YieldFi Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

Jorge

April 24, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

YieldFi is an asset management protocol that issues yield-bearing ERC-4626 vault tokens (`YTokens`), which represent a user's share of the trading profits and yield generated by the protocol. This audit focused on several new features, including integration with Chainlink CCIP for cross-chain messaging (in addition to the existing LayerZero integration), a new ERC20 token called `PerpetualBond`—a bond token that continuously generates yield while held, and a new two-step withdrawal process enabled by the `Manager` contract.

Another new feature allows users to deposit using any supported token through a two-step deposit process, also handled by the `Manager` contract. Currently, only stablecoins such as USDC, USDT, and DAI are supported. However, in the near future, YieldFi plans to add support for ETH, BTC, and various derivatives as assets within the YToken vaults.

4.1 Actors and Roles

- **1. Actors:**
  - **YieldFi protocol:** Deploys and manages the smart contracts.
  - **Off-chain services:** Execute deposits and withdrawals, and apply exchange rates when the deposited asset differs from the vault's YToken asset.
  - **Custodian:** Holds custody of user assets and generates yield.
  - **End users:** Interact with the protocol by depositing assets.
- **2. Roles:**
  - **ADMIN_ROLE:** Can set a new administrator contract. Has the ability to grant and revoke roles within the admin contract, and pause/unpause parts or all of the protocol. Also configures the other contracts.
  - **MINTER_AND_REDEEMER_ROLE:** Can set minimum deposit and withdrawal amounts in both `Manager` and `PerpetualBond`, execute user deposits and withdrawals, and configure gas fees in `YToken`.

- **COLLATERAL_MANAGER_ROLE:** Responsible for syncing asset/share state across chains, withdrawing assets to custody wallets, and approving `YToken` to spend `LockBox` assets.
- **REWARDER_ROLE:** Can distribute yield and rewards for both `PerpetualBond` and `YToken`.
- **MANAGER_ROLE:** Assigned to the `Manager` contract; can mint and burn `Receipt` tokens.
- **BRIDGE_ROLE:** Assigned to bridge contracts (`BridgeCCIP`, `BridgeLR`, and `BridgeMB`); can call `Manager::manageAssetAndShares` and `LockBox::unlock`.
- **BOND_ROLE:** Assigned to the `PerpetualBond` contract; can call `Manager::withdrawBondYield` and mint/burn `BondReceipt` tokens.
- **LOCKBOX_ROLE:** Assigned to the `LockBox` contract; can call `Manager::manageAssetAndShares` and `LockBox::unlock`.

4.2 Key Components

- **YTokens:** ERC-4626 vault tokens that track users' share of protocol yield and profits.
- **Manager:** Facilitates two-step deposits and withdrawals, and enforces protocol rules around minimum amounts and yield processing.
- **Off-chain service:** Executes orders, performs token conversions, and handles pricing logic.
- **PerpetualBond:** A bond-like ERC20 token that continuously accrues yield while held.

4.3 Centralization Risks

As highlighted above, the protocol relies on multisig wallets to perform core operations such as trading. This introduces a high degree of trust in these entities to act honestly and maintain operational integrity. These wallets, along with the off-chain services responsible for price discovery and executing withdrawals, must be secured with great care.

If any of these components were to be compromised, it could jeopardize the safety of user funds and the integrity of the protocol. Cyfrin recommends that the YieldFi team remain vigilant and adopt modern operational security best practices to mitigate risks in these areas.

5 Audit Scope

```
contracts/administrator/Access.sol
contracts/administrator/Administrator.sol
contracts/bridge/ccip/BridgeCCIP.sol
contracts/bridge/Bridge.sol
contracts/bridge/BridgeLR.sol
contracts/bridge/BridgeMB.sol
contracts/core/l1/LockBox.sol
contracts/core/l1/Yield.sol
contracts/core/tokens/YToken.sol
contracts/core/tokens/YTokenL2.sol
contracts/core/BondReceipt.sol
contracts/core/Manager.sol
contracts/core/MPC.sol
contracts/core/PerpetualBond.sol
contracts/core/Receipt.sol
contracts/core/SwapHelper.sol
contracts/libs/Codec.sol
contracts/libs/Common.sol
```

```
contracts/libs/Constants.sol
contracts/libs/Helpers.sol
```

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the YieldFi smart contracts provided by YieldFi. In this period, a total of 29 issues were found.

During the audit, two critical severity issues were uncovered, both related to the new CCIP messaging system. The first was a lack of validation of the sender chain and address, allowing anyone to pass messages to the `BridgeCCIP` contract and mint/burn or lock/unlock any number of YToken shares. The second involved improper decoding of message data, causing all CCIP messages to revert, as the destination chain ID used by CCIP could not fit into the 32-bit value it was being decoded into.

The high severity finding involved a bug where, in the worst case, tokens from the wrong owner could be burned when withdrawing on someone else's behalf using approvals.

Five medium severity issues were also identified, including a bypass of the blacklist for `PerpetualBond` tokens, an edge case where a 0% fee could result in a revert or, worst case, a 100% fee, a rounding issue in the L2 ERC-4626 vaults, missing oracle liveness checks, and scenarios where user funds could become locked.

Several low and informational findings were also reported, though they were not as severe.

The test suite was well written and covered most of the relevant scenarios effectively.

During the audit one extra PR and a commit, `d082bfd`, with new features were also reviewed by Cyfrin.

After the audit, the YieldFi team discovered an issue related to fee calculation, where the protocol fee was being applied to the total amount, including the gas fee, during withdrawals. This resulted in users being charged a slightly higher fee than intended. The issue was addressed and resolved in commits `9f45088` and `58a3949`.

Additionally, the YieldFi team decided to deploy new contracts rather than upgrading the existing ones already deployed. As part of this change, all legacy state variables were removed from the contracts in commit `37d3e79`.

