Pashov Audit Group

Stake DAO Security Review

July 21st 2025 - July 24th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Stake DAO

Stake DAO is a platform leveraging veTokenomics and Liquid Lockers to maximize governance utility, yield generation, and liquidity across multiple blockchain protocols. It integrates products like sdTokens, boosted strategies, and an on-chain Votemarket to optimize token locking, voting power, and cross-chain reward incentives.

5. Executive Summary

A time-boxed security review of the **stake-dao/contracts-monorepo** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Stake DAO**. A total of **21** issues were uncovered.

Protocol Summary	
Project Name	Stake DAO
Protocol Type	Yield optimizer
Timeline	July 21st 2025 - July 24th 2025

Review commit hash:
- 2fc661343b77cc0c153c58a2ca75578395c66907 (stake-dao/contracts-monorepo)

Fixes review commit hash:
- 07874f72b8142090b261438617be93d671a99994 (stake-dao/contracts-monorepo)

Scope

`Accountant.sol` `RewardVult.sol` `CurveLendingMarketFactory.sol` `CurveCryptoswapOracle.sol` `CurveStableswapOracle.sol` `MorphoMarketFactory.sol` `StrategyWrapper.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
Critical	2
Medium	8
Low	11
Total findings	**21**

Summary of findings

ID	Title	Severity	Status
[C-01]	Missing update of extra reward per token during deposit	Critical	Resolved
[C-02]	Checkpoints are almost always outdated due to missing `_update` override	Critical	Acknowledged
[M-01]	`StrategyWrapper.depositAssets()` reverts due to missing approval	Medium	Resolved
[M-02]	`IERC20Metadata` interface is not compatible with `USDT`	Medium	Resolved
[M-03]	Initial position's liquidity so close to liquidation	Medium	Resolved
[M-04]	Deploy functions is vulnerable to DoS attack vectors	Medium	Resolved
[M-05]	Unclaimed rewards are lost if user deposits before claiming rewards	Medium	Resolved
[M-06]	Market creation fails due to wrong pool address in `oracle` deployment	Medium	Resolved
[M-07]	Lp price calculation is not correct for oracles	Medium	Acknowledged
[M-08]	Oracles are vulnerable to flash loan attack vectors	Medium	Acknowledged
[L-01]	`getConversionPath()` does not handle the zero address feed	Low	Resolved
[L-02]	Lack of sequencer availability check for Layer 2 networks	Low	Acknowledged

ID	Title	Severity	Status
[L-03]	`name()` function can revert for some Curve pools	Low	**Resolved**
[L-04]	`CurveLendingMarketFactory` contract size exceeds mainnet limit	Low	**Resolved**
[L-05]	Lack of protocol ID check in Curve lending market deployment	Low	**Resolved**
[L-06]	Morpho market creation might require excessive funds	Low	**Resolved**
[L-07]	Morpho market creation might fail for some IRMs	Low	**Resolved**
[L-08]	Small rewards stuck	Low	Acknowledged
[L-09]	Lack of heartbeat configurability makes oracles brittle to Chainlink changes	Low	Acknowledged
[L-10]	Oracle price manipulation causes incorrect collateral and utilization	Low	Acknowledged
[L-11]	`CurveStableswapOracle.price()` can overestimate the price of the collateral token	Low	**Resolved**

Critical findings

[C-01] Missing update of extra reward per token during deposit

Severity

Impact: High

Likelihood: High

Description

`StrategyWrapper._deposit()` updates the reward per token paid for extra rewards at the time of deposit.

```
        // 3. Keep track of the Extra reward tokens checkpoints at deposit time
        address[] memory rewardTokens = REWARD_VAULT.getRewardTokens();
        for (uint256 i; i < rewardTokens.length; i++) {
    @>      checkpoint.rewardPerTokenPaid[rewardTokens[i]] =
extraRewardPerToken[rewardTokens[i]];
        }
```

However, the extra reward per token is not updated during the deposit, so the value of `extraRewardPerToken` corresponds to the last time extra rewards were claimed.

