2021-09-bvecvx (BadgerDAO)

Benchmark Security Review Performed by Bug Hunter

16.05.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

This security-review report was generated with Truscova's Bug Hunter tool to benchmark its performance on selected Yield Farming projects. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2021-09-bvecvx was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2021-09-bvecvx
  Commit hash: `1d64bd58c7a4224cc330cef283561e90ae6a3cf5`

Bug Hunter reviewed the following files:

veCVXStrategy.sol CvxLocker.sol CvxStakingProxy.sol StrategyCvxHelper.sol BaseStrategy.sol SettV3.sol Controller.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	1
Medium	2
Low	11

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
16.05.2025	Project submitted
16.05.2025	Security review concluded

Detailed Findings

1. Lack of Input Sanitization in `initialize` Function Parameters

Severity	Low
Finding ID	BH-2021-09-bvecvx-01
Target	veCVX/contracts/deps/Controller.sol
Function name	initialize

Description

The `initialize` function sets several critical contract roles (`governance`, `strategist`, `keeper`, `rewards`) using externally provided addresses, but does not validate them. If any of these addresses are zero or incorrectly set, it can lead to loss of functionality or introduce risks such as ungovernable or misconfigured contract behavior.

```
function initialize(
        address _governance,
        address _strategist,
        address _keeper,
        address _rewards
    ) public initializer {
        governance = _governance;
        strategist = _strategist;
        keeper = _keeper;

        rewards = _rewards;
        onesplit = address(0x50FDA034C0Ce7a8f7EFDAebDA7Aa7cA21CC1267e);
    }
```

Recommendation

Add input sanitization to check that all passed addresses are valid (i.e., not the zero address). Use `require(addr != address(0), "invalid address")` for each parameter to ensure proper initialization.

2. Inaccurate CVX Lock Amount Derived from Ratio Difference

Severity	High
Finding ID	BH-2021-09-bvecvx-02
Target	veCVX/contracts/veCVXStrategy.sol
Function name	manualRebalance

Description

The function calculates the CVX amount to lock using `cvxToLock = newLockRatio.sub(currentLockRatio)`, where both variables are derived from a flawed comparison of ratios. Even if these ratios were correctly unit-aligned (which they are not), subtracting ratios and assuming the result maps directly to a CVX token amount introduces significant inaccuracies. The actual CVX amount needed to achieve the target lock percentage should be calculated directly based on `totalCVXBalance` and `balanceInLock`, not the difference in their relative ratios.

```
function manualRebalance(uint256 toLock) external whenNotPaused {
        _onlyGovernance();
        require(toLock <= MAX_BPS, "Max is 100%");

        if (processLocksOnRebalance) {
            LOCKER.processExpiredLocks(false);
        }

        if (harvestOnRebalance) {
            harvest();
        }

        uint256 balanceOfWant = IERC20Upgradeable(want).balanceOf(address(this));
        uint256 balanceOfCVX = IERC20Upgradeable(CVX).balanceOf(address(this));
        uint256 balanceInLock = LOCKER.balanceOf(address(this));
        uint256 totalCVXBalance =
balanceOfCVX.add(balanceInLock).add(wantToCVX(balanceOfWant));

        uint256 currentLockRatio = balanceInLock.mul(10**18).div(totalCVXBalance);
        uint256 newLockRatio = totalCVXBalance.mul(toLock).div(MAX_BPS);
        uint256 toWantRatio = totalCVXBalance.mul(MAX_BPS.sub(toLock)).div(MAX_BPS);
```

```
        if (newLockRatio <= currentLockRatio) {
            uint256 toDeposit = IERC20Upgradeable(CVX).balanceOf(address(this));
            if (toDeposit > 0) {
                CVX_VAULT.deposit(toDeposit);
            }
            return;
        }

        uint256 cvxToLock = newLockRatio.sub(currentLockRatio);

        uint256 maxCVX = IERC20Upgradeable(CVX).balanceOf(address(this));
        if (cvxToLock > maxCVX) {
            LOCKER.lock(address(this), maxCVX, LOCKER.maximumBoostPayment());
        } else {
            LOCKER.lock(address(this), cvxToLock, LOCKER.maximumBoostPayment());
        }

        uint256 cvxLeft = IERC20Upgradeable(CVX).balanceOf(address(this));
        if (cvxLeft > 0) {
            CVX_VAULT.deposit(cvxLeft);
        }
    }
```

Recommendation

Instead of using a ratio difference to compute token amounts, calculate the desired locked CVX amount explicitly and subtract the current locked balance:

```
uint256 desiredLockAmount = totalCVXBalance.mul(toLock).div(MAX_BPS);
if (balanceInLock >= desiredLockAmount) {
    // Already locked enough, do nothing or deposit remaining
} else {
    uint256 amountToLock = desiredLockAmount.sub(balanceInLock);
    // Proceed to lock only what's needed
}
```