Summary

Project Name	YieldFi
Repository	contracts
Commit	40caad6c6062...
Audit Timeline	Apr 1st - Apr 10th
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	1
Medium Risk	5
Low Risk	7
Informational	10
Gas Optimizations	4
Total Issues	29

Summary of Findings

[C-1] Missing source validation in CCIP message handling	Resolved
[C-2] All CCIP messages reverts when decoded	Resolved
[H-1] Incorrect `owner` passed to `Manager::redeem` in YToken withdrawal flow	Resolved
[M-1] Commented-out blacklist check allows restricted transfers	Resolved
[M-2] `Manager::_transferFee` returns invalid `feeShares` when `fee` is zero	Resolved
[M-3] `YtokenL2::previewMint` and `YTokenL2::previewWithdraw` round in favor of user	Resolved
[M-4] Missing L2 sequencer uptime check in `OracleAdapter`	Resolved
[M-5] Direct YToken deposits can lock funds below minimum withdrawal threshold	Resolved
[L-1] Hardcoded `extraArgs` violates CCIP best practices	Resolved
[L-2] Static `gasLimit` will result in overpayment	Resolved
[L-3] Unverified `_receiver` can cause irrecoverable token loss	Resolved
[L-4] Hardcoded CCIP `feeToken` prevents LINK discount usage	Resolved
[L-5] Chainlink router configured twice	Resolved
[L-6] Missing vesting check in `PerpetualBond::setVestingPeriod`	Resolved
[L-7] Balance check for yield claims in `PerpetualBond::_validate` can be easily bypassed	Resolved
[I-1] `PerpetualBond.epoch` not updated after yield distribution	Resolved
[I-2] Order not eligible at `eligibleAt`	Resolved
[I-3] `_receiverGas` check excludes minimum acceptable value	Resolved
[I-4] Unused errors	Resolved
[I-5] Potential risk if callback logic is enabled in the future	Acknowledged
[I-6] Lack of `_disableInitializers` in upgradeable contracts	Resolved
[I-7] Unused imports	Resolved
[I-8] Unused constants	Resolved

[I-9] Lack of event emissions on important state changes	Resolved
[I-10] Access to `LockBox::unlock` doesn't follow principle of least privilege	Resolved
[G-1] `BridgeCCIP.isL1` can be immutable	Resolved
[G-2] `bondFaceValue` read in `PerpetualBond::_convertToBond` can be cached	Resolved
[G-3] Unnecessary external call in `YToken::_decimalsOffset` and `YTokenL2::_decimalsOffset`	Acknowledged
[G-4] Order read twice in `Manager::executeOrder`	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Missing source validation in CCIP message handling

Description: YieldFi integrates with Chainlink CCIP to facilitate cross-chain transfers of its yield tokens (`YToken`). This functionality is handled by the `BridgeCCIP` contract, which manages token accounting for these transfers.

However, in the `BridgeCCIP::_ccipReceive` function, there is no validation of the message sender from the source chain:

```
/// handle a received message
function _ccipReceive(Client.Any2EVMMessage memory any2EvmMessage) internal override {
    bytes memory message = abi.decode(any2EvmMessage.data, (bytes)); // abi-decoding of the sent text
    BridgeSendPayload memory payload = Codec.decodeBridgeSendPayload(message);
    bytes32 _hash = keccak256(abi.encode(message, any2EvmMessage.messageId));
    require(!processedMessages[_hash], "processed");

    processedMessages[_hash] = true;

    require(payload.amount > 0, "!amount");

    ...
}
```

As a result, an attacker could craft a malicious `Any2EVMMessage` containing valid data and trigger the minting or unlocking of arbitrary tokens by sending it through CCIP to the `BridgeCCIP` contract.

Impact: An attacker could drain the bridge of tokens on L1 or mint an unlimited amount of tokens on L2. While a two-step redeem process offers some mitigation, such an exploit would still severely disrupt the protocol's accounting and could be abused when claiming yield for example.

Recommended Mitigation: Consider implementing validation to ensure that messages are only accepted from trusted peers on the source chain:

```
mapping(uint64 sourceChain => mapping(address peer => bool allowed)) public allowedPeers;
...
function _ccipReceive(
    Client.Any2EVMMessage memory any2EvmMessage
) internal override {
    address sender = abi.decode(any2EvmMessage.sender, (address));
    require(allowedPeers[any2EvmMessage.sourceChainSelector][sender],"allowed");
    ...
```

YieldFi: Fixed in commit a03341d

Cyfrin: Verified. `sender` is now verified to be a trusted sender.

7.1.2 All CCIP messages reverts when decoded

Description: YieldFi has integrated Chainlink CCIP alongside its existing LayerZero support to enable cross-chain token transfers using multiple messaging protocols. To support this, a custom message payload is used to indicate the token transfer. This payload is decoded in `Codec::decodeBridgeSendPayload` as follows:

```
(uint32 dstId, address to, address token, uint256 amount, bytes32 trxnType) = abi.decode(_data,
↪  (uint32, address, address, uint256, bytes32));
```

This same decoding logic is reused for CCIP message processing.

However, Chainlink uses a `uint64` for `dstId`, and their chain IDs (e.g., Ethereum mainnet) all exceed the `uint32` range. For instance, Ethereum's CCIP chain ID is `5009297550715157269`, which is well beyond the limits of `uint32`.

Impact: All CCIP messages will revert during decoding due to the overflow when casting a `uint64` value into a `uint32`. Since the contract is not upgradeable, failed messages cannot be retried, resulting in permanent loss of funds—tokens may be either locked or burned depending on the sending logic.

Proof of Concept: Attempting to process a message with `dstId = 5009297550715157269` in the `CCIP Receive: Should handle received message successfully` test causes the transaction to revert silently. The same behavior is observed when manually decoding a 64-bit value as a 32-bit integer using Remix.