As a result, extra rewards accumulated since the last claim will be unfairly distributed to new depositors, while the existing depositors will not receive their fair share of the additional rewards.

Proof of concept

Add the following code to the file `StrategyWrapper.t.sol` and run `forge test --mt test_audit_extraRewardsUnfairDistribution`.

```
function test_audit_extraRewardsUnfairDistribution() public {
    // Setup
    address firstUser = makeAddr("firstUser");
    address secondUser = makeAddr("secondUser");

    (RewardVault[] memory vaults,) = deployRewardVaults();
    RewardVault rewardVault = vaults[0];

    wrapper = new RestrictedStrategyWrapper(rewardVault, address(new MorphoMock()),
address(this));

    vm.mockCall(
        address(protocolController),
        abi.encodeWithSelector(IProtocolController.isRegistrar.selector, address(this)),
        abi.encode(true)
    );
```

```
    rewardVault.addRewardToken(Common.WETH, address(this));

    // 1. First user deposits to the vault
    _depositIntoWrapper(rewardVault, firstUser, 1e18);

    // 2. Deposit some extra reward token into the reward vault
    uint128 extraRewardAmount = 1e20;
    deal(Common.WETH, address(this), extraRewardAmount);
    IERC20(Common.WETH).approve(address(rewardVault), extraRewardAmount);
    rewardVault.depositRewards(Common.WETH, extraRewardAmount);

    // 3. Second user deposits to the vault after 1 week
    skip(7 days);
    _depositIntoWrapper(rewardVault, secondUser, 100_000e18);

    // 4. Second user withdraws immediately after depositing
    _withdrawFromWrapper(secondUser, 100_000e18);

    // 5. First user claims extra rewards
    _claimExtraRewardsFromWrapper(firstUser);

    // The second user received most of the extra rewards, while they should have been
    // all distributed to the first user
    assertApproxEqRel(
        IERC20(Common.WETH).balanceOf(firstUser),
        extraRewardAmount * 1 / 100_001,
        1e15 // 0.1% (rounding)
    );
    assertApproxEqRel(
        IERC20(Common.WETH).balanceOf(secondUser),
        extraRewardAmount * 100_000 / 100_001,
        1e15 // 0.1% (rounding)
    );
}
```

Recommendations

```
        // 3. Keep track of the Extra reward tokens checkpoints at deposit time
        address[] memory rewardTokens = REWARD_VAULT.getRewardTokens();
+       _updateExtraRewardState(rewardTokens);
        for (uint256 i; i < rewardTokens.length; i++) {
            checkpoint.rewardPerTokenPaid[rewardTokens[i]] =
extraRewardPerToken[rewardTokens[i]];
        }
```

[C-02] Checkpoints are almost always outdated due to missing `_update` override

Severity

Impact: High

Likelihood: High

Description

Strategy tokens are used as collateral token in Morpho Blue markets. It's crucial to update checkpoints in case of transfers because checkpoints are always handled with `msg.sender`. Currently, `_update` function is not overridden in strategy wrapper, which means checkpoints will be outdated almost always.

There are many impacts of this situation:

1. Liquidator can't redeem LP tokens because he doesn't have any checkpoints for those tokens. It means it has no value for liquidators.

```
        UserCheckpoint storage checkpoint = userCheckpoints[msg.sender];
        checkpoint.balance -= amount; // revert here due to underflow
```

1. Even if the borrower is liquidated in the market, he can claim rewards because his checkpoint still exists. Checkpoints are updated only while `deposit` and `withdraw` calls are made.

2. It breaks the fungibility feature of ERC20. It behaves like non-fungible tokens.

Recommendations

Override `_update` function and update checkpoints in there.

