolution	Fixed - InceptionVault can claim rewards from Inception via the SymbioticAdapter.

Issue_42	inactiveBalance does not match the natspec comments
Severity	**Informational**
Description	In the natspec it is stated that the inactiveBalance() will return: "Sum of pending withdrawals and claimable amount" However in reality it returns: *return pendingWithdrawalAmount();*
Recommendations	Remove claimable amount from natspec.
Comments / Resolution	Fixed - Added a call to claimableAmount()

SymbioticAdapterClaimer

The SymbioticAdapterClaimer is a lightweight helper contract used by the SymbioticAdapter to complete withdrawals from Symbiotic vaults. Upon deployment, it approves the adapter to spend the specified asset. It exposes a single claim() function, which can only be called by the adapter and forwards the withdrawal claim to the target vault on behalf of the specified recipient for a given epoch.

Appendix: Integration with IsymbioticAdapter

The ISymbioticAdapter deploys and manages reusable SymbioticAdapterClaimer contracts to facilitate withdrawals from the Symbiotic vault, restricted to approved vault addresses. Upon finalizing a withdrawal, the claim function in the SymbioticAdapterClaimer transfers the withdrawn funds directly to the Inception vault.

Core Invariants:

INV 1: Only the ISymbioticAdapter can call claim().

Privileged Functions

- None.

No issues found.

2nd Audit

1. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	4	3		1	
Medium	11	5	1	3	
Low	15	6		7	
Informational	24	10		14	1
Governance	1			1	
Total	55	24	1	26	1

After the initial audit, a second round was conducted which directly extends on the fixes of the first audit round and introduces a second team which fully audits the resolution commit. The following commit was audited:

Initial Commit:
https://github.com/inceptionlrt/smart-contracts/tree/48f7375f8defe39b82f6e45f82afc9d9045b1631/projects/vaults/contracts

Resolution Commit:
https://github.com/inceptionlrt/smart-contracts/tree/bbec68845ca632366fe0da2d76ed81fba10a1e69/projects/vaults/contracts

All previously acknowledged issues will be listed here for consistency reasons. All descriptions and invariants can be found in the first report above.

Please be aware that this audit is conducted on the as-is codebase.

While some issues may be corresponding to upgradeability concerns, any potential issues which may arise from upgrading the codebase on-chain are not covered by this audit.

We recommend proper testing before upgrading as well as a migration consultation.

This audit does not include any migration process.

2. Detection

InceptionAssetsHandler

Issue_01	setRewardsTimeline must not be called during an active distribution
Severity	**Governance**

Description	The setRewardsTimeline() function can be called at any time by the owner to modify the rewardsTimeline parameter. However, this variable directly impacts the totalAssets() calculation:
	function totalAssets() public view returns (uint256) {
	uint256 elapsedDays = (block.timestamp - startTimeline) / 1 days;
	uint256 totalDays = rewardsTimeline / 1 days;
	if (elapsedDays > totalDays) return _asset.balanceOf(address(this));
	*uint256 reservedRewards = (currentRewards / totalDays) * (totalDays - elapsedDays);*
	return (_asset.balanceOf(address(this)) - reservedRewards);
	}
	Modifying rewardsTimeline mid-distribution causes an immediate shift in totalAssets():
	● **Increasing** rewardsTimeline reduces totalAssets() by increasing reservedRewards
	● **Decreasing** it has the opposite effect, increasing totalAssets().
	This allows an attacker to sandwich the setRewardsTimeline() call—interacting before and after the change to exploit the instant impact on totalAssets() for profit.

Recommendations	Consider restricting the setRewardsTimeline() to revert during an active distribution.
Comments / Resolution	Acknowledged.

Issue_02	Possible step-wise jump reward farming
Severity	**Low**

Description	The current implementation of totalAssets() results in daily reward emissions:
	function totalAssets() public view returns (uint256) {
	uint256 elapsedDays = (block.timestamp - startTimeline) / 1 days;
	uint256 totalDays = rewardsTimeline / 1 days;
	if (elapsedDays > totalDays) return _asset.balanceOf(address(this));
	*uint256 reservedRewards = (currentRewards / totalDays) * (totalDays - elapsedDays);*
	return (_asset.balanceOf(address(this)) - reservedRewards);
	}
	This means that rewards accumulated during the day are **released all at once** when a full day has elapsed. Attackers may leverage this and farm rewards with a short-term deposit. While instant withdrawal fees exist, this type of timing-based attack may still be profitable in some scenarios.

Recommendations	Consider implementing a second based emission instead.
Comments / Resolution	Acknowledged.

Issue_03	Precision loss due to division before multiplication.
Severity	**Informational**
Description	The following line in totalAssets() contains a division before multiplication, which creates unnecessary precision loss: *uint256 reservedRewards = (currentRewards / totalDays) * (totalDays - elapsedDays);*
Recommendations	Consider changing the order of operations to increase precision.
Comments / Resolution	Resolved by following recommended mitigation.

Issue_04	Wrong event emission, the old reward treasury is emitted instead of the new one.
Severity	**Informational**

Description	In setRewardsTreasury(), the following event is emitted:
	function setRewardsTreasury(address treasury) external onlyOwner {
	require(treasury != address(0), NullParams());
	emit SetRewardsTreasury(rewardsTreasury);
	rewardsTreasury = treasury;
	}
	However, the event logs the old rewardsTreasury instead of the new one - treasury.

Recommendations	Consider updating the event if this is not intended.
Comments / Resolution	Resolved by following recommended mitigation.

AdapterHandler

Issue_05	Storage collision due to increasing the storage gap
Severity	**High**
Description	The storage gap is assigned 39 slots, while in fact, 12 slots have been used: */*** ** @dev Reserved storage gap to allow for future upgrades without shifting storage layout.* ** @notice Occupies 38 slots (50 total slots minus 12 used).* **/* *uint256[50 - 11] private __gap;* The root cause is that there is a new variable "_adapters" which uses the type EnumerableSet.AddressSet. This type is a struct of type Set (link) that contains an array and a mapping, which occupies two storage slots instead of one. Given that in total three extra storage slots are being used on this update, but the array gap is only decremented by two slots, it will cause the contract to use one extra slot than before, corrupting all the state on InceptionVault_S. Due to using more storage in this implementation—**51 slots instead of 50**—a storage collision occurs: While in the currently deployed version the InceptionToken is at slot 351, it has been moved to slot 352 in the new version, which was previously used for withdrawMinAmount. As a result, a storage collision will occur when the contracts are upgraded.
Recommendations	Consider using a smaller storage gap (1 slot less) to avoid corrupting the storage. Furthermore, we highly recommend testing the storage layout before upgrading and consider a migration consultation.
Comments / Resolution	Resolved by following recommended mitigation.

Issue_06	Funds from Mellow Withdrawals Can Be Taken by Other Users
Severity	**High**
Description	Mellow withdrawals can be either instant, queued, or a combination of both, depending on the availability of assets at the time of the request. In cases of partial instant withdrawals, the immediately available portion is sent directly to the Vault, while the remainder is queued. Once the queue delay ends, the remaining funds are delivered to the Vault, and WithdrawalQueue.claim() is called. This updates totalAmountRedeem, which adjusts the vault's share price to ensure only the original withdrawing users can claim those funds. However, there is an issue during this flow: the instant portion is not reserved for the withdrawing user during the queuing period. This leaves it exposed to other users, who can withdraw these assets via flashWithdraw() before the queued portion arrives. Exploit Scenario: 1. The Vault has flashCapacity = 0. 2. There are 25 reward assets pending distribution. 3. Bob requests to withdraw 50 assets. 4. The Vault withdraws 50 assets from Mellow: a. 25 assets are received instantly. b. 25 assets are queued. 5. The 25 instant assets are deposited into the Vault, increasing flashCapacity to 25. 6. Alice observes this and calls flashWithdraw(), burning 25 shares and withdrawing the 25 instant assets. 7. flashCapacity returns to 0. 8. Later, when the queued 25 assets are received from Mellow: a. They are deposited into the Vault. b. WithdrawalQueue.claim() is called, increasing totalAmountRedeem by 50. 9. Bob calls redeem() expecting 50 assets. The Vault only has 25 (from the queued withdrawal), so it takes the remaining 25 from the rewards, which was meant for distribution. 10. As a result, the Vault's real balance becomes less than the pending rewards, causing totalAssets() to underflow and revert.