Recommended Mitigation: Consider updating the type of `dstId` to `uint64` to match the Chainlink format. This change should be safe, as `dstId` is not used after decoding in the current LayerZero integration.

YieldFi: Fixed in commit `14fc17a`

Cyfrin: Verified. `dstId` is now a `uint64` in `Codec.BridgeSendPayload`.

7.2 High Risk

7.2.1 Incorrect `owner` passed to `Manager::redeem` in YToken withdrawal flow

Description: YieldFi's yield tokens (`YTokens`) implement a more complex withdrawal mechanism than a standard ERC-4626 vault. Instead of executing withdrawals immediately, they defer them to a central `Manager` contract, which queues the request for off-chain processing and later execution on-chain.

As with any ERC-4626 vault, third parties are allowed to initiate a withdrawal or redemption on behalf of a user, provided the appropriate allowances are in place.

However, in `YToken::_withdraw`, the wrong address is passed to the `manager.redeem` function. The same issue is also present in `YTokenL2::_withdraw`:

```
// Override _withdraw to request funds from manager
function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)
↪  internal override nonReentrant notPaused {
    require(receiver != address(0) && owner != address(0) && assets > 0 && shares > 0, "!valid");
    require(!IBlackList(administrator).isBlackListed(caller) &&
    ↪  !IBlackList(administrator).isBlackListed(receiver), "blacklisted");
    if (caller != owner) {
        _spendAllowance(owner, caller, shares);
    }
    // Instead of burning shares here, just redirect to Manager
    // The share burning will happen during order execution
    // Don't update totAssets here either, as the assets haven't left the system yet
    // @audit-issue `msg.sender` passed as owner
    IManager(manager).redeem(msg.sender, address(this), asset(), shares, receiver, address(0), "");
}
```

In this call, `msg.sender` is passed as the `owner` to `manager.redeem`, even though the correct `owner` is already passed into `_withdraw`. This works as expected when `msg.sender == owner`, but fails in delegated withdrawal scenarios where a third party is acting on the owner's behalf. In such cases, the `manager.redeem` call may revert, or worse, may burn the wrong user's tokens if `msg.sender` happens to have shares.

Impact: When a third party initiates a withdrawal on behalf of another user (`caller != owner`), the incorrect owner is passed to `manager.redeem`. This can cause the call to revert, blocking the withdrawal. In a worst-case scenario, if `msg.sender` (the caller) also holds shares, it may result in unintended burning of their tokens instead of the intended owner's.

Proof of Concept: Place the following test in `yToken.ts` under `describe("Withdraw and Redeem")`, it should pass but fails with `"!balance"`:

```
it("Should handle redeem request through third party", async function () {
  // Grant manager role to deployer for manager operations
  await administrator.grantRoles(MINTER_AND_REDEEMER, [deployer.address]);

  const sharesToRedeem = toN(50, 18); // 18 decimals for shares

  await ytoken.connect(user).approve(u1.address, sharesToRedeem);

  // Spy on manager.redeem call
  const redeemTx = await ytoken.connect(u1).redeem(sharesToRedeem, user.address, user.address);

  // Wait for transaction
  await redeemTx.wait();

  // to check if manager.redeem was called we can check the event of manager contract
  const events = await manager.queryFilter("OrderRequest");
  expect(events.length).to.be.greaterThan(0);
  expect(events[0].args[0]).to.equal(user.address); // owner, who's tokens should be burnt
  expect(events[0].args[1]).to.equal(ytoken.target); // yToken
  expect(events[0].args[2]).to.equal(usdc.target); // Asset
```

```
  expect(events[0].args[4]).to.equal(sharesToRedeem); // Amount
  expect(events[0].args[3]).to.equal(user.address); // Receiver
  expect(events[0].args[5]).to.equal(false); // isDeposit (false for redeem)
});
```

Recommended Mitigation: Pass the correct `owner` to `manager.redeem` in both `YToken::_withdraw` and `YTokenL2::_withdraw`, instead of using `msg.sender`.

YieldFi: Fixed in commit `adbb6fb`

Cyfrin: Verified. `owner` is now passed to `manager.redeem`.

7.3 Medium Risk

7.3.1 Commented-out blacklist check allows restricted transfers

Description: In `PerpetualBond::_update`, the line intended to restrict transfers between non-blacklisted users is currently commented out:

```
function _update(address from, address to, uint256 amount) internal virtual override {
    // Placeholder for Blacklist check
    // require(!IBlackList(administrator).isBlackListed(from) &&
    ↪  !IBlackList(administrator).isBlackListed(to), "blacklisted");
```

This effectively disables blacklist enforcement on transfers of `PerpetualBond` tokens.

Impact: Blacklisted addresses can freely hold and transfer `PerpetualBond` tokens, bypassing any intended access control or compliance restrictions.

Recommended Mitigation: Uncomment the blacklist check in `_update` to enforce transfer restrictions for blacklisted users.

YieldFi: Fixed in commit a820743

Cyfrin: Verified. Line doing the blacklist check is now uncommented.