Medium findings

[M-01] `StrategyWrapper.depositAssets()` reverts due to missing approval

Severity

Impact: Low

Likelihood: High

Description

`StrategyWrapper.depositAssets()` deposits the underlying LP tokens received by the user into the reward vault and wraps the received shares.

However, before the deposit, the function does not approve the reward vault to transfer the underlying LP tokens from the wrapper contract, causing the deposit to revert.

As a result, the `depositAssets()` function cannot be used, and users are forced to deposit the underlying LP tokens directly into the reward vault and call the `depositShares()` function instead.

Recommendations

```
        SafeERC20.safeTransferFrom(IERC20(REWARD_VAULT.asset()), msg.sender, address(this),
amount);
+       SafeERC20.forceApprove(IERC20(REWARD_VAULT.asset()), address(REWARD_VAULT), amount);
        uint256 shares = REWARD_VAULT.deposit(amount, address(this), address(this));
```

[M-02] `IERC20Metadata` interface is not compatible with `USDT`

Severity

Impact: Medium

Likelihood: Medium

Description

`MorphoMarketFactory.create()` approves the Morpho protocol to transfer the loan token. However, the `IERC20Metadata` interface is not compatible with `USDT` and other non-compliant ERC20 tokens that do not return a value on `approve()`, making the transaction revert.

```
        // Feed the market with exactly 1 token so that there is available liquidity for the
subsequent borrow.
        uint256 loanToSupply = 10 ** loan.decimals();
        loan.safeTransferFrom(msg.sender, address(this), loanToSupply);
    @>  loan.approve(address(MORPHO_BLUE), loanToSupply);
```

As a result, `USDT` and similar tokens cannot be used as a loan token in the Morpho market.

Recommendations

```
-   loan.approve(address(MORPHO_BLUE), loanToSupply);
+   loan.forceApprove(address(MORPHO_BLUE), loanToSupply);
```

[M-03] Initial position's liquidity so close to liquidation

Severity

Impact: Medium

Likelihood: Medium

Description

Currently, after deployment and creation Morpho Blue market system directly opens a position. But this position is so close to liquidation, and it can be liquidated immediately.

```
        uint256 collateralToSupply = Math.mulDiv(
            Math.mulDiv(borrowAmount, 10 ** oracle.ORACLE_BASE_EXPONENT(), oracle.price(),
Math.Rounding.Ceil),
            1e18,
            lltv,
            Math.Rounding.Ceil
        );
        uint256 buffer = 2 * (10 ** (collateral.decimals() - loan.decimals()));
        collateralToSupply += buffer;
```

In here, only 2 wei of additional collateral is added. Most probably, it can be liquidated in the next block because of interest accrual.

Recommendations

As all the lending protocols did, create a position at LTV point not LLTV point. (We don't have LTV in Morpho). If 90% is LLTV level, 85% can be a good LTV level for the initial position.

[M-04] Deploy functions is vulnerable to DoS attack vectors

Severity

Impact: Medium

Likelihood: Medium

Description

While creating the morpho market, there are several checks for market parameters in morpho blue. If the market is already created with the same parameters, it will fail in the required line.

```
        require(isIrmEnabled[marketParams.irm], ErrorsLib.IRM_NOT_ENABLED);
        require(isLltvEnabled[marketParams.lltv], ErrorsLib.LLTV_NOT_ENABLED);
@>      require(market[id].lastUpdate == 0, ErrorsLib.MARKET_ALREADY_CREATED);

        // Safe "unchecked" cast.
        market[id].lastUpdate = uint128(block.timestamp);
```

Malicious actor can simulate the transaction and frontrun the market parameters with createMarket function. In here, oracle address should be determined by attacker in order to execute the attack, but it's already very easy because the address of oracle depends on the contract's nonce and address.

Recommendations

Check whether the market has already been created or not.