	This leads to a complete DoS on both deposits and withdrawals until enough new funds are added to the Vault—something that may take weeks, depending on the withdrawal epoch duration on the active adapters. As a final note, the admins could solve this DoS by directly sending tokens to the contract, but that would result in a direct loss of funds for the protocol.
Recommendations	1. Introduce a variable in WithdrawalQueue to track partially fulfilled Mellow withdrawals (i.e., the instant portion of a pending withdrawal). 2. Subtract this tracked value from the flashCapacity, so these funds are reserved and cannot be used by others. 3. Update the totalAssets() logic to safely handle underflows, ensuring it does not revert when the Vault's actual balance is lower than the pending rewards. Please be aware that intrusive logic or refactoring may fall beyond the standard resolution round.
Comments / Resolution	Fixed by implementing the following mitigations: 1. Created totalPendingRedeemAmount variable to track the funds from partially fulfilled Mellow withdrawals. 2. Subtracted totalPendingRedeemAmount from the flash capacity. 3. Updated totalAssets to handle underflows.

Issue_07	Users can grief undelegations with a direct transfer to the Inception vault
Severity	**Medium**
Description	Consider the following scenario → 1.) Operator calls undelegate() (AdapterHandler.sol) , this call would firstly call withdraw() on the adapter to initiate a withdrawal and then calls undelegate on the WithdrawalQueue.sol 2.) In the WithdrawalQueue's undelegate call it invokes the _afterUndelegate hook to check if undelegations were done correctly → *function _afterUndelegate(uint256 epoch, WithdrawalEpoch storage withdrawal) internal {* *uint256 requested = IERC4626(inceptionVault).convertToAssets(withdrawal.totalRequestedShares);* *uint256 totalUndelegated = withdrawal.totalUndelegatedAmount + withdrawal.totalClaimedAmount;* *// ensure that the undelegated assets are relevant to the ratio* *require(* *requested >= totalUndelegated ?* *requested - totalUndelegated <= MAX_CONVERT_THRESHOLD* *: totalUndelegated - requested <= MAX_CONVERT_THRESHOLD,* *UndelegateNotCompleted()* *);* 3.) It firstly calculates requested assets based on the total requested shares in the epoch and compares that with the undelegated amount , the total deviation should be within MAX_CONVERT_THRESHOLD.

	4.) An attacker can frontrun an operator’s undelegate call and directly transfer minimal assets to the inception vault , this direct transfer would increase the value returned by the convertToAssets() (because ratio goes down) and the check , require(*requested >= totalUndelegated ?* *requested - totalUndelegated <= MAX_CONVERT_THRESHOLD* would revert. The impact would be high in cases where the attacker deliberately griefs the undelegations and during this time the staker (i.e. adapters) gets slashed , which would reset the epoch and could result in losses which the users would bear.
Recommendations	Consider increasing the value of *MAX_CONVERT_THRESHOLD.*
Comments / Resolution	Acknowledged. Inception Team can use flash-bots to submit undelegations to prevent attackers from deliberately frontrunning the undelegations and causing DoS.

Issue_08	rewardsTimeline is not initialized
Severity	**Informational**
Description	The rewardsTimeline is set using the setRewardsTimeline function. However it is not initialized. As a result, if addRewards() is called, there will be DOS due to division by 0 in totalAssets(): *uint256 reservedRewards = (currentRewards / totalDays) * (totalDays - elapsedDays);*
Recommendations	Consider initializing the rewardsTimeline.
Comments / Resolution	Fixed by adding a check that ensures the timeline to be set before adding rewards.

Issue_09	Emergency functions cannot be used during paused state
Severity	**Informational**
Description	The AdapterHandler implements emergencyUndelegate and emergencyClaim, however these functions are also restricted with a whenNotPaused modifier, so they cannot be accessed in emergency situations.
Recommendations	Determine if this is the intended behavior and remove the whenNotPaused modifiers from the emergency functions if necessary.
Comments / Resolution	Acknowledged.

Issue_10	claimAdapterFreeBalance and claimAdapterRewards are missing the whenNotPaused modifier.
Severity	**Informational**
Description	According to the Natspec, both functions claimAdapterFreeBalance and claimAdapterRewards should only be callable when the contract is not paused. However, this limitation is not enforced in the code.
Recommendations	Consider adjusting the comments or adding a whenNotPaused modifier to these functions.
Comments / Resolution	Resolved by fixing the Natspec.

InceptionVault_S

Issue_11	An attacker can front-run setWithdrawalQueue and steal funds from the vault
Severity	**High**
Description	When the withdrawalQueue is changed, setWithdrawalQueue can be called, to update it with its new address. However, this is not safe, as an attacker could take advantage of the legacy settings in the withdrawalQueue and front-run the setWithdrawalQueue with a call to the redeem function. Consider the following scenario: 1) In T1, owner initializes the withdrawalQueue with legacy withdrawals including the attacker. In the same transaction the owner calls setWithdrawalQueue. 2) The attacker sees the owner's T1 in the mempool, so he front-runs it with T0, which calls the redeem function. 3) Now, since the new withdrawalQueue also includes the attackers legacy amount, the attacker will be able to redeem again.
Recommendations	We recommend pausing the protocol before initializing the withdrawalQueue contract. To ensure a safe update, it is also advisable to include a check in the setWithdrawalQueue function that verifies the protocol is in a paused state.
Comments / Resolution	Acknowledged. The Inception team will take the following steps to ensure secure withdrawalQueue migration: 1) The operator will call SymbioticHandler::updateEpoch and complete all currently pending withdrawals. 2)The owner will pause the vault and record all unclaimed withdrawals, if there is any unprocessed withdrawal, it will be caught during the recording phase and processed again, by unpausing and invoking the updateEpoch() function.

Issue_12	Some functions can be griefed via a tiny donation
Severity	**Medium**
Description	There are some functions within the contract in scope that have strict requirements of having 0 balance before doing some actions. These functions are the following: - InceptionVault_S.migrateDepositBonus - AdapterHandler.removeAdapter - InceptionSymbioticAdapter.removeVault - InceptionWstETHMellowAdapter.removeVault These requirements allow attackers to execute a griefing attack by performing a tiny donation/deposit that prevents from executing these actions. While the attacker does not profit from this attack, migrating the depositBonus may be prevented, causing stuck funds.
Recommendations	It's recommended to consider if it's necessary to create a bool argument on these functions that allow admins to indicate if the checks for 0 balance should be ignored or not.
Comments / Resolution	Resolved by introducing skipEmptyCheck variable.

Issue_13	previewMint does not account for the depositBonus
Severity	**Low**
Description	The previewMint will return how many assets are required to mint the corresponding number of shares. However, it does not take the depositBonus into consideration, which will cause it to return more assets. Per the EIP-4626, previewMint "MUST return as close to and no fewer than the exact amount of assets that would be deposited in a mint"
Recommendations	The previewMint function should take the deposit bonus into consideration when calculating the required assets. Alternatively, we recommend acknowledging this issue, because fixing it would require introduction of unnecessary complexity.
Comments / Resolution	Acknowledged.

