Joint-Strategy Audit

1. Replace all .transfer() with .safeTransfer()

E.g.: such as in L593 of Joint.sol

```
IERC20(tokenA).transfer(address(providerA), balanceA);
```

Impact

Low

Recommendation

There is a chance that the transfer function of the ERC20 in this strategy could fail silently, which can occur with tokens that are not fully compliant with ERC20 that return "false" instead of reverting when a transfer fails. To mitigate this, check the return value of the transfer call or use OpenZeppelin's safeERC20.

2. Incorrect comment

Proof of concept

In Joint.sol, we have

```
uint256 internal constant RATIO_PRECISION = 1e4;
```

this means the following line has an incorrect comment. The comment should say 5%, not 0.1%

```
maxPercentageLoss = 500; // 0.1%
```

Impact

Informational

Recommendation

Fix typo

3. Comment typo

The revert text "sandwidched" in Joint.sol is a typo and should say "sandwiched" in four instances in Joint.sol

- https://github.com/fp-crypto/joint-strategy/blob/3c12bbff1f3af66599571e9f79d642ce62020d85/contracts/Joint.sol#L646-L647
- https://github.com/fp-crypto/joint-strategy/blob/3c12bbff1f3af66599571e9f79d642ce62020d85/contracts/Joint.sol#L668-L669

Impact

Informational

Recommendation

Fix typo

4. "now" keyword deprecated

The contract uses Solidity 0.6.12, so it supports the "now" keyword, but this keyword was deprecated in Solidity 0.7.0 and should be replaced with block.timestamp.

Proof of concept/Steps to Reproduce

The "now" keyword is used 5 times in Joint.sol when interacting with IUniswapV2Router02

Solidity docs suggesting change of "now" to "block.timestamp"
https://docs.soliditylang.org/en/v0.8.12/070-breaking-changes.html#how-to-update-your-code

block.timestamp should be used for long intervals.

Impact

Informational

Recommendation

Use block.timestamp instead of now, especially if upgrading solidity versions.

5. Missing sandwich attack warnings

The Joint.sol file has a **WARNING** comment about sandwich attacks for the `_closePosition` and `createLP` functions, but there is no warning on several other functions such as `sellCapital`, `swapTokenForTokenManually`, `removeLiquidityManually`. However all Yearn mainnet strategies use flashbots for private relays to protect against sandwich attacks.

Impact

Informational

Risk Breakdown

The assumptions for whether or not sandwich attacks are a risk should be more clearly defined. If the manual functions are at risk of sandwich attacks and need require statements to protect from this attack, a warning comment should be added with clarification that the require statements should prevent the attack.

Recommendation

Clarify assumptions of sandwich attacks

6. Typo in require statement

There is a typo where a line of code was copied but the variable was not renamed properly: https://github.com/fp-crypto/joint-strategy/blob/master/contracts/Joint.sol#L669

Proof of concept/Steps to Reproduce

```
require(expectedBalanceA <= balanceOfA(), "!sandwidched");
require(expectedBalanceA <= balanceOfB(), "!sandwidched");
```

The second line of these two should use expectedBalanceB, not expectedBalanceA, because it is compared to balanceOfB.

Impact

High

Risk Breakdown

balanceOfB is not properly checked, so a sneaky sandwich attack could happen if done only on tokenB

Recommendation

Fix the typo

7. Remove duplicate require checks

The `removeLiquidityManually` function of Joint.sol has a require function to check the amount received by the Uniswap removeLiquidity call. Uniswap already has a require function that can be used for this purpose that is already called, and we can save gas by removing this duplicate require function.