[M-05] Unclaimed rewards are lost if user deposits before claiming rewards

Severity

Impact: Medium

Likelihood: Medium

Description

In the `StrategyWrapper` contract, when a user deposits, the rewardVault shares are transferred to the wrapper, and the `_deposit()` function is invoked to checkpoint the user's balance and `rewardPerTokenPaid`:

```
function _deposit(uint256 amount) internal {
        // 1. Update the internal user checkpoint
        UserCheckpoint storage checkpoint = userCheckpoints[msg.sender];
        checkpoint.balance += amount;
```

```
        // 2. Keep track of the Main reward token checkpoint
        checkpoint.rewardPerTokenPaid[MAIN_REWARD_TOKEN_SLOT] = _getGlobalIntegral();

        // 3. Keep track of the Extra reward tokens checkpoints at deposit time
        address[] memory rewardTokens = REWARD_VAULT.getRewardTokens();
        for (uint256 i; i < rewardTokens.length; i++) {
            checkpoint.rewardPerTokenPaid[rewardTokens[i]] =
extraRewardPerToken[rewardTokens[i]];
        }

        // 4. Mint wrapped tokens (1:1) for the caller
        _mint(msg.sender, amount);

        emit Deposited(msg.sender, amount);
    }
```

However, if the user had unclaimed rewards or extra rewards prior to this deposit, those rewards are **lost** because `rewardPerTokenPaid` is **overwritten** with the current global value, preventing the user from claiming rewards accumulated prior to the new deposit.

Recommendations

Ensure that users **claim their pending rewards** before making new deposits.

[M-06] Market creation fails due to wrong pool address in `oracle` deployment

Severity

Impact: Medium

Likelihood: Medium

Description

In the `CurveLendingMarketFactory` contract `deploy()` functions, the `_curvePool` parameter passed during the deployment of the oracle (`CurveCryptoswapOracle` or `CurveStableswapOracle`) is incorrectly set to `rewardVault.asset()` . However, `_curvePool` is expected to be the address of the actual Curve pool used for fetching the LP token price.

Using the `rewardVault.asset()` address instead of the proper Curve pool address results in failed oracle price fetching. Consequently, any attempt to create a market will fail due to failure when trying to fetch the price (`CURVE_POOL.price()` will revert).

```
function deploy(
        IRewardVault rewardVault,
        StableswapOracleParams calldata oracleParams,
        MarketParams calldata marketParams,
```

```
        ILendingFactory lendingFactory
    ) external onlyOwner returns (IStrategyWrapper, IOracle, bytes memory) {
        //...

        // 2. Deploy the oracle
        IOracle oracle = new CurveStableswapOracle(
            rewardVault.asset(),
            address(collateral),
            oracleParams.loanAsset,
            oracleParams.loanAssetFeed,
            oracleParams.loanAssetFeedHeartbeat,
            oracleParams.baseFeed,
            oracleParams.baseFeedHeartbeat
        );
        //...
    }
```

```
onstructor(
        address _curvePool,
        address _collateralToken,
        address _loanAsset,
        address _loanAssetFeed,
        uint256 _loanAssetHeartbeat,
        address[] memory _token0ToUsdFeeds,
        uint256[] memory _token0ToUsdHeartbeats
    ) {
        //...
        CURVE_POOL = ICurveCryptoSwapPool(_curvePool);
        //...
    }
```

Recommendations

Update the `CurveLendingMarketFactory.deploy()` functions to pass the correct Curve pool address to the oracle constructor instead of `rewardVault.asset()`.

[M-07] Lp price calculation is not correct for oracles

Severity

Impact: Medium

Likelihood: Medium

Description

In both crpto and stable swap oracles, it firstly gets the estimated price of the LP price, but it's not accurate enough to calculate the price of LP token. Price of LP token is estimated with additional **ramp** logic inside of the curve pool contract, and it makes the calculation inaccurate.