Issue_14	claim() will shift the ratio
Severity	**Low**

Description	The claim() function will perform the following changes to the ratio:
	totalAmountRedeem += withdrawal.totalClaimedAmount; //decreases totalAssets
	totalSharesToWithdraw -= withdrawal.totalRequestedShares; //decreases totalSupply
	However, when there is an ongoing reward accumulation, the totalAssets will not be derived using the current ratio, because users who are withdrawing do not receive rewards. As a result, after claim() is called, there will be an instant shift in the ratio, potentially allowing attackers to sandwich the claim call and extract more assets. While there are protocol fees, this attack may still be profitable in some cases.

Recommendations	Fixing this issue will not be trivial so consider acknowledging it.
Comments / Resolution	Acknowledged.

Issue_15	migrateDepositBonus Fails to Update Bonus Tracking on Target Vault
Severity	**Low**

Description	The migrateDepositBonus function is intended to transfer the remaining deposit bonus from a sunset vault to a new one:
	`function migrateDepositBonus(address newVault) external onlyOwner {`
	`require(getTotalDelegated() == 0, ValueZero());`
	`require(newVault != address(0), InvalidAddress());`
	`require(depositBonusAmount > 0, NullParams());`
	`uint256 amount = depositBonusAmount;`
	`depositBonusAmount = 0;`
	`_asset.safeTransfer(newVault, amount);`
	`emit DepositBonusTransferred(newVault, amount);`
	`}`
	However, while the funds are transferred, the depositBonusAmount is not updated on the receiving vault. This causes the new vault's share price to jump unexpectedly, since the additional assets are not backed by any increase in depositBonusAmount.
	Given that sunset vaults often have large unclaimed bonuses due to mass withdrawals, this behavior creates an opportunity for MEV bots to exploit the discrepancy.
	Exploit Scenario:
	1. An attacker takes a flash loan.
	2. The attacker deposits into the new vault.
	3. The migrateDepositBonus function is called on the old vault, transferring bonus funds to the new vault and increasing its share price.
	4. The attacker immediately withdraws, taking profit from the artificially boosted share price.
	This sandwich strategy is profitable as long as the share price increase outweighs any withdrawal fees.

Recommendations	Modify migrateDepositBonus to accept an optional bool flag. If the target address is another Inception Vault, this flag should trigger an update to the recipient's depositBonusAmount, ensuring the

	accounting remains correct and the share price stays consistent with the vault's bonus tracking.
Comments / Resolution	Fixed by sending to deposit bonus to the caller and not another vault.

Issue_16	The Vault doesn't comply with ERC-4626
Severity	**Low**
Description	The Vault has a lot of comments referencing the ERC-4626 standard as a reference, but its behavior wildly differs from it. Here is a summary of the differences between ERC-4626 and the vault: - totalAssets should return the total amount of underlying assets managed by the vault, but the vault uses getTotalDeposited for this purpose. - maxDeposit and maxMint are missing checks for targetCapacity. - previewDeposit should check that targetCapacity > 0, the vault is not paused, and that the resulting shares are not zero - previewMint should take into account the deposit bonus, and not be called from mint. - maxWithdraw and previewWithdraw are non-existent. - withdraw function lacks the owner parameter - The withdraw and redeem functions are confusing as they do different types of withdrawals depending on their arguments. The above list is not extensive and just points out the obvious discrepancies.
Recommendations	Consider acknowledging this issue and documenting the discrepancies between the Inception vault and the ERC4626 standard.
Comments / Resolution	Acknowledged.

Issue_17	InceptionVault_S::mint doesn't provide a maxAmountIn parameter, leading to front-running risk
Severity	**Informational**
Description	Users can deposit funds into the vault by calling deposit function, which mints iShares for the user. The deposit function does not provide a minOut parameter, which means that the user limits the amount of assets paid to the vault, and ensures that he receives shares >= minOut amount. The mint function in the InceptionVault_S contract is the other way around, where users provide the amount of iShares they want to receive, and the function calculates the amount of assets that should be paid to the vault, and mints the iShares for the user. However, the mint function does not provide a maxAmountIn parameter, which means that the user cannot limit the amount of assets paid to the vault, and it could be front-run by other users who could donate assets to the vault before the user mints iShares. This could lead to the user paying more than expected for the iShares, as the amount of assets paid to the vault could be greater than the amount calculated by the previewMint function.
Recommendations	Consider adding slippage protection on mint().
Comments / Resolution	Acknowledged.

Issue_18	The ratioFeed shall be legacy
Severity	**Informational**

Description	The ratioFeed is still named the same, even though this is no longer used. There is also the following function which currently has no purpose and will never be used:
	function setRatioFeed(IinceptionRatioFeed newRatioFeed) external onlyOwner {
	if (address(newRatioFeed) == address(0)) revert NullParams();
	emit RatioFeedChanged(address(ratioFeed), address(newRatioFeed));
	ratioFeed = newRatioFeed;
	}

Recommendations	Consider renaming the ratioFeed and removing the unnecessary setRatioFeed function.
Comments / Resolution	Resolved by following recommended mitigation.

Issue_19	mint() may return more shares than requested
Severity	**Informational**
Description	Due to the deposit bonus, the mint function may return more shares than requested. This shall be clearly noted, so that integrating protocols are aware of this behavior.
Recommendations	Consider stating that the mint() can return more shares than requested in the natspec and documentation.
Comments / Resolution	Resolved by following recommended mitigation.

Issue_20	Flash withdraws will revert even if there is enough flash capacity
Severity	**Informational**
Description	flashWithdraw() will check if flashCapacity is enough before subtracting the fees: *function calculateFlashWithdrawFee(uint256 amount) public view returns (uint256) {* *uint256 capacity = getFlashCapacity();* *if (amount > capacity) revert InsufficientCapacity(capacity);* If the flashCapacity is enough after the fees have been subtracted, the transaction will still revert.
Recommendations	Consider checking the capacity after subtracting the fees.
Comments / Resolution	Acknowledged.

Issue_21	previewRedeem does not comply with EIP-4626
Severity	**Informational**
Description	EIP states that previewRedeem may revert by the same reason that would cause `redeem()` to revert. However, in order to prevent a revert when *amount>capacity*, it caps the capacity to the amont: *uint256 flash = amount <= capacity ? capacity : amount;*
Recommendations	Revert when amount > capacity
Comments / Resolution	Acknowledged - Known issue if reverting for external integrators

Issue_22	setDepositBonusParams and setFlashWithdrawFeeParams do not validate the curve params
Severity	**Informational**

Description	The setDepositBonusParams and setFlashWithdrawFeeParams will change key variables for the calculation of the deposit bonus and the flash withdrawal fee. However, these params are not sufficiently validated:
	`function setDepositBonusParams(uint64 newMaxBonusRate, uint64 newOptimalBonusRate, uint64 newDepositUtilizationKink)`
	`    external`
	`onlyOwner`
	`  {`
	`    if (newMaxBonusRate > MAX_PERCENT) {`
	`      revert ParameterExceedsLimits(newMaxBonusRate);`
	`    }`
	`    if (newOptimalBonusRate > MAX_PERCENT) {`
	`      revert ParameterExceedsLimits(newOptimalBonusRate);`
	`    }`
	`    if (newDepositUtilizationKink > MAX_PERCENT) {`
	`      revert ParameterExceedsLimits(newDepositUtilizationKink);`
	`    }`
	`    if (newOptimalBonusRate > newMaxBonusRate) revert InconsistentData();`
	`maxBonusRate = newMaxBonusRate;`
	`optimalBonusRate = newOptimalBonusRate;`
	`depositUtilizationKink = newDepositUtilizationKink;`
	Passing a wrong rate or a kink could result in a choppy or discontinuous curve, allowing for fee and deposit bonus manipulations.