7.3.2 `Manager::_transferFee` returns invalid `feeShares` when `fee` is zero

Description: When a user deposits directly into `Manager::deposit`, the protocol fee is calculated via the `Manager::_transferFee` function:

```
function _transferFee(address _yToken, uint256 _shares, uint256 _fee) internal returns (uint256) {
    if (_fee == 0) {
        return _shares;
    }
    uint256 feeShares = (_shares * _fee) / Constants.HUNDRED_PERCENT;

    IERC20(_yToken).safeTransfer(treasury, feeShares);

    return feeShares;
}
```

The issue is that when `_fee == 0`, the function returns the full `_shares` amount instead of returning `0`. This leads to incorrect logic downstream in `Manager::_deposit`, where the result is subtracted from the total shares:

```
// transfer fee to treasury, already applied on adjustedShares
uint256 adjustedFeeShares = _transferFee(order.yToken, adjustedShares, _fee);

// Calculate adjusted gas fee shares
uint256 adjustedGasFeeShares = (_gasFeeShares * order.exchangeRateInUnderlying) / currentExchangeRate;

// transfer gas to caller
IERC20(order.yToken).safeTransfer(_caller, adjustedGasFeeShares);

// remaining shares after gas fee
uint256 sharesAfterAllFee = adjustedShares - adjustedFeeShares - adjustedGasFeeShares;
```

If `_fee == 0`, the `adjustedFeeShares` value will incorrectly equal `adjustedShares`, causing `sharesAfterAllFee` to underflow (revert), assuming `adjustedGasFeeShares` is non-zero.

Impact: Deposits into the `Manager` contract with a fee of zero will revert if any gas fee is also deducted. In the best-case scenario, the deposit fails. In the worst case—if the subtraction somehow passes unchecked—it could result in zero shares being credited to the user.

Recommended Mitigation: Update `_transferFee` to return `0` when `_fee == 0`, to ensure downstream calculations behave correctly:

```
  if (_fee == 0) {
-     return _shares;
+     return 0;
  }
```

YieldFi: Fixed in commit 6e76d5b

Cyfrin: Verified. `_transferFee` now returns `0` when `_fee = 0`

7.3.3 `YtokenL2::previewMint` and `YTokenL2::previewWithdraw` round in favor of user

Description: For the L2 `YToken` contracts, assets are not managed directly. Instead, the vault's exchange rate is provided by an oracle, using the exchange rate from L1 as the source of truth.

This architectural choice requires custom implementations of functions like `previewMint`, `previewDeposit`, `previewRedeem`, and `previewWithdraw`, as well as the internal `_convertToShares` and `_convertToAssets`. These have been re-implemented to rely on the oracle-provided exchange rate instead of local accounting.

However, both `previewMint` and `previewWithdraw` currently perform rounding in favor of the user:

- `YTokenL2::previewMint`:

```
// Calculate assets based on exchange rate
return (grossShares * exchangeRate()) / Constants.PINT;
```

- `YTokenL2::previewWithdraw`:

```
// Calculate shares needed for requested assets based on exchange rate
uint256 sharesWithoutFee = (assets * Constants.PINT) / exchangeRate();
```

This behavior contradicts the security recommendations in EIP-4626, which advise rounding in favor of the vault to prevent value leakage.

Impact: By rounding in favor of the user, these functions allow users to receive slightly more shares or assets than they should. While the two-step withdrawal process limits the potential for immediate exploitation, this rounding error can result in a slow and continuous value leak from the vault—especially over many transactions or in the presence of automation.

Recommended Mitigation: Update `previewMint` and `previewWithdraw` to round in favor of the vault. This can be done by adopting the modified `_convertToShares` and `_convertToAssets` functions with explicit rounding direction, similar to the approach used in the OpenZeppelin ERC-4626 implementation.

YieldFi: Fixed in commit a820743

Cyfrin: Verified. the preview functions now utilizes `_convertToShares` and `_convertToAssets` with the correct rounding direction.

7.3.4 Missing L2 sequencer uptime check in `OracleAdapter`

Description: On L2, the `YToken` exchange rate is provided by custom Chainlink oracles. The exchange rate is queried in `OracleAdapter::fetchExchangeRate`:

```
function fetchExchangeRate(address token) external view override returns (uint256) {
    address oracle = oracles[token];
    require(oracle != address(0), "Oracle not set");

    (, /* uint80 roundId */ int256 answer, , /* uint256 startedAt */ uint256 updatedAt /* uint80
    ↪  answeredInRound */, ) = IOracle(oracle).latestRoundData();

    require(answer > 0, "Invalid price");
```

```
    require(updatedAt > 0, "Round not complete");
    require(block.timestamp - updatedAt < staleThreshold, "Stale price");

    // Get decimals and normalize to 1e18 (PINT)
    uint8 decimals = IOracle(oracle).decimals();

    if (decimals < 18) {
        return uint256(answer) * (10 ** (18 - decimals));
    } else if (decimals > 18) {
        return uint256(answer) / (10 ** (decimals - 18));
    } else {
        return uint256(answer);
    }
}
```

However, this protocol is intended to be deployed on L2 networks such as Arbitrum and Optimism, where it's important to verify that the sequencer is up. Without this check, if the sequencer goes down, the latest round data may appear fresh, when in fact it is stale, for advanced users submitting transactions from L1.

Impact: If the L2 sequencer goes down, oracle data will stop updating. Actually stale prices can appear fresh and be relied upon incorrectly. This could be exploited if significant price movement occurs during the downtime.

Recommended Mitigation: Consider implementing a sequencer uptime check, as shown in the Chainlink example, to prevent usage of stale oracle data during sequencer downtime.

YieldFi: Fixed in commits `bb26a71` and `e9c160f`

Cyfrin: Verified. Sequencer uptime is now verified on L2s.

7.3.5 Direct YToken deposits can lock funds below minimum withdrawal threshold

Description: In `Manager::deposit`, there is a check enforcing a minimum deposit amount inside `Manager::_validate`:

```
uint256 normalizedAmount = _normalizeAmount(_yToken, _asset, _amount);
require(IERC4626(_yToken).convertToShares(normalizedAmount) >= minSharesInYToken[_yToken],
↪  "!minShares");
```

A similar check exists in the redeem flow, again via `Manager::_validate`:

```
require(_amount >= minSharesInYToken[_yToken], "!minShares");
```

However, no such minimum is enforced when depositing directly into a `YToken`. In both `YToken::_deposit` and `YTokenL2::_deposit`, the only requirement is:

```
require(receiver != address(0) && assets > 0 && shares > 0, "!valid");
```

As a result, a user could deposit an amount that results in fewer shares than `minSharesInYToken[_yToken]`, which cannot be withdrawn through the `Manager` due to its minimum withdrawal check, effectively locking their funds.