In order to determine the price of an asset, we should check how many other assets we can withdraw while redeeming LP token. For instance, if redeeming 1e18 LP token returns me 1e18 wETH and 1e18 rETH, it means LP price is equal to `1 + 1 * rETH_exchange_rate`. However, in Curve pool there are additional `ramp` logics that affects the price incorrectly.

get_virtual_price function

```
@view
@external
def get_virtual_price() -> uint256:
    """
    @notice The current virtual price of the pool LP token
    @dev Useful for calculating profits
    @return LP token virtual price normalized to 1e18
    """
    D: uint256 = self.get_D(self._xp(self._stored_rates()), self._A())
    # D is in the units similar to DAI (e.g. converted to precision 1e18)
    # When balanced, D = n * x_u - total virtual value of the portfolio
    token_supply: uint256 = ERC20(self.lp_token).totalSupply()
    return D * PRECISION / token_supply
```

However, when we check the removeLiquidity function, logic is much simpler than this. We just need to compare the balances with total LP supply in order to get the accurate price.

> Note same problem exist for both stable swap and crypto swap

```
@external
@nonreentrant('lock')
def remove_liquidity(_amount: uint256, min_amounts: uint256[N_COINS]) -> uint256[N_COINS]:
    """
    @notice Withdraw coins from the pool
    @dev Withdrawal amounts are based on current deposit ratios
    @param _amount Quantity of LP tokens to burn in the withdrawal
    @param min_amounts Minimum amounts of underlying coins to receive
    @return List of amounts of coins that were withdrawn
    """
    _lp_token: address = self.lp_token
    total_supply: uint256 = ERC20(_lp_token).totalSupply()
    amounts: uint256[N_COINS] = empty(uint256[N_COINS])

    for i in range(N_COINS):
        _balance: uint256 = self.balances[i]
        value: uint256 = _balance * _amount / total_supply
        assert value >= min_amounts[i], "Withdrawal resulted in fewer coins than expected"
        self.balances[i] = _balance - value
        amounts[i] = value
        if i == 0:
            raw_call(msg.sender, b"", value=value)
        else:
            assert ERC20(self.coins[1]).transfer(msg.sender, value)

    CurveToken(_lp_token).burnFrom(msg.sender, _amount)  # Will raise if not enough
```

I have also checked the price difference of rETH stable pool, and it's very different from the actual price (usually lower than the accurate price).

Correct price formula for rETH stable swap pool is equal to:

```
eth_bal   = pool.balances[0]
reth_bal  = pool.balances[1]
rate_rETH = rETH_contract.exchangeRate()

# convert both to ETH-units:
total_ETH_value = eth_bal + reth_bal * rate_rETH / 1e18

# get LP supply:
lp_supply = ERC20(lp_token).totalSupply()

# price per LP in ETH:
price = total_ETH_value / lp_supply
```

This price mismatch will cause incorrect liquidations and incorrect calculations in Morpho Blue market. This is why it's crucial to calculate it accurately.

Recommendations

Calculate the price of LP token by using balances and token supply of LP token instead of direct fetch as `get_virtual_price` / `lp_price` .

[M-08] Oracles are vulnerable to flash loan attack vectors

Severity

Impact: High

Likelihood: Law

Description

Both of the oracles directly fetch the current price of LP tokens in the pool. However, this is not safe because price can be manipulated for a very short time-range by multiple attack vectors. One of the biggest threats is flashloan attack vectors. When the pool is imbalanced in terms of liquidity, oracles will return an incorrect value for LP price.

The root cause of this situation is the direct usage of Pool state in oracle price calculation. Implementation like TWAP is crucial in this kind of scenarios. Morpho Blue will use this vulnerable oracle, and attacker can liquidate every position by manipulating the price of LP token.

```
        uint256 priceLpInPeg = CURVE_POOL.get_virtual_price(); // @audit on-chain price without
TWAP
```

Recommendations

Implement a TWAP logic in order to eliminate imbalanced curve pool state scenarios. Time averaged price will be more accurate and smooth than the current price calculation.