The require statements that Uniswap's `removeLiquidity` function already has built in are:
https://github.com/Uniswap/v2-periphery/blob/dda62473e2da448bc9cb8f4514dadda4aeede5f4/contracts/UniswapV2Router02.sol#L117-L118

The duplicate require statements in Joint.sol are:
https://github.com/fp-crypto/joint-strategy/blob/3c12bbff1f3af66599571e9f79d642ce62020d85/contracts/Joint.sol#L668-L669

Proof of concept/Steps to Reproduce

Original function in Joint.sol

```
function removeLiquidityManually(
    uint256 amount,
    uint256 expectedBalanceA,
    uint256 expectedBalanceB
) external onlyVaultManagers {
    IUniswapV2Router02(router).removeLiquidity(
        tokenA,
        tokenB,
        amount,
        0,
        0,
        address(this),
        now
    );
    require(expectedBalanceA <= balanceOfA(), "!sandwidched");
    require(expectedBalanceA <= balanceOfB(), "!sandwidched");
}
```

The same function with 2 require statements removed provides a gas savings

```
function removeLiquidityManually(
    uint256 amount,
    uint256 expectedBalanceA,
    uint256 expectedBalanceB
) external onlyVaultManagers {
    IUniswapV2Router02(router).removeLiquidity(
        tokenA,
        tokenB,
        amount,
        expectedBalanceA - balanceOfA(),
        expectedBalanceB - balanceOfB(),
        address(this),
```

```
            now
        );
    }
```

Impact

Gas Savings

Risk Breakdown

None

Recommendation

See code above

8. latestRoundData may return stale prices

In LPHedgingLib.sol, need to check that data is not stale: https://consensys.net/diligence/audits/2021/09/fei-protocol-v2-phase-1/#chainlinkoraclewrapper-latestrounddata-might-return-stale-results

Proof of concept/Steps to Reproduce

The current code does not check the round or timestamp of the latestRoundData call

```
(, int256 answer, , , ) = pp.latestRoundData();
return uint256(answer);
```

The strategy code may have borrowed the code directly from Hegic's PriceCalculator.sol contract at 0x1BA4b447d0dF64DA64024e5Ec47dA94458C1e97f, which makes this same mistake

```
function _currentPrice() internal view returns (uint256 price) {
    (, int256 latestPrice, , , ) = priceProvider.latestRoundData();
    price = uint256(latestPrice);
}
```

Impact

Medium

Risk Breakdown

Using stale prices could result in the strategy using incorrect information to make assumptions about current profit or loss

Recommendation

Perform a check for stale oracle prices. Suggest to the Hegic team to do the same

```
(uint80 round, int256 answer, , uint256 time, uint80 answeredRound) =
pp.latestRoundData();
require(answeredRound >= round, "Stale price: round");
require(time != 0, , "Stale price: time");
return uint256(answer);
```

9. Inconsistent assumptions

The assumption is that mainnet strats aren't vulnerable to sandwich attacks because they use private relays, but a couple manual functions in Joint.sol have a require statement with a revert message of "!sandwidched", indicating a sandwich attack could happen. This revert message is at odds with the sandwich attack mitigation assumption.

Impact

Informational

10. Dead code can be removed [NEED VERIFICATION]

The `prepareReturn` function of ProviderStrategy.sol has code to check for `_totalDebt > totalAssets`. This will not happen because the final part of `closePositionReturnFunds` in Joint.sol, which is called right before to `prepareReturn` function, has a similar check for losses.

The possibly unnecessary ProviderStrategy.sol code:
https://github.com/fp-crypto/joint-strategy/blob/3c12bbff1f3af66599571e9f79d642ce62020d85/contracts/ProviderStrategy.sol#L90-L100

The code performing the same check in Joint.sol:
https://github.com/fp-crypto/joint-strategy/blob/3c12bbff1f3af66599571e9f79d642ce62020d85/contracts/Joint.sol#L236-L251

Impact

Gas saving (if verified)

11. Unnecessary sandwich warnings and mitigation [NEED VERIFICATION]

Uniswap's `removeLiquidity` is not an attack vector:

```
// **WARNING**: This call is sandwichable, care should be taken
//              to always execute with a private relay
IUniswapV2Router02(router).removeLiquidity(
    tokenA,
    tokenB,
    balanceOfPair(),
    0,
    0,
    address(this),
    now
);
```