Recommendations	We recommend enforcing some validations on the inputs of the setDepositBonusParams and setFlashWithdrawFeeParams , to ensure that the curve is continuous at capacity == optimalCapacity.

	Alternatively if these params are not going to be changed we recommend removing the setter functions or acknowledging the issue.
Comments / Resolution	Acknowledged.

Issue_23	redeem() should be renamed to flashRedeem()
Severity	**Informational**

Description	When a user calls redeem() which takes shares as input , it performs a flash withdraw under the hood plus passes 0 as the slippage control parameter (minOut) →
	function redeem(
	uint256 shares,
	address receiver,
	address owner
	) external nonReentrant whenNotPaused returns (uint256) {
	if (owner != msg.sender) revert MsgSenderIsNotOwner();
	__beforeWithdraw(receiver, shares);
	(uint256 assets, uint256 fee) = _flashWithdraw(shares, receiver, owner, 0);
	emit Withdraw(owner, receiver, owner, assets, shares);
	emit WithdrawalFee(fee);
	return assets;
	}
	This can be confusing for the user since he would expect a regular redeem according to the totalClaimable in the epoch but would in turn be doing a flashWithdraw which also incurs a flash withdrawal fee.

Recommendations	Consider renaming the function to flashRedeem().

Comments / Resolution	Acknowledged.

InceptionToken

Issue_24	Pause On Inception Token Transfer Also Pauses Mints And Burns
Severity	Low
Description	The share token in the system is supposed to be the inception token which can be paused to pause token transfers → *function _beforeTokenTransfer(* *address from,* *address to,* *uint256 amount* *) internal override {* *super._beforeTokenTransfer(from, to, amount);* *require(!paused(), "InceptionToken: token transfer while paused");* *}* But this pause will also pause mints and burns which will in turn pause all the deposits and withdraws from the vault , this would result in an unintended halt of the critical functionalities of the system.
Recommendations	Consider updating the _beforeTokenTransfer hook and only revert if “to” and “from” are not 0 addresses. Alternatively, this issue can be acknowledged.
Comments / Resolution	Acknowledged.

WithdrawalQueue

Disclaimer: This audit treated the setWithdrawalQueue function only in the context of migrating from the currently deployed version to the new version of the vault. setWithdrawalQueue must not be used outside of this migration scenario, as doing so could lead to potential accounting discrepancies

Issue_25	Slashing may prevent withdrawals
Severity	**Medium**

Description	When there are withdrawals, the operator may choose from which adapters to withdraw.
	However, in case of a full slashing on the SymbioticVault, the claim for that adapter will be reverted:
	
	`function _claim(`
	`    uint256 epoch`
	`) internal returns (uint256 amount) {`
	`    // ...`
	
	`    amount = withdrawalsOf(epoch, msg.sender);`
	
	`    if (amount == 0) {`
	`        revert InsufficientClaim();`
	`    }`
	
	`    // ...`
	`}`
	
	This revert can be avoided by ignoring the fully slashed adapter, when calling AdapterHandler::claim(), passing all the other adapters instead.
	However, there can be cases when the operator decides to only withdraw from the SymbioticAdapter. When that happens, if there is a full shashing, ignoring the slashed adapter is not possible due to the following validation in the withdrawalQueue:
	
	`require(`

	adapters.length > 0 && adapters.length == vaults.length && adapters.length == claimedAmounts.length,
	ValueZero()
	);
	This will not allow the operator to pass 0 adapters, fully preventing the withdrawal from completion.

Recommendations	Consider removing the adapter.length validation.
Comments / Resolution	

Issue_26	Slashing will be missed in an edge case
Severity	**Medium**

Description	The _isSlashed() function will perform the following check to recognize slashing when the whole undelegated amount has been claimed:
	if (withdrawal.totalClaimedAmount >= withdrawal.totalUndelegatedAmount) {
	if (
	currentAmount < withdrawal.totalClaimedAmount
	&& withdrawal.totalClaimedAmount - currentAmount > MAX_CONVERT_THRESHOLD
	) {
	return true;
	}
	}
	However, when one of the adapters being undelegated is the InceptionWstETHMellowAdapter, a portion of the assets will be instantly claimed without being undelegated, therefore, they will not be recorded in the totalUndelegatedAmount:
	function withdraw(
	...

```
    ) external override onlyTrustee whenNotPaused returns (uint256, uint256) {
        address claimer = _getOrCreateClaimer(emergency);
        uint256 balanceState = _asset.balanceOf(claimer);
        // claim from mellow
        IERC4626(_mellowVault).withdraw(amount, claimer, address(this));
        _claimerVaults[claimer] = _mellowVault;
        uint256 claimedAmount = (_asset.balanceOf(claimer) - balanceState);
        uint256 undelegatedAmount = amount - claimedAmount;
```

As a result, when the InceptionWstETHMellowAdapter is used, slashing may remain undetected, due to totalClaimedAmount exceeding the totalUndelegatedAmount even if one of the withdrawn adapters has been slashed during the queue.

Consider the following scenario:

0) The vault has 200eth, 100eth staked in Mellow and 100 eth staked in eigenlayer.(400/400 ratio = 1:1)

1) A user withdraws 150e18.

2) The operator splits the undelegation between Mellow and EigenLayer

3) The Mellow adapter will claim 100 tokens and undelegate 0 (due to using free balance in Mellow)

totalClaimedAmount = 100

totalUndelegatedAmount = 0

The EigenLayer adapter will undelegate 50 tokens:
totalClaimedAmount = 100

totalUndelegatedAmount = 50

4) During claiming slashing occurs, so now the 50 tokens are slashed and only 25 are claimed:

totalClaimedAmount = 125

	totalUndelegatedAmount = 50 Now _isSlashed is executed entering here: *if (withdrawal.totalClaimedAmount >= withdrawal.totalUndelegatedAmount) {* However, the user will receive **125 assets(totalClaimedAmount)** for 150 shares. The vault has 325eth, 0 staked in Mellow and 25 eth staked in eigenlayer. This makes the **currentAmount = 131e18** As a result, the following check will return false, not recognizing the slashing: *if (* *currentAmount < withdrawal.totalClaimedAmount* *&& withdrawal.totalClaimedAmount - currentAmount > MAX_CONVERT_THRESHOLD* *) {*
Recommendations	Consider implementing the following changes: 1. Add all the assets that were claimed on the execution of the claim(). - These assets are part of the withdrawal.totalUndelegatedAmount - *the amount of assets that were queued for withdrawal on Mellow.* 2. Discount from withdrawal.totalUndelegatedAmount all the claimed assets (not withdrawal.totalClaimedAmount, but, a new variable that adds all the assets claimed from the adapter associated to a withdrawal request from the claim() call). 3. Check if there are any remaining undelegated assets, which were not claimed.If so, that flags a slashing. *Please be aware that intrusive logic or refactoring may fall beyond the standard resolution round.* Comment from supervisor: Such a change seems to be very intrusive and given that this is the second round we target to have as small changes as possible. If this can be acknowledged under

	any scenario I highly recommend acknowledging it. The risk of introducing more severe issues due to code changes is existing.
Comments / Resolution	Fixed. Refactoring to update withdrawal.totalUndelegatedAmount by the instant claimed amount from Mellow when delegating prevents the edge case from occurring.