Low findings

[L-01] `getConversionPath()` does not handle the zero address feed

`CurveCryptoswapOracle.getConversionPath()` returns a string with the conversion path for the price calculation.

```
function getConversionPath() external view returns (string memory path) {
    uint256 length = token0ToUsdFeeds.length;
    for (uint256 i; i < length; i++) {
@>      path = string.concat(path, token0ToUsdFeeds[i].description(), " -> ");
    }
    path = string.concat(path, " -> ", LOAN_ASSET_FEED.description());
}
```

However, it is not taken into account that `token0ToUsdFeeds` can contain the zero address to signify an "identity hop" in the conversion path. If that is the case, the transaction will revert on the call to `description()` for the zero address.

It is recommended to either handle the case for the zero address in the `getConversionPath()` function or not to allow setting the zero address in the first place, as it does not have any practical use.

[L-02] Lack of sequencer availability check for Layer 2 networks

Chainlink's price feeds in layer 2 networks are updated through the sequencer, which can become unavailable. The Chainlink documentation recommends integrating a Sequencer Uptime Data Feed, which continuously monitors and records the last known status of the sequencer.

The implementation of `_fetchFeedPrice()` for `CurveStableswapOracle` and `CurveCryptoswapOracle` lacks this validation, which can result in the protocol using an outdated price.

Consider creating wrapper contracts to be used in layer 2 networks that override the `_fetchFeedPrice()` function to include a check for the sequencer's availability.

[L-03] `name()` function can revert for some Curve pools

Some Curve pools are not compatible with the `ICurvePool` interface, as they do not have a `name()` and `decimals()` function.

For such pools, the `name()` function in the oracle contract will revert.

```
    function name() external view returns (string memory _name) {
        _name = string.concat(CURVE_POOL.name(), "-", LOAN_ASSET.symbol(), " Stable Oracle");
    }
```

Consider adding a fallback mechanism to handle such cases.

```
+   string private immutable POOL_NAME;

    constructor(
        address _curvePool,
        address _collateralToken,
        address _loanAsset,
        address _loanAssetFeed,
        uint256 _loanAssetFeedHeartbeat,
        address _baseFeed,
-       uint256 _baseFeedHeartbeat
+       uint256 _baseFeedHeartbeat,
+   string memory _poolName
    ) {
+       try CURVE_POOL.name() returns (string memory poolName) {
+           POOL_NAME = poolName;
+       } catch {
+           require(bytes(_poolName).length > 0, "EmptyPoolName()");
+           POOL_NAME = _poolName;
+       }

(...)

    function name() external view returns (string memory _name) {
-       _name = string.concat(CURVE_POOL.name(), "-", LOAN_ASSET.symbol(), " Stable Oracle");
+       _name = string.concat(POOL_NAME, "-", LOAN_ASSET.symbol(), " Stable Oracle");
    }
```

[L-04] `CurveLendingMarketFactory` contract size exceeds mainnet limit

The `CurveLendingMarketFactory` contract exceeds the size limit for mainnet deployment. Even when setting the optimizer runs to 1, the runtime size is 24,988 bytes, while the limit is 24,576 bytes.

```
forge build --sizes
Runtime Size (B) | Initcode Size (B) | Runtime Margin (B) | Initcode Margin (B) |
24,988           | 25,327            | -412               | 23,825              |

(...)

Error: some contracts exceed the runtime size limit (EIP-170: 24576 bytes)
```

[L-05] Lack of protocol ID check in Curve lending market deployment

In `CurveLendingMarketFactory.sol`, when creating a new market on Curve using a cryptoswap oracle, it is checked that the protocol ID of the reward vault matches the expected ID for Curve.

```
    function deploy(
        IRewardVault rewardVault,
        CryptoswapOracleParams calldata oracleParams,
        MarketParams calldata marketParams,
        ILendingFactory lendingFactory
    ) external onlyOwner returns (IStrategyWrapper, IOracle, bytes memory) {
        require(PROTOCOL_CONTROLLER.vaults(rewardVault.gauge()) == address(rewardVault),
InvalidRewardVault());
@>      require(rewardVault.PROTOCOL_ID() == bytes4(keccak256("CURVE")), InvalidProtocolId());
```

However, on the creation of a new market on Curve using a stableswap oracle, this check is not performed.