Impact: Users can bypass the minimum share threshold by depositing directly into a `YToken`. If the resulting share amount is below the minimum allowed for withdrawal via the `Manager`, the user will be unable to exit their position. This can lead to unintentionally locked funds and a poor user experience.

Recommended Mitigation: Consider enforcing the `minSharesInYToken[_yToken]` threshold in `YToken::_deposit` and `YTokenL2::_deposit` to prevent deposits that are too small to be withdrawn. Additionally, consider validating post-withdrawal balances to ensure users are not left with non-withdrawable "dust" (i.e., require remaining shares to be either `0` or > `minSharesInYToken[_yToken]`).

YieldFi: Fixed in commit `221c7d0`

Cyfrin: Verified. Minimum shares is now verified in the YToken contracts. Manager also verifies that there is no dust left after redeem.

7.4 Low Risk

7.4.1 Hardcoded `extraArgs` violates CCIP best practices

Description: When sending cross-chain messages via CCIP, Chainlink recommends keeping the `extraArgs` parameter mutable to allow for future upgrades or configuration changes, as outlined in their best practices.

However, this recommendation is not followed in `BridgeCCIP::send`, where `extraArgs` is hardcoded:

```
// Sends the message to the destination endpoint
Client.EVM2AnyMessage memory evm2AnyMessage = Client.EVM2AnyMessage({
    receiver: abi.encode(_receiver), // ABI-encoded receiver address
    data: abi.encode(_encodedMessage), // ABI-encoded string
    tokenAmounts: new Client.EVMTokenAmount[](0), // Empty array indicating no tokens are being sent
    // @audit-issue `extraArgs` hardcoded
    extraArgs: Client._argsToBytes(Client.EVMExtraArgsV2({ gasLimit: 200_000, allowOutOfOrderExecution:
    ↪  true })),
    feeToken: address(0) // For msg.value
});
```

Impact: Because `extraArgs` is hardcoded, any future changes would require deploying a new version of the bridge contract.

Recommended Mitigation: Consider making `extraArgs` mutable by either passing it as a parameter to the `send` function or deriving it from configurable contract storage.

YieldFi: Fixed in commits `3cc0b23` and `fd4b7ab5`

Cyfrin: Verified. `extraArgs` is now passed as a parameter to the call.

7.4.2 Static `gasLimit` will result in overpayment

Description: Since unspent gas is not refunded, Chainlink recommends carefully setting the `gasLimit` within the `extraArgs` parameter to avoid overpaying for execution.

In `BridgeCCIP::send`, the `gasLimit` is hardcoded to `200_000`, which is also Chainlink's default:

```
extraArgs: Client._argsToBytes(Client.EVMExtraArgsV2({ gasLimit: 200_000, allowOutOfOrderExecution:
↪  true })),
```

This hardcoded value directly affects every user bridging tokens, as they will be consistently overpaying for execution costs on the destination chain.

Recommended Mitigation: A more efficient approach would be to measure the gas usage of the `_ccipReceive` function using tools like Hardhat or Foundry and set the `gasLimit` accordingly—adding a margin for safety. This ensures that the protocol avoids overpaying for gas on every cross-chain message.

This issue also reinforces the importance of making `extraArgs` mutable, so the gas limit and other parameters can be adjusted if execution costs change over time (e.g., due to protocol upgrades like EIP-1884).

YieldFi: Fixed in commit `3cc0b23`

Cyfrin: Verified. `extraArgs` is now passed as a parameter to the call.

7.4.3 Unverified `_receiver` can cause irrecoverable token loss

Description: When a user bridges their YTokens using CCIP, they call `BridgeCCIP::send`. One of the parameters passed to this function is `_receiver`, which is intended to be the destination contract on the receiving chain:

```
function send(address _yToken, uint64 _dstChain, address _to, uint256 _amount, address _receiver)
↪  external payable notBlacklisted(msg.sender) notBlacklisted(_to) notPaused {
    require(_amount > 0, "!amount");
    require(lockboxes[_yToken] != address(0), "!token !lockbox");
    require(IERC20(_yToken).balanceOf(msg.sender) >= _amount, "!balance");
```

```
    require(_to != address(0), "!receiver");
    require(tokens[_yToken][_dstChain] != address(0), "!destination");

    bytes memory _encodedMessage = abi.encode(_dstChain, _to, tokens[_yToken][_dstChain], _amount,
    ↪  Constants.BRIDGE_SEND_HASH);

    // Sends the message to the destination endpoint
    Client.EVM2AnyMessage memory evm2AnyMessage = Client.EVM2AnyMessage({
        // @audit-issue `_receiver` not verified
        receiver: abi.encode(_receiver), // ABI-encoded receiver address
        data: abi.encode(_encodedMessage), // ABI-encoded string
        tokenAmounts: new Client.EVMTokenAmount[](0), // Empty array indicating no tokens are being sent
        extraArgs: Client._argsToBytes(Client.EVMExtraArgsV2({ gasLimit: 200_000,
        ↪  allowOutOfOrderExecution: true })),
        feeToken: address(0) // For msg.value
    });
```

However, the `_receiver` parameter is not validated. If the user provides an incorrect or malicious address, the message may be delivered to a contract that cannot handle it, resulting in unrecoverable loss of the bridged tokens.

Recommended Mitigation: Validate the `_receiver` address against a trusted mapping, such as the `peers` mapping mentioned in a previous finding, to ensure it corresponds to a legitimate contract on the destination chain.