Issue_27	All Epochs Incorrectly Marked as "Slashed" Due to Yield and Rounding
Severity	**Medium**

Description	The protocol attempts to detect whether a withdrawal has been slashed while it was queued by comparing the expected amount (based on vault share price) with the actual amount claimed:
	function _isSlashed(WithdrawalEpoch storage withdrawal) internal view returns (bool) {
	uint256 currentAmount = IERC4626(inceptionVault).convertToAssets(withdrawal.totalRequestedShares);
	if (withdrawal.totalClaimedAmount >= withdrawal.totalUndelegatedAmount) {
	if (currentAmount < withdrawal.totalClaimedAmount && withdrawal.totalClaimedAmount - currentAmount > MAX_CONVERT_THRESHOLD) {
	return true;
	}
	} else if (currentAmount > withdrawal.totalClaimedAmount && currentAmount - withdrawal.totalClaimedAmount > MAX_CONVERT_THRESHOLD) {
	return true;
	} else if (currentAmount < withdrawal.totalClaimedAmount && withdrawal.totalClaimedAmount - currentAmount > MAX_CONVERT_THRESHOLD) {
	return true;
	}

	return false; *}* However, this logic contains an issue: the threshold for discrepancy (MAX_CONVERT_THRESHOLD) is set to 50 wei, which is too small given that vault values typically use 18 decimal places. As a result, almost every epoch gets marked as "slashed", even when no actual slashing has occurred. **Root Causes:** 1. **Tiny Rounding Errors**: Even in normal non-slashed withdrawals, minor rounding differences between undelegation and claiming (e.g., due to Mellow or Symbiotic internal rounding) can cause totalClaimedAmount to be at least a wei lower than totalUndelegatedAmount, bypassing the first condition and entering the else-if blocks. 2. **Using the Current Share Price**: The convertToAssets function uses the current vault share price, which is often higher than it was when undelegation occurred (typically days earlier). This artificially inflates the "expected" claim amount, making it seem like less was claimed—even when no slashing occurred. 3. **Griefing via Donations**: Anyone can donate a small amount of assets to the vault to increase the share price slightly. Since the slashing logic is based on current share price, this donation can intentionally trigger false slashing on queued withdrawals. Because all epochs are marked will be marked as “slashed,” normal withdrawal finalization is blocked. This forces the operator to manually handle withdrawals via forceUndelegateAndClaim and delaying all withdrawals significantly.
Recommendations	Consider using the MAX_CONVERT_THRESHOLD in the following check: *if (withdrawal.totalClaimedAmount >= withdrawal.totalUndelegatedAmount) {*
Comments / Resolution	Partially resolved. Root Causes 2 && 3 are fixed. Root cause 1 can still occur. The tiny rounding errors may cause totalClaimedAmount to be less than the totalUndelegatedAmount, causing the first if block to be skipped. Due to using the currentAmount, any natural reward accrual will lead to wrongly marking the withdrawal as slashed.

	Consider adding the following to the first check: *if (withdrawal.totalClaimedAmount >= withdrawal.totalUndelegatedAmount - MAX_CONVERT_THRESHOLD)*

Issue_28	Withdrawal Epoch Can Get Permanently Stuck if a Symbiotic Vault Is 100% Slashed
Severity	**Medium**

Description	If a queued withdrawal on Symbiotic is slashed by 100%, any attempt to claim funds will revert due to the following check in the Symbiotic vault's _claim function:
	`function _claim(`
	`    uint256 epoch`
	`) internal returns (uint256 amount) {`
	`    // ...`
	`    amount = withdrawalsOf(epoch, msg.sender);`
	`    if (amount == 0) {`
	`        revert InsufficientClaim();`
	`    }`
	`    // ...`
	`}`
	In this scenario, since the slashing wipes out the full amount, withdrawalsOf() returns 0, causing the call to revert. This prevents the operator from successfully completing the claim on the Symbiotic adapter.
	This failure propagates to the main protocol, which depends on successful claim calls from all adapters before marking an epoch as redeemable. Specifically, the following check in _afterClaim() will prevent finalization:
	`function _afterClaim(`
	`    uint256 epoch,`
	`WithdrawalEpoch storage withdrawal,`
	`    address[] calldata adapters,`
	`    address[] calldata vaults`

	) internal { *require(withdrawal.adaptersClaimedCounter == withdrawal.adaptersUndelegatedCounter, ClaimNotCompleted());* *// ...* *}* Because the Symbiotic adapter can never be claimed, the adaptersClaimedCounter will never match adaptersUndelegatedCounter, and the withdrawal epoch will remain stuck indefinitely. Attempts to manually resolve the situation with forceUndelegateAndClaim() will also fail, as the totalUndelegatedAmount will remain greater than 0, making the epoch ineligible for forced redemption. As a result, users in that withdrawal epoch will be permanently unable to redeem their funds.
Recommendations	Remove the requirement that: *adaptersClaimedCounter == adaptersUndelegatedCounter* This gives the trusted operator flexibility to finalize epochs even when one of the adapters (e.g., Symbiotic) constantly reverts the txn when trying to claim any funds due to total slashing.
Comments / Resolution	Fixed. adapterCounters have been removed.

Issue_29	Stuck funds when forcing a withdrawal to an epoch with partially claimed tokens.
Severity	Low
Description	The following edge case has appeared from this last round of fixes: 1. Operator starts an undelegation from Mellow and some funds are received instantly a. The claimed amount is added to the new variable totalPendingRedeemAmount. 2. After some time, the operator decides to complete that withdrawal epoch by using forceUndelegateAndClaim. a. The previously claimed amount won't be subtracted from totalPendingRedeemAmount, causing those funds to be stuck on the vault because the won't be included on the flash capacity. This exact sequence will cause some stuck funds on the vault that will have to be recovered via an upgrade. Even though it's not common to use forceUndelegateAndClaim to finish an already-started withdrawal epoch, it's an edge case that could happen under extraordinary circumstances.
Recommendations	If the operator doesn't plan to use forceUndelegateAndClaim with withdrawal epochs that have been started but still not claimed, then this issue can be acknowledged safely. However, if this sequence of actions is expected from the operator, it's recommended to fix this issue by simply adding one line on forceUndelegateAndClaim before all state updates. *totalPendingRedeemAmount -= withdrawal.totalClaimedAmount*
Comments / Resolution	

Issue_30	Withdrawals could only be fulfilled by either undelegation or flash capacity, but not both, leading to blocked withdrawals
Severity	**Low**
Description	When users deposit into the vault, these funds get deposited into multiple adapters, Symbiotic and Mellow are examples. However, not all deposited funds could be delegated into these adapters, as the protocol introduces a target capacity concept, which is the amount of assets that could be delegated to the adapters. This is checked before any delegation: *uint256 freeBalance = getFreeBalance();* *require(amount <= freeBalance, InsufficientCapacity(freeBalance));* On the other hand, when users request a withdrawal, the vault operator can undelegate the staked assets from the adapters, or fund the withdrawal from the funds sitting in the vault, which is called flash capacity. Both could be done by calling the undelegate function in the AdapterHandler contract, which is inherited by all vaults. If no requests are passed, it will use the flash capacity to fulfill the withdrawal: *if (requests.length == 0) return _undelegateAndClaim(undelegatedEpoch);* However, this introduces a problem, as the undelegate could fail if the requested amount is greater than the undelegated amount, and less than the flash capacity, which would lead to a revert. This is because the undelegate function checks that the requested amount is less than or equal to the undelegated amount, and if it is not, it reverts with UndelegateNotCompleted error. On the other hand, if the requested amount is greater than the free balance, it will revert with InsufficientFreeBalance error. This keeps the withdrawals stuck in the withdrawal queue, as the undelegation is not completed, and the flash capacity is not used, leading to blocking withdrawals.