This code is called from this function which unnecessarily prevents price movements:

```
function liquidatePositionManually(
    uint256 expectedBalanceA,
    uint256 expectedBalanceB
) external onlyVaultManagers {
    (uint256 balanceA, uint256 balanceB) = _closePosition();
    require(expectedBalanceA <= balanceA, "!sandwidched");
    require(expectedBalanceB <= balanceB, "!sandwidched");
}
```

Additionally,

This is because if the attacker moves the price away from the "true" market price, they will create an arbitrage opportunity which you will in part be realizing by withdrawing your funds. E.g. you will "cash out" at an "unfair" price on one of the assets.

Take the following scenario:

```
starting reserves:
x = 10
y = 10
```

```
k = 100

joint starts withdraw tx
mev_bot moves price to 4:1
reserves are now:

x = 20
y = 5
k = 100

joint receives:
20x
5y

(20 + 5) > (10 + 10)  // assuming a true price of 1:1
```

Impact

Informational

12. Save gas by using local variable

`balanceOfPair()` is called twice in the `_closePosition()` function of Joint.sol when the result could be cached locally and re-used:

```
if (balanceOfPair() == 0) {
    return (0, 0);
}

// **WARNING**: This call is sandwichable, care should be taken
//              to always execute with a private relay
IUniswapV2Router02(router).removeLiquidity(
    tokenA,
    tokenB,
    balanceOfPair(),
```

```
        0,
        0,
        address(this),
        now
    );
```

The same issue is found in the `openPosition()` function of Joint.sol:

```
function openPosition() external onlyProviders {
    // No capital, nothing to do
    if (balanceOfA() == 0 || balanceOfB() == 0) {
        return;
    }

    require(
        balanceOfStake() == 0 &&
            balanceOfPair() == 0 &&
            investedA == 0 &&
            investedB == 0
    ); // don't create LP if we are already invested

    (uint256 amountA, uint256 amountB, ) = createLP();
    (uint256 costHedgeA, uint256 costHedgeB) = hedgeLP(); //@note not
implemented in this contract

    investedA = amountA.add(costHedgeA);
    investedB = amountB.add(costHedgeB);

    depositLP();

    if (balanceOfStake() != 0 || balanceOfPair() != 0) {
        _returnLooseToProviders();
    }
}
```

Impact

Informational

13. addLiquidity, swapExactTokensForTokens can be sandwiched via uncle-bandit

Due to the amount of capital that yearn strategies can amass, it should be assumed that they will be the target of uncle-bandit attacks. These can be executed entirely opportunistically by searchers and any uniswap trade should be assumed to be already under the watchful eye of MEV extractors.

This means that if the flashbots block gets orphaned, there will be an opportunity for searchers to use the underlying swap as a part of their bundle. Some napkin math shows that ~6% of mainnet ethereum blocks are orphaned, resulting in a non-trivial liklihood that this poistion closure gets sandwiched regardless of private channel usage.

- https://etherscan.io/chart/uncles

Impact

Medium

Mitigation

Require keepers to pass in expected output amounts, these amounts could be computed using either:

- `call`ing the functions and checking the output prior to `send`ing
- exposing `external view` methods to compute the expected output prior to `send`ing

14. estimatedTotalAssetsAfterBalance doesn't include value of held options

The `SushiJoint` strategy holds the following assets:

- Token A
- Token B
- LP (A<>B)
- Claim on LP (Staked in MasterChef)
- Hegic Options

Assuming `getHedgeProfit` does not return the underlying value of the call options, a fact I failed to validate without the Hegic contracts, then it would stand to reason that the strategy would underreport its asset value or NAV.

Impact

TBD

Mitigation

It might be preferable to value the options held by the strategy by calling out to Hegic for the current market price of the options if they were to be liquidated by the strategy.

15. Reference address of all external contracts used.

This is to mitigate any potential of calling any wrong contracts.

Impact

Misc.