Consider adding the same check in the `deploy()` version for stableswap oracles for consistency and to prevent user mistakes on deployment.

[L-06] Morpho market creation might require excessive funds

`MorphoMarketFactory.create()` prevents zero utilization rate decay at market creation. This is done by supplying 1 token and borrowing for 90% of its value, following the guidance in the Morpho documentation.

However, instead of supplying $1 worth of loan tokens, as recommended, 1 loan token is supplied. This could result in a very high amount of funds being required to create the market, depending on the price of the loan token. For example, if the loan token is WBTC, at current prices, it would be required to supply 1 WBTC valued at $118,000 plus collateral tokens worth 90% of the supplied WBTC, which would be $106,200 worth of collateral tokens. Requiring a total $224,200 worth of funds to create a market.

Consider receiving the amount of loan tokens to supply as a parameter in the `create()` function, so that the market can be created with a more reasonable amount of funds.

[L-07] Morpho market creation might fail for some IRMs

`MorphoMarketFactory.create()` prevents zero utilization rate decay at market creation. This is done by supplying 1 token and borrowing for 90% of its value, following the guidance in the Morpho documentation.

However, it is important to note that this issue and its solution are specific to the Adaptive Curve IRM. While currently this is the only IRM enabled in Morpho, others may be added in the future. In the best case, the mitigation strategy would be performed without the need, and in the worst case, the deployment would fail due to restrictions in the IRM implementation, such as not allowing borrowing up to 90% of the deposited tokens.

It is recommended to either document this limitation in the contract and the official documentation or to apply the mitigation strategy only when the Adaptive Curve IRM is used.

[L-08] Small rewards stuck

In the `StrategyWrapper` contract, the `_updateExtraRewardState()` function is called before users claim their extra rewards via `claimExtraRewards()`, where it first pulls any pending extra rewards from the `RewardVault`, then updates the `extraRewardPerToken` value for each extra reward token as follows:

```
function _updateExtraRewardState(address[] memory tokens) internal {
        //...
        uint256[] memory amounts = REWARD_VAULT.claim(tokens, address(this));
        for (uint256 i; i < tokens.length; i++) {
            uint256 amount = amounts[i];
            if (amount > 0) extraRewardPerToken[tokens[i]] += Math.mulDiv(amount, 1e18, supply);
        }
    }
```

However, if the newly claimed amount is too small relative to the `totalSupply`, this calculation may round down to zero, effectively discarding the value and preventing any updates to `extraRewardPerToken`, as a result, the small amount of extra reward tokens will be stuck in the contract and unclaimable by depositors.

Recommendation:
Consider storing and accumulating the unaccounted reward amount per token to be added later when it becomes large enough to affect `extraRewardPerToken`.

[L-09] Lack of heartbeat configurability makes oracles brittle to Chainlink changes

In both `CurveStableswapOracle` and `CurveCryptoswapOracle` contracts, the Chainlink `heartbeat` values are set as **immutable** at the time of deployment. If Chainlink changes the heartbeat configuration of a price feed (e.g., increases the duration between valid updates), the oracles in these contracts will continue using the old threshold.

This will cause the `price()` function to **revert** due to failed staleness checks, even though the Chainlink feed is behaving as expected under its new configuration. While this issue seems to be **acknowledged** as per the documentation, Morpho markets **do not support updating the oracle** after deployment, so any market relying on these oracles will become **stalled or corrupted**, impacting core functionalities like borrowing and liquidations.