Recommendations	Fixing this issue would require refactoring. Given that the funds are not stuck and emergencyUndelegate() and emergencyClaim() can be used in this rare case, we recommend acknowledging the issue.
Comments / Resolution	Acknowledged.

Issue_31	adapterUndelegated is reset on _undelegate()
Severity	**Informational**
Description	In _undelegate(), the adapterUndelegated is set to the undelegatedAmount. This will not be accurate in the unlikely scenario when there are multiple undelegations for one epoch from the same adapter, due to setting instead of increasing the adapterUndelegated: *withdrawal.adapterUndelegated[adapter][vault] = undelegatedAmount;*
Recommendations	Considerusing \`+=\` insteadof \`=\`.
Comments / Resolution	Fixed by implementing recommendations.

Issue_32	Slashing may be missed due to ongoing reward distribution
Severity	**Informational**

Description	During claiming the following validation is performed in order to detect slashing in _isSlashed:
	
	`if (withdrawal.totalClaimedAmount >= withdrawal.totalUndelegatedAmount) {`
	`    if (`
	`        currentAmount < withdrawal.totalClaimedAmount`
	`            && withdrawal.totalClaimedAmount - currentAmount > MAX_CONVERT_THRESHOLD`
	`    ) {`
	`        return true;`
	`    }`
	`} else if (`
	`1)  currentAmount > withdrawal.totalClaimedAmount`
	`        && currentAmount - withdrawal.totalClaimedAmount > MAX_CONVERT_THRESHOLD`
	`) {`
	`    return true;`
	`} else if (`
	`    currentAmount < withdrawal.totalClaimedAmount`
	`        && withdrawal.totalClaimedAmount - currentAmount > MAX_CONVERT_THRESHOLD`
	`) {`
	`    return true;`
	`}`
	
	If the totalClaimedAmount is less than the totalUndelegatedAmount, the check in point 1 will be entered. The currentAmount is calculated using the following formula:
	
	`uint256 currentAmount = IERC4626(inceptionVault).convertToAssets(withdrawal.totalRequestedShares);`
	
	This approach uses the current ratio to determine the amount; however, the ratio includes accrued rewards. In edge cases, pending rewards may fully offset the impact of slashing, causing the conditional check in point 1 to return false. As a result, the slashing event may go unrecognized by the system, despite having occurred.

Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

InceptionLibrary

Issue_33	Users can limit their fee exposure by splitting their withdrawal into smaller portions, paying less fee to the protocol
Severity	**Medium**
Description	The subtractor calculation uses the midpoint between the starting and the finishing capacity for the transaction. This is used to increase the fee when the capacity goes below the optimal capacity proportionally: *uint256 subtractor = (feeSlope * (capacity - amount / 2)) / targetCapacity;* The subtractor can be exploited by any user by splitting their withdrawal into small portions to keep the subtractor as high as possible, which will substantially decrease their fee up to 10-15%, which is a direct loss of revenue for the protocol.
Recommendations	In order to prevent this attack, consider introducing a withdrawal delay.
Comments / Resolution	Acknowledged

Issue_34	Users can extract more depositBonus by splitting their deposit into smaller amounts
Severity	**Medium**
Description	The subtractor calculation uses the midpoint between the starting and the finishing capacity for the transaction. This is used to proportionally decrease user's deposit as the capacity gets closer to the optimalCapacity. *uint256 subtractor = (bonusSlope * (capacity + replenished / 2)) / targetCapacity;* This can be exploited by any user by splitting their deposit into small portions to keep the subtractor as low as possible, which will substantially increase their bonus.
Recommendations	In order to prevent this attack, consider introducing a deposit delay.
Comments / Resolution	Acknowledged.

Issue_35	Users can sweep a portion of the deposit bonus for free
Severity	**Informational**
Description	The deposit bonus is granted to users upon depositing. This can be gamed by users to extract a big portion of the deposit bonus for free. Consider the following scenario: 1) The capacity is below the optimal capacity 2) A user takes a flash loan and makes a deposit that will bring the capacity up to the targetCapacity, in order to extract the maximum deposit bonus possible. 3) After that the user slaps the received Inception tokens at a DEX to return the flash loan and keep the profit
Recommendations	We recommend acknowledging this issue.
Comments / Resolution	Acknowledged

InceptionBaseAdapter

Issue_36	IBaseAdapter should add a storage gap
Severity	**Medium**
Description	The IBaseAdapter is a base contract to be inherited by the adapters of the Inception protocol. However, it does not implement a storage gap which could be problematic in case of future upgrades.
Recommendations	Consider adding a storage gap to the IBaseAdapter.
Comments / Resolution	Resolved by following the recommended mitigation.

InceptionEigenAdapter

Issue_37	withdraw() will queue less tokens than expected
Severity	**Medium**
Description	In the withdraw() function the shares to be queued are determined using the following function: *sharesToWithdraw[0] = _strategy.underlyingToShares(amount);* In EigenLayer there are two types of shares: ● depositShares - shares returned from *underlyingToShares*, but do not factor in the slashing. ● withdrawableShares - apply the slashing to the depositShares. Due to EigenLayer's slashing update - ELIP-002, in queueWithdrawal(), the depositShares need to be passed: */*** ** @param strategies The strategies to withdraw from* ** @param depositShares For each strategy, the number of deposit shares to withdraw. Deposit shares can* ** be queried via `getDepositedShares`.* ** NOTE: The number of shares ultimately received when a withdrawal is completed may be lower depositShares* ** if the staker or their delegated operator has experienced slashing.* ** @param __deprecated_withdrawer This field is ignored. The only party that may complete a withdrawal* ** is the staker that originally queued it. Alternate withdrawers are not supported.* **/* *struct QueuedWithdrawalParams {* *IStrategy[] strategies;* *uint256[] depositShares;*

	address __deprecated_withdrawer; *}* This means that the slashing factor will be applied internally, reducing the deposit shares to the withdrawable shares. When withdraw() is called, the operator intends to withdraw the inputted amount of tokens. However, using the *underlyingToShares* will result in withdrawing less tokens, because the slashing factor will further reduce the amount of tokens withdrawn. Consider the following scenario: 1) There is a slashing factor of 0.5e18. The operator wants to withdraw 100 tokens at a 1:1 strategy share ratio. 2) The *underlyingToShares* will return 100e18 shares, which are going to be inputted in queueWithdrawals(), as deposit shares. 3) After applying the slashing factor, only 50e18 tokens will be claimed, even though there was no slashing during the queue.
Recommendations	Consider using the convertToDepositShares() function on the sharesToWithdraw[0] to properly calculate the deposit shares.
Comments / Resolution	Fixed by following recommendations.