YieldFi: Fixed in commit a03341d

Cyfrin: Verified. `_receiver` is now verified to be a trusted peer.

7.4.4 Hardcoded CCIP `feeToken` prevents LINK discount usage

Description: In `BridgeCCIP::send`, the `feeToken` parameter is hardcoded:

```
// Sends the message to the destination endpoint
Client.EVM2AnyMessage memory evm2AnyMessage = Client.EVM2AnyMessage({
    receiver: abi.encode(_receiver), // ABI-encoded receiver address
    data: abi.encode(_encodedMessage), // ABI-encoded string
    tokenAmounts: new Client.EVMTokenAmount[](0), // Empty array indicating no tokens are being sent
    extraArgs: Client._argsToBytes(Client.EVMExtraArgsV2({ gasLimit: 200_000, allowOutOfOrderExecution:
    ↪  true })),
    // @audit-issue hardcoded fee token
    feeToken: address(0) // For msg.value
});
```

Chainlink CCIP supports paying fees using either the native gas token or `LINK`. By hardcoding `feeToken = address(0)`, the protocol forces all users to pay with the native gas token, removing flexibility.

This design choice simplifies implementation but has cost implications: CCIP offers a 10% fee discount when using `LINK`, so users holding `LINK` are unable to take advantage of these reduced fees.

Recommended Mitigation: Consider allowing users to choose their preferred payment token—either `LINK` or native gas—based on their individual cost and convenience preferences.

YieldFi: Fixed in commits 3cc0b23 and e9c160f

Cyfrin: Verified.

7.4.5 Chainlink router configured twice

Description: In `BridgeCCIP`, there is a dedicated storage slot for the CCIP router address, `router`:

```
contract BridgeCCIP is CCIPReceiver, Ownable {
    address public router;
```

This value can be updated by the admin through BridgeCCIP::setRouter:

```
function setRouter(address _router) external onlyAdmin {
    require(_router != address(0), "!router");
    router = _router;
    emit SetRouter(msg.sender, _router);
}
```

The router is then used in BridgeCCIP::send to send messages via CCIP:

```
IRouterClient(router).ccipSend{ value: msg.value }(_dstChain, evm2AnyMessage);
```

However, the inherited CCIPReceiver contract already defines an immutable router address (i_ccipRouter), which is used to validate that incoming CCIP messages originate from the correct router.

This introduces an inconsistency: if BridgeCCIP.router is changed, the contract will continue to *send* messages via the new router, but *receive* messages only from the original, immutable i_ccipRouter. This mismatch could break cross-chain communication or make message delivery non-functional.

Recommended Mitigation: Since the router address in CCIPReceiver is immutable, any future change to the router would already require redeployment of the BridgeCCIP contract. Therefore, the router storage slot and the setRouter function in BridgeCCIP are redundant and potentially misleading. We recommend removing both and relying exclusively on the i_ccipRouter value inherited from CCIPReceiver.

YieldFi: Fixed in commit 3cc0b23

Cyfrin: Verified. router removed and i_ccipRouter used from the inherited contract.

7.4.6 Missing vesting check in PerpetualBond::setVestingPeriod

Description: Both YToken and PerpetualBond support reward vesting through a configurable vesting period. The admin can update this period via the setVestingPeriod function. However, there is an inconsistency in how the two contracts validate changes to the vesting period:

- YToken::setVestingPeriod includes a check to ensure that no rewards are currently vesting:

```
function setVestingPeriod(uint256 _vestingPeriod) external onlyAdmin {
    require(getUnvestedAmount() == 0, "!vesting");
    require(_vestingPeriod > 0, "!vestingPeriod");
    vestingPeriod = _vestingPeriod;
}
```

- PerpetualBond::setVestingPeriod lacks this check:

```
function setVestingPeriod(uint256 _vestingPeriod) external onlyAdmin {
    // @audit-issue no check for `getUnvestedAmount() == 0`
    require(_vestingPeriod > 0, "!vestingPeriod");
    vestingPeriod = _vestingPeriod;
    emit VestingPeriodUpdated(_vestingPeriod);
}
```

This means the vesting period in PerpetualBond can be modified even while tokens are still vesting, which could lead to inconsistent or unexpected vesting behavior.

Recommended Mitigation: To align with the YToken implementation and ensure consistency, add a check in PerpetualBond::setVestingPeriod to ensure getUnvestedAmount() == 0 before allowing updates to the vesting period.

YieldFi: Fixed in commit f0bf88c

Cyfrin: Verified. unvestedAmount is now checked.

7.4.7 Balance check for yield claims in `PerpetualBond::_validate` can be easily bypassed

Description: In `PerpetualBond::_validate`, there's a check to ensure that users have a non-zero balance before claiming yield:

```
// Yield claim
require(balanceOf(_caller) > 0, "!bond balance"); // Caller must hold bonds to claim yield
require(accruedRewardAtCheckpoint[_caller] > 0, "!claimable yield"); // Must have claimable yield
```

However, this check can be bypassed by holding a trivial amount, such as 1 wei, of `PerpetualBond` tokens. A more meaningful check would ensure that the user's balance exceeds the `minimumTxnThreshold`, similar to how other parts of the contract enforce value-based thresholds.

Consider updating the balance check to compare against `minimumTxnThreshold` using the bond-converted value:

```
- require(balanceOf(_caller) > 0, "!bond balance");
+ require(_convertToBond(balanceOf(_caller)) > minimumTxnThreshold, "!bond balance");
```

Additionally, the second check on `accruedRewardAtCheckpoint[_caller]` is redundant, since `PerpetualBond::requestYieldClaim` already performs a value-based threshold check:

```
// Convert yield amount to bond tokens for threshold comparison
uint256 yieldInBondTokens = _convertToBond(claimableYieldAmount);

// Check if the yield claim is worth executing
require(yieldInBondTokens >= minimumTxnThreshold, "!min txn threshold");
```

This makes the `accruedRewardAtCheckpoint` check in `_validate` unnecessary.