Recommendation:
Consider storing heartbeat values in a mutable storage variable that can be updated by a governance or admin.

[L-10] Oracle price manipulation causes incorrect collateral and utilization

In the `MorphoMarketFactory.create()` function, the amount of `collateralToSupply` is calculated based on the current `lp_price()` from either the `CurveStableswapOracle` or `CurveCryptoswapOracle`. However, both oracle's documentation **acknowledge** that the LP token price can be **manipulated within a single block** due to flash deposits/withdrawals.

Since `collateralToSupply` is pulled from the deployer based on this price, two risks arise:

- If the price is artificially low, more collateral than necessary will be pulled from the deployer.
- If the price is artificially high, less collateral than required will be pulled, and the expected utilization ratio (e.g., 90%) will not be achieved accurately.

After the block settles, and if the LP price drops back to its real value, the position may appear under-collateralized, potentially triggering liquidation in the next block.

```
 function create(IStrategyWrapper collateral, IERC20Metadata loan, IOracle oracle, address irm,
uint256 lltv)
        external
        onlyDelegateCall
        returns (Id id)
    {
        //...
        uint256 collateralToSupply = Math.mulDiv(
            Math.mulDiv(borrowAmount, 10 ** oracle.ORACLE_BASE_EXPONENT(), oracle.price(),
Math.Rounding.Ceil),
            1e18,
            lltv,
            Math.Rounding.Ceil
        );

        //...
        vault.safeTransferFrom(msg.sender, address(this), collateralToSupply);
        //...
    }
```

Recommendation:
Consider adding slippage tolerance validation against recent pricing before finalizing market creation.

[L-11] `CurveStableswapOracle.price()` can overestimate the price of the collateral token

The `CurveStableswapOracle.price()` function returns the price of 1 collateral token in terms of the loan token. The first steps of the calculation are:

1. Get the price of the LP token in its "unit of account". For example, for the `cbBTC/wBTC` pool, this will be the price of the LP token in BTC terms.
2. Get the price of the "unit of account" in USD from the base feed. Following the example, this will be the price of BTC in USD.

```
194:     function price() external view returns (uint256) {
195:         // 1. Get the price of the LP token in its "unit of account" (e.g., USD for USDC/
crvUSD or ETH for wstETH/wETH).
196:         // This value always has 18 decimals.
197:         uint256 priceLpInPeg = CURVE_POOL.get_virtual_price();
198:
199:         // 2. Get the price of the "unit of account" in USD from the base feed.
200:         uint256 pricePegInUsd;
201:         if (address(BASE_FEED) != address(0)) {
202:             pricePegInUsd = _fetchFeedPrice(BASE_FEED, BASE_FEED_HEARTBEAT);
```

So, `priceLpInPeg * pricePegInUsd` will give the price of the LP token in USD (scaled by 18 + `BASE_FEED_DECIMALS`).

In the first step, `get_virtual_price` returns the total liquidity in the pool divided by the total supply of LP tokens, but the calculation of the total liquidity assumes that all tokens in the pool are priced equally; however, **this is not necessarily the case**.

In the example of the `cbBTC/wBTC` pool, the price of `BTC`, `cbBTC`, and `wBTC` can differ significantly, especially during periods of high volatility. We can eliminate the divergence with respect to the underlying asset and use as a unit of account one of the assets in the pool, for example, `cbBTC`, but we have yet to assume that the price of `cbBTC` is equal to the price of `wBTC`, which is not necessarily true.

This can lead to the oracle returning a price that overestimates the value of the LP token, potentially leading to the undercollateralization of loans in the lending market.

Recommendations

This article provides a good overview of the issue and how to mitigate it.

The recommended approach is querying the price of each asset in the pool in USD terms and using the minimum price for the `pricePegInUsd` variable. This ensures that the collateral token is always priced conservatively.