Issue_38	DoS When Depositing Into Strategies Deployed via EigenLayer Factory
Severity	**Low**
Description	The EigenLayer adapter includes a safety check that calls getTVLLimits() on the target strategy before depositing, to ensure the strategy's deposit cap is not exceeded. While this works for older, manually deployed strategies, it breaks compatibility with the newer ones. According to EigenLayer's documentation, only the original strategies implement getTVLLimits(). The newer strategies—deployed permissionlessly through the StrategyFactory—do not include this function. As a result, any call to getTVLLimits() on these newer strategies will revert. This creates a denial of service condition: any EigenLayer adapter that tries to deposit into one of these factory-deployed strategies will fail. The adapter reverts, and deposits into these strategies become impossible, even if the strategy itself is otherwise functional and safe to use. This check is not only incompatible but also unnecessary. EigenLayer's own documentation notes that the getTVLLimits() mechanism has never been actively used to enforce caps on canonical strategies. In practice, it has become a vestigial check.
Recommendations	To resolve this, it is recommended to remove the getTVLLimits() call from the EigenLayer adapters entirely. Doing so will ensure full compatibility with all current and future EigenLayer strategies, including those deployed through the official factory, while maintaining alignment with the platform's intended behavior.
Comments / Resolution	Fixed by following recommendations.

Issue_39	The withdrawer field in QueuedWithdrawalParams is now deprecated
Severity	**Informational**
Description	Due to the ELIP-002 being merged, the withdrawer field is now deprecated: */*** ** @param strategies The strategies to withdraw from* ** @param depositShares For each strategy, the number of deposit shares to withdraw. Deposit shares can* ** be queried via `getDepositedShares`.* ** NOTE: The number of shares ultimately received when a withdrawal is completed may be lower depositShares* ** if the staker or their delegated operator has experienced slashing.* ** @param __deprecated_withdrawer This field is ignored. The only party that may complete a withdrawal* ** is the staker that originally queued it. Alternate withdrawers are not supported.* **/* *struct QueuedWithdrawalParams {* *IStrategy[] strategies;* *uint256[] depositShares;* *address __deprecated_withdrawer;* *}*
Recommendations	Consider renaming it to *__deprecated_withdrawer.*
Comments / Resolution	Fixed by following recommendations.

Issue_40	Setting receiveAsTokens to false may disturb the withdrawalQueue
Severity	**Informational**
Description	The claim() function allows the operator to input receiveAsTokens: *bool receiveAsTokens = abi.decode(_data[2], (bool[]))[0];* However, setting the receiveAsTokens as false may disturb the withdrawal by wrongly recognizing slashing due to less tokens claimed. This may unnecessarily increase the user's waiting time.
Recommendations	Consider acknowledging this issue and ensuring that the operator always sets receiveAsTokens to true when claiming an epoch that was previously undelegated during the withdrawal flow. The receiveAsTokens flag should be set to false **only** when completing a queued withdrawal that was enqueued on the DelegationManager as a result of an undelegation or redelegation.
Comments / Resolution	Acknowledged.

Issue_41	Operators Can Now Be Fully Slashed On EL Therefore Stakers Should Be Given A Buffer Time To Exit
Severity	**Informational**
Description	In the new Eigen Layer upgrade operator can register to operator sets , and these operators in an operator set can be slashed by the AVS , the magnitude of slashing is decided by the operator (operator on EL) i.e if the slashable magnitude set by the operator is 50% then 50% of the operator's shares can be slashed at anytime by the AVS. Imagine a scenario where the inception vault's operator undelegates from the current operator and delegates to a new operator which has much higher slashable magnitude , though these external operators might be trusted by inception , the users might not align with the risk factors of these new operators . An immediate slash could result in a loss of 100% of the staker's stake and the staker here is the inception vault. Users should be made aware prior to the new delegations about the operator chosen that the vault would be delegated to , that way users not aligning with the terms of the new operator can perform a safe exit.
Recommendations	Depositors should have a buffer time where they can exit the system safely when an operator is selected for delegations.
Comments / Resolution	Acknowledged.

Issue_42	Delegations in EigenLayer will not work for Fee-on-Transfer tokens
Severity	**Informational**
Description	In the delegate() function in the InceptionEigenAdapter.sol → *function delegate(* *address operator,* *uint256 amount,* *bytes[] calldata _data* *) external override onlyTrustee whenNotPaused returns (uint256) {* *// depositIntoStrategy* *if (amount > 0 && operator == address(0)) {* *_beforeDepositAssetIntoStrategy(amount);* *// transfer from the vault* *_asset.safeTransferFrom(msg.sender, address(this), amount);* *// deposit the asset to the appropriate strategy* *return _strategy.sharesToUnderlying(* *_strategyManager.depositIntoStrategy(_strategy, _asset, amount)* *);* *}* If the strategy's underlying asset is a FoT token then the token amount actually received by the adapter would be less than "amount" , and therefore the depositIntoStrategy() call would revert. This would mean that it would be impossible to delegate funds for such strategies.
Recommendations	Consider acknowledging this issue if such tokens will not be used.
Comments / Resolution	Acknowledged.

InceptionEigenAdapterWrap

Issue_43	Wrong return value on withdraw
Severity	**Medium**
Description	Due to the fix to issue named "*withdraw() will queue less tokens than expected*", now the variable sharesToWithdraw is not in terms of actual withdrawal shares, but in terms of deposit shares. When sharesToWithdraw is converted into underlying in the return statement, the return value will be higher because the deposit shares may be higher than withdrawable shares if some slashing happened. This will cause the returned undelegated amount to be higher than the real undelegated amount, causing a revert on WithdrawalQueue later because the system will perceive that the assets being withdrawn are higher than the requested assets in that epoch.
Recommendations	Use withdrawableShares[0] as the argument on sharesToUnderlyingView in the return statement.
Comments / Resolution	

Issue_44	Incorrect TVL limit check
Severity	**Low**
Description	The strategy works with stETH, but the validation is performed using wstETH: *(uint256 maxPerDeposit, uint256 maxTotalDeposits) = _strategy.getTVLLimits();* *require(amount <= maxPerDeposit, ExceedsMaxPerDeposit(maxPerDeposit, amount));* *uint256 currentBalance = _asset.balanceOf(address(_strategy));* *require(currentBalance + amount <= maxTotalDeposits, ExceedsMaxTotalDeposited(maxTotalDeposits, currentBalance));*
Recommendations	Consider validating using the correct asset.
Comments / Resolution	Resolved by removing the TVL checks.

Issue_45	DoS on delegate() When Depositing More stETH Than Available
Severity	**Low**

Description	The InceptionEigenAdapterWrap contract handles deposits into EigenLayer strategies by receiving wstETH, unwrapping it to stETH, and then depositing the resulting stETH into the target strategy.
	`function delegate(`
	`    address operator,`
	`    uint256 amount,`
	`    bytes[] calldata _data`
	`) external override onlyTrustee whenNotPaused returns (uint256) {`
	`    // depositIntoStrategy`
	`    if (amount > 0 && operator == address(0)) {`
	`        _beforeDepositAssetIntoStrategy(amount);`
	`        // transfer from the vault`
	`        _asset.safeTransferFrom(msg.sender, address(this), amount);`
	`        amount = wrappedAsset().unwrap(amount);`
	`        // deposit the asset to the appropriate strategy`
	`        return wrappedAsset().getWstETHByStETH(_strategy.sharesToUnderlying(`
	`            _strategyManager.depositIntoStrategy(`
	`                _strategy, wrappedAsset().stETH(), amount`
	`            )`
	`        ));`
	`    }`
	`    // ...`
	`}`
	Due to internal rounding in the stETH contract, the unwrap call can return a slightly higher value (typically 1–2 wei more) than the actual amount of stETH received. This discrepancy leads to a revert in the subsequent depositIntoStrategy call, which tries to transfer more stETH than the contract actually holds. Since the ERC20 transfer call reverts on insufficient balance, this results in a full denial of service on the delegate() call.

	This issue blocks deposits and can render the adapter unusable on some deposits.
Recommendations	To fix this, the contract should compare the unwrapped amount against the actual stETH balance and, if needed, adjust the value to transfer the full available balance instead.
Comments / Resolution	Fixed by following the recommendations.