YieldFi: Fixed in commit `f0bf88c`

Cyfrin: Verified. Balance check removed as the user might still have yield even if they have no tokens (sold/transferred). Yield check in `_validate` is also removed as it's redundant.

7.5 Informational

7.5.1 `PerpetualBond.epoch` not updated after yield distribution

Description: In `PerpetualBond::distributeBondYield` the caller is supposed to provide a `nonce` that matches `epoch + 1`:

```
function distributeBondYield(uint256 _yieldAmount, uint256 nonce) external notPaused onlyRewarder {
    require(nonce == epoch + 1, "!epoch");
```

However, `epoch` is never incremented afterwards, consider incrementing `epoch`.

YieldFi: Fixed in commit `5c1f0e7`

Cyfrin: Verified. `epoch` now is incremented with the new `nonce`.

7.5.2 Order not eligible at `eligibleAt`

Description: Both in `PerpetualBond::executeOrder` and `Manager::executeOrder` there's a check that the order executed is still eligible:

```
require(block.timestamp > order.eligibleAt, "!waitingPeriod");
```

`eligibleAt` indicates that the order should be eligible at this timestamp which is not what the check verifies. Consider changing > to >=:

```
- require(block.timestamp > order.eligibleAt, "!waitingPeriod");
+ require(block.timestamp >= order.eligibleAt, "!waitingPeriod");
```

YieldFi: Fixed in commit `e9c160f`

Cyfrin: Verified.

7.5.3 `_receiverGas` check excludes minimum acceptable value

Description: In the LayerZero bridge contracts `BridgeLR::send` and `BridgeMB::send`, there's a check to ensure the user has provided sufficient `_receiverGas`:

```
require(_receiverGas > MIN_RECEIVER_GAS, "!gas");
```

The variable name `MIN_RECEIVER_GAS` suggests that the specified amount should be *inclusive*, meaning the minimum acceptable value is valid. However, the current > check excludes `MIN_RECEIVER_GAS` itself. To align with the semantic expectation, consider changing the comparison to >=:

```
- require(_receiverGas > MIN_RECEIVER_GAS, "!gas");
+ require(_receiverGas >= MIN_RECEIVER_GAS, "!gas");
```

Same applies to the call `Bridge::setMIN_RECEIVER_GAS` and the check in `Bridge::quote` as well.

YieldFi: Fixed in commit `9aa242b`

Cyfrin: Verified.

7.5.4 Unused errors

Description: In the library `Common` there are two unused errors:

```
error SignatureVerificationFailed();
error BadSignature();
```

Consider removing these.

YieldFi: Fixed in commit 9aa242b

Cyfrin: Verified.

7.5.5 Potential risk if callback logic is enabled in the future

Description: Both the `Manager` and `PerpetualBond` contracts implement a two-step process for user interactions. As part of these calls, users can provide a `_callback` address and accompanying `_callbackData`. For example, here are the parameters for `Manager::deposit`:

```
function deposit(..., address _callback, bytes calldata _callbackData) external notPaused nonReentrant {
```

However, these parameters are currently not passed along when the request is stored, as shown later in `Manager::deposit`:

```
uint256 receiptId = IReceipt(receipt).mint(msg.sender, Order(..., address(0), ""));
```

Here, `address(0)` and empty `""` are hardcoded instead of using the user-supplied values.

Later, in the `executeOrder` flow (e.g., `Manager::executeOrder`), the callback is conditionally executed:

```
// Execute the callback
if (order.callback != address(0)) {
    (bool success, ) = order.callback.call(order.callbackData);
    require(success, "callback failed");
}
```

If the original user-provided `_callback` and `_callbackData` were passed through and used here, it would pose a serious security risk. Malicious users could exploit this to execute arbitrary external calls and potentially steal tokens that are approved to the `Manager` or `PerpetualBond` contracts.

If callback functionality is not currently intended, consider removing or disabling the `_callback` and `_callbackData` parameters entirely to avoid the risk of these being enabled in the future. Alternatively, ensure strict validation and access control if support for callbacks is added later.

YieldFi: Acknowledged.

7.5.6 Lack of `_disableInitializers` in upgradeable contracts

Description: YieldFi utilizes upgradeable contracts. It's best practice to disable the ability to initialize the implementation contracts.

Consider adding a constructor with the OpenZeppelin `_disableInitializers` in all the upgradeable contracts:

```
constructor() {
    _disableInitializers();
}
```

YieldFi: Fixed in commit 584b268

Cyfrin: Verified. Constructor with `_disableInitializers` added to all upgradeable contracts.

7.5.7 Unused imports

Description: Consider removing the following unused imports:

- contracts/bridge/Bridge.sol Line: 7
- contracts/bridge/Bridge.sol Line: 9
- contracts/bridge/Bridge.sol Line: 13

- contracts/bridge/Bridge.sol Line: 15
- contracts/bridge/Bridge.sol Line: 18
- contracts/bridge/Bridge.sol Line: 20
- contracts/bridge/BridgeMB.sol Line: 17
- contracts/bridge/ccip/BridgeCCIP.sol Line: 4
- contracts/bridge/ccip/BridgeCCIP.sol Line: 13
- contracts/core/Manager.sol Line: 6
- contracts/core/Manager.sol Line: 15
- contracts/core/Manager.sol Line: 17
- contracts/core/OracleAdapter.sol Line: 6
- contracts/core/PerpetualBond.sol Line: 7
- contracts/core/PerpetualBond.sol Line: 13
- contracts/core/interface/IPerpetualBond.sol Line: 4
- contracts/core/l1/LockBox.sol Line: 10
- contracts/core/l1/LockBox.sol Line: 13
- contracts/core/l1/Yield.sol Line: 5
- contracts/core/l1/Yield.sol Line: 10
- contracts/core/l1/Yield.sol Line: 11
- contracts/core/l1/Yield.sol Line: 12
- contracts/core/l1/Yield.sol Line: 13
- contracts/core/l1/Yield.sol Line: 14
- contracts/core/l1/Yield.sol Line: 16
- contracts/core/tokens/YToken.sol Line: 8
- contracts/core/tokens/YToken.sol Line: 14
- contracts/core/tokens/YTokenL2.sol Line: 12

YieldFi: Fixed in commit 8264429

Cyfrin: Verified.