3. Lack of Logical Consistency Check Between Minimum and Maximum Stake Limits

Severity	Low
Finding ID	BH-2021-09-bvecvx-03
Target	veCVX/contracts/locker/CvxLocker.sol
Function name	setStakeLimits

Description

The `setStakeLimits` function sets new values for `minimumStake` and `maximumStake` but only verifies that each value is less than or equal to `denominator`. It does not enforce that `_minimum <= _maximum`, which could result in an invalid configuration where the minimum stake exceeds the maximum stake. This could block valid user operations or create unintended behavior in stake validation logic elsewhere in the contract.

```
function setStakeLimits(uint256 _minimum, uint256 _maximum)
        external
        onlyOwner
    {
        require(_minimum <= denominator, "min range");
        require(_maximum <= denominator, "max range");
        minimumStake = _minimum;
        maximumStake = _maximum;
        updateStakeRatio(0);
    }
```

Recommendation

Add a sanity check to ensure that the minimum stake does not exceed the maximum:

```
require(_minimum <= _maximum, "minimum > maximum");
```

4. Block Lock Bypass via Uninitialized `blockLock` Mapping Entry

Severity	Medium
Finding ID	BH-2021-09-bvecvx-04
Target	contracts/deps/SettV3.sol
Function name	_blockLocked

Description

The `_blockLocked` function uses `blockLock[msg.sender] < block.number` to restrict repeated actions within the same block. However, since `blockLock` is a mapping and mappings in Solidity default to zero, a new address that has never been written to will pass this check by default. This allows a user to bypass the restriction unintentionally if the `blockLock` value has not been explicitly initialized elsewhere in the contract, undermining the protection mechanism.

```
function _blockLocked() internal view {
    require(blockLock[msg.sender] < block.number, "blockLocked");
}
```

Recommendation

Ensure that `blockLock[msg.sender]` is explicitly set in all state-mutating functions that rely on `_blockLocked` for protection. For example, after a successful call, assign `blockLock[msg.sender] = block.number;` to enforce the lock.

5. Incorrect Variable Usage in `setBoost` Validation Allows Bypass of Limits

Severity	Medium
Finding ID	BH-2021-09-bvecvx-05
Target	/veCVX/contracts/locker/CvxLocker.sol
Function name	setBoost

Description

In the `setBoost` function, input parameters `_max` and `_rate` are intended to be validated against hard caps (15% and 3x, respectively). However, the function incorrectly validates the existing state variables (`maximumBoostPayment` and `boostRate`) instead of the new input values (`_max` , `_rate`). This allows the contract owner to bypass the limits by supplying values exceeding the intended thresholds, as no checks are enforced on the new parameters being set.

```
function setBoost(
        uint256 _max,
        uint256 _rate,
        address _receivingAddress
    ) external onlyOwner {
        require(maximumBoostPayment < 1500, "over max payment"); //max 15%
        require(boostRate < 30000, "over max rate"); //max 3x
        require(_receivingAddress != address(0), "invalid address"); //must point
somewhere valid
        nextMaximumBoostPayment = _max;
        nextBoostRate = _rate;
        boostPayment = _receivingAddress;
    }
```

Recommendation

Update the validation logic to correctly check the new inputs `_max` and `_rate` against their respective caps:

```
require(_max < 1500, "over max payment");
require(_rate < 30000, "over max rate");
```

6. Unrestricted External Access to `setApprovals` Function

Severity	Low
Finding ID	BH-2021-09-bvecvx-06
Target	veCVX/contracts/locker/CvxLocker.sol
Function name	setApprovals

Description

The `setApprovals` function is marked `external` and lacks any access control (e.g., `onlyOwner` or `onlyGovernance`). This allows any external address to call it and repeatedly reset token approvals. While this may not have a direct security impact due to the use of safe approval patterns, it could still result in unnecessary gas costs, denial-of-service via griefing, or unexpected interaction with external staking contracts if approvals are manipulated in edge cases.

```
function setApprovals() external {
        IERC20(cvxCrv).safeApprove(cvxcrvStaking, 0);
        IERC20(cvxCrv).safeApprove(cvxcrvStaking, uint256(-1));

        IERC20(stakingToken).safeApprove(stakingProxy, 0);
        IERC20(stakingToken).safeApprove(stakingProxy, uint256(-1));
    }
```

Recommendation

Restrict access to the `setApprovals` function to trusted roles only, such as `onlyOwner` or a governance-controlled modifier, to prevent unnecessary or malicious external usage.