Issue_46	Unnecessary approval to strategyManager
Severity	**Informational**
Description	In the intialize function the _asset is approved to the strategyManager: *function initialize(* *...* *) public initializer {* *...* *// approve spending by strategyManager* *_asset.safeApprove(strategyManager, type(uint256).max);* This approval is unnecessary, as the InceptionEigenAdapter wrap will not deposit wstEth, into the strategy. Instead, it will only deposit stEth.
Recommendations	Consider removing the redundant approval
Comments / Resolution	Resolved, by following the recommended mitigation.

InceptionSymbioticAdapter

Issue_47	emergencyClaims are limited to one at a time until the active withdrawal is claimed from the SymbioticVault
Severity	**Informational**
Description	The epoch when active withdrawals can be claimed is assigned to the vault and assigned claimer on the withdrawals mapping: *withdrawals[vaultAddress][claimer] = vault.currentEpoch() + 1;* This mapping is not cleaned off until the active withdrawal is claimed from the SymbioticVault. *delete withdrawals[vaultAddress][claimer];* However, for emergencyWithdrawals, the same emergencyClaimer is reused as the claimer for the withdrawal. The following check will cause that all emergencyWithdrawals to revert while there is an active emergency withdrawal, effectively preventing additional emergency withdrawals: *require(* *withdrawals[vaultAddress][claimer] == 0,* *WithdrawalInProgress()* *);*
Recommendations	Consider enabling multiple emergency withdrawals simultaneously or acknowledging this issue.
Comments / Resolution	Acknowledged. The SymbioticAdapter is limited to only one emergencyWithdrawal at a time.

InceptionWstETHMellowAdapter

Issue_48	Emergency withdrawals that the MellowVault processes instantly will cause accounting issues
Severity	**High**

Description	Mellow vaults can instantly process a withdrawal if there are enough available assets, when this happens, undelegatedAmount may be 0 because nothing has been undelegated, Mellow vault will send the full requested amount to the claimer.
	
	When undelegatedAmount is 0, _removePendingClaimer is invoked:
	if (undelegatedAmount == 0) _removePendingClaimer(claimer);
	However, when emergency withdrawing, claimer is the emergencyClaimer, so, _removePendingClaimer() will end up adding the emergencyClaimer to the availableClaimers
	function _removePendingClaimer(address claimer) internal {
	delete _claimerVaults[claimer];
	pendingClaimers.remove(claimer);
	availableClaimers.push(claimer);
	}
	On the next normal withdrawal, the emeregencyClaimer will be added to the pendingClaimers because it is in the pendingClaimers.
	Having the emergencyClaimer as a claimer in the pendingClaimers will cause accounting issues for the internalBalance() because the emergencyClaimer's pending withdrawals will be counted twice, one as emergencyPendingWithdrawals and again as pendingWithdrawals.

Recommendations	Consider calling removePendingWithdrawals only when the emergency flag is false.
Comments / Resolution	Fixed by following the recommendations.

Issue_49	_claimerVaults is never cleared for emergency claimer in InceptionWstETHMellowAdapter
Severity	Low
Description	*The _claimerVaults mapping is used to track the vaults associated with each claimer. However, in the case of emergency claimers, it is set on every withdrawal request, regardless of whether the claimer is an emergency claimer or not.* *It is only cleared in the claim function, where it checks if the claimer is an emergency claimer.*
Recommendations	Consider ensuring that the _claimerVaults mapping is cleared for emergency claimers, or not setting it for emergency claimer at all.
Comments / Resolution	Acknowledged.

Issue_50	claimableWithdrawalAmount can't query the claimableAssetsOf() for emergency withdrawals
Severity	Informational

Description	The claimableWithdrawalAmount can only query the non-emergency withdrawals:
	function claimableWithdrawalAmount() public view returns (uint256 total) {
	return _claimableWithdrawalAmount(false);
	}

Recommendations	Consider adding a parameter to allow specifying if querying for normal or emergency withdrawals.
Comments / Resolution	Fixed by following the recommendations.

Issue_51	Insufficient validation when removing a Mellow vault
Severity	**Low**
Description	When removing a Mellow vault, the following validation is performed: *function removeVault(address vault) external onlyOwner {* *require(vault != address(0), ZeroAddress());* *require(* *getDeposited(vault) == 0 &&* *pendingWithdrawalAmount(vault, true) == 0 &&* *pendingWithdrawalAmount(vault, false) == 0,* *VaultNotEmpty()* *);* However, the pendingWithdrawalAmount() for a single vault does not consider the claimableWithdrawalAmount, which means, the vault may be removed even though there are assets to be claimed.
Recommendations	Consider adding a new function that allows to check the claimableAssetsOf on a single vault for normal and emergency situations.
Comments / Resolution	Fixed by following the recommendations.

Issue_52	Allocations are not updated when a vault is removed
Severity	**Informational**

Description	Consider the following scenario →
	 1.) In the InceptionWstETHMellowAdapter.sol a vault was removed using the removeVault() function , this vault had an allocation of 5 for example.
	
	 2.) After this tx a delegate() tx executes and invokes _delegateAuto() which calculates delegated amount based on current vault's allocation and the totalAllocation →
	 for (uint8 i = 0; i < mellowVaults.length; i++) {
	 uint256 allocation = allocations[address(mellowVaults[i])];
	 if (allocation > 0) {
	 *uint256 localBalance = (amount * allocation) / allocationsTotal;*
	
	 3.) But this calculation would also account for the allocation of the removed vault , this is because totalAllocations was not updated while removing the vault and because of this the active vaults will receive less deposit than expected.

Recommendations	When a vault is removed adjust the totalAllocation by the vault's allocation.
Comments / Resolution	Acknowledged.

MellowAdapterClaimer

Issue_53	SymbioticAdapterClaimer and MellowAdapterClaimer can never be paused
Severity	**Low**
Description	The SymbioticAdapterClaimer and MellowAdapterClaimer contracts are used to claim rewards from the Symbiotic and Mellow adapters, respectively. The contracts inherit from the PausableUpgradeable contract,which allows the contract to be paused and unpaused by the owner. However, the public pause and unpause functions are not implemented in the SymbioticAdapterClaimer and MellowAdapterClaimer contracts, which means that the contracts can never be paused or unpaused. This could lead to unexpected behavior, as the contracts can never be paused, and leading to broken intended functionality, as the owner cannot pause the contract in case of an emergency.
Recommendations	Consider implementing the pause and unpause functions in the SymbioticAdapterClaimer and MellowAdapterClaimer contracts, so that the owner can pause and unpause the contract in case of an emergency.
Comments / Resolution	Failed Resolution - Only owner can pause/unpause, but the AdapterClaimer's owner is the Adapter, and the Adapter has not a function to call pause() / unpause() on the Claimer.

Issue_54	The MellowAdapterClaimer does not have a storage gap
Severity	**Informational**
Description	The MellowAdapterClaimer is an upgradable contract but does not have a storage gap.
Recommendations	Consider adding a storage gap.
Comments / Resolution	Acknowledged.

SymbioticAdapterClaimer

Issue_55	DoS when using tokens that don’t return a bool on approval
Severity	**Low**
Description	In both SymbioticAdapterClaimer and MellowAdapterClaimer, the same approval is made during the execution of the constructor. *require(* *IERC20(asset).approve(_adapter, type(uint256).max),* *ApprovalFailed()* *);* However, some tokens don’t return a boolean on approvals, causing a revert and a DoS on deployment.
Recommendations	It’s recommended to use safeIncreaseAllowance.
Comments / Resolution	Fixed by following the recommendations.