7.5.8 Unused constants

Description: In `Constants.sol` there are a some unused constants, consider removing thses:

- #L21: `SIGNER_ROLE`
- #L38: `VESTING_PERIOD`
- #L41 `MAX_COOLDOWN_PERIOD`
- #L44: `MIN_COOLDOWN_PERIOD`
- #L47 `ETH_SIGNED_MESSAGE_PREFIX`
- #L50`REWARD_HASH`
- #L56-L59 `DEPOSIT`, `WITHDRAW`, `DEPOSIT_L2`, `WITHDRAW_L2`

YieldFi: Fixed in commit 125ec4a

Cyfrin: Verified.

7.5.9 Lack of event emissions on important state changes

Description: The following functions change state but doesn't emit an event. Consider emitting an event from the following:

- Access::setAdministrator
- Administrator::cancelAdminRole
- Administrator::cancelTimeLockUpdate
- Bridge::setMIN_RECEIVER_GAS
- BridgeMB::setManager
- BridgeCCIP::setManager
- Manager::setTreasury
- Manager::setReceipt
- Manager::setCustodyWallet
- Manager::setMinSharesInYToken
- OracleAdapter::setStaleThreshold
- LockBox::setManager
- YToken::setManager
- YToken::setYield
- YToken::setVestingPeriod
- YToken::setFee
- YToken::setGasFee
- YToken::updateTotalAssets
- YTokenL2::setManager
- YTokenL2::setFee
- YTokenL2::setGasFee

YieldFi: Fixed in commit b978ddf

Cyfrin: Verified.

7.5.10 Access to `LockBox::unlock` doesn't follow principle of least privilege

Description: The function LockBox::unlock has the modifier onlyBridgeOrLockBox which allows callers with either the role `BRIDGE_ROLE` or `LOCKBOX_ROLE` to access the call.

The function is however only called from the bridge contracts. Consider removing the access from the `LOCKBOX_ROLE` to follow principle of least privileges.

YieldFi: Fixed in commit f0c751a

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 `BridgeCCIP.isL1` can be immutable

Description: `BridgeCCIP.isL1` is only assigned in the constructor. Therefore it can be made immutable as immutable values are cheaper to read.

Consider making `BridgeCCIP.isL1` immutable.

YieldFi: Fixed in commit 823b010

Cyfrin: Verified.

7.6.2 `bondFaceValue` read in `PerpetualBond::_convertToBond` can be cached

Description: The storage value `bondFaceValue` is read twice in `PerpetualBond::__convertToBond`:

```
function _convertToBond(uint256 assetAmount) internal view returns (uint256) {
    if (bondFaceValue == 0) return 0; // Prevent division by zero
    return (assetAmount * 1e18) / bondFaceValue;
}
```

The value can be cached and only read once:

```
function _convertToBond(uint256 assetAmount) internal view returns (uint256) {
    // cache read
    uint256 _bondFaceValue = bondFaceValue;
    if (_bondFaceValue == 0) return 0; // Prevent division by zero
    return (assetAmount * 1e18) / _bondFaceValue;
}
```

YieldFi: Fixed in commit 823b010

Cyfrin: Verified.

7.6.3 Unnecessary external call in `YToken::_decimalsOffset` and `YTokenL2::_decimalsOffset`

Description: In `YToken::_decimalsOffset` and `YTokenL2::_decimalsOffset` the decimals of the underlying token is queried:

```
function _decimalsOffset() internal view virtual override returns (uint8) {
    return 18 - IERC20Metadata(asset()).decimals();
}
```

This value is however already stored in the OpenZeppelin base contract `ERC4626Upgradeable` and can be used instead of an external call.

YieldFi: Acknowledged.

7.6.4 Order read twice in `Manager::executeOrder`

Description: In `Manager::executeOrder` the order data is fetched from the Receipt:

```
Order memory order = IReceipt(receipt).readOrder(_receiptId);
require(block.timestamp > order.eligibleAt, "!waitingPeriod");
require(_fee <= Constants.ONE_PERCENT, "!fee");
if (order.orderType) {
    _deposit(msg.sender, _receiptId, _amount, _fee, _gas);
} else {
    _withdraw(msg.sender, _receiptId, _amount, _fee, _gas);
}
```

Then order is read again in both `Manager::_deposit`:

```
function _deposit(address _caller, uint256 _receiptId, uint256 _shares, uint256 _fee, uint256
↪  _gasFeeShares) internal {
    Order memory order = IReceipt(receipt).readOrder(_receiptId);
```

and `Manager::_withdraw`:

```
function _withdraw(address _caller, uint256 _receiptId, uint256 _assetAmountOut, uint256 _fee, uint256
↪  _gasFeeShares) internal {
    Order memory order = IReceipt(receipt).readOrder(_receiptId);
```

This extra read is unnecessary. Consider passing the `Order memory order` as a parameter to `Manager::_deposit` and `Manager::_withdraw` instead. Thus saving to read the data again from the receipt:

```
function _deposit(..., Order memory order) internal {

function _withdraw(..., Order memory order) internal {
```

YieldFi: Fixed in commit 823b010

Cyfrin: Verified.