7. Lack of Bounds and Existence Checks in `findEpochId` May Cause Unexpected Behavior

Severity	Low
Finding ID	BH-2021-09-bvecvx-07
Target	veCVX/contracts/locker/CvxLocker.sol
Function name	findEpochId

Description

The `findEpochId` function performs a binary search on the `epochs` array to find the epoch associated with a normalized `_time`. However, it does not check whether `epochs.length` is zero before subtracting one, which could cause an underflow in `uint256 max = epochs.length - 1;`. Additionally, the function assumes that a matching epoch will always be found, but if no epoch exactly matches the given `_time`, the function returns the `min` index, which may not correspond to a valid or meaningful epoch depending on the use case.

```
function findEpochId(uint256 _time) external view returns (uint256 epoch) {
        uint256 max = epochs.length - 1;
        uint256 min = 0;

        //convert to start point
        _time = _time.div(rewardsDuration).mul(rewardsDuration);

        for (uint256 i = 0; i < 128; i++) {
            if (min >= max) break;

            uint256 mid = (min + max + 1) / 2;
            uint256 midEpochBlock = epochs[mid].date;
            if (midEpochBlock == _time) {
                //found
                return mid;
            } else if (midEpochBlock < _time) {
                min = mid;
            } else {
                max = mid - 1;
            }
        }
```

```
        return min;
    }
```

Recommendation

Add a check to ensure `epochs.length > 0` at the beginning of the function. Also, consider clearly documenting or asserting behavior when no exact match is found, or returning a sentinel value if applicable. For example:

```
require(epochs.length > 0, "no epochs");
```

8. Unbounded Loop in `updateReward` Modifier May Lead to Gas Limit Reversion

Severity	Low
Finding ID	BH-2021-09-bvecvx-08
Target	veCVX/contracts/locker/CvxLocker.sol
Function name	updateReward

Description

The `updateReward` modifier iterates over the `rewardTokens` array to update reward tracking variables for each token. If `rewardTokens` becomes excessively large, the loop may consume too much gas, potentially leading to transaction reversion due to out-of-gas errors. While this may not be a concern in typical usage, it introduces a scalability risk and may make the system fragile under administrative misuse (e.g., adding too many reward tokens).

```
modifier updateReward(address _account) {
        {
            //stack too deep
            Balances storage userBalance = balances[_account];
            uint256 boostedBal = userBalance.boosted;
            for (uint256 i = 0; i < rewardTokens.length; i++) {
                address token = rewardTokens[i];
                rewardData[token].rewardPerTokenStored = _rewardPerToken(token)
                    .to208();
                rewardData[token].lastUpdateTime = _lastTimeRewardApplicable(
                    rewardData[token]
                        .periodFinish
                )
                    .to40();
                if (_account != address(0)) {
                    //check if reward is boostable or not. use boosted or locked
balance accordingly
                    rewards[_account][token] = _earned(
                        _account,
                        token,
                        rewardData[token].useBoost
                            ? boostedBal
                            : userBalance.locked
```

```
                    );
                    userRewardPerTokenPaid[_account][token] = rewardData[token]
                        .rewardPerTokenStored;
                }
            }
        }
        _;
    }
```

Recommendation

Enforce an upper bound on the number of allowed reward tokens, or split reward updates into batched functions if dynamic expansion is needed. Additionally, consider emitting events to help monitor gas usage trends for large reward token sets.

9. Use of Deprecated `now` Keyword and Missing Slippage Protection in `_swapcvxCRVToWant`

Severity	Low
Finding ID	BH-2021-09-bvecvx-09
Target	veCVX/contracts/veCVXStrategy.sol
Function name	_swapcvxCRVToWant

Description

The `_swapcvxCRVToWant` function uses the deprecated `now` keyword as the deadline parameter in the `swapExactTokensForTokens` call. `now` is an alias for `block.timestamp` and is considered outdated in modern Solidity. Additionally, the function sets the `amountOutMin` parameter to `0`, which exposes the swap to MEV attacks or slippage exploitation. An attacker could manipulate the price between tokens, causing the swap to result in unfavorable or maliciously front-run trades.

```
function _swapcvxCRVToWant() internal {
        uint256 toSwap = IERC20Upgradeable(reward).balanceOf(address(this));

        if (toSwap == 0) {
            return;
        }

        // Sushi reward to WETH to want
        address ;
        path[0] = reward;
        path[1] = WETH;
        path[2] = CVX;
        IUniswapRouterV2(SUSHI_ROUTER).swapExactTokensForTokens(
            toSwap,
            0,
            path,
            address(this),
            now
        );

        // Deposit into vault
        uint256 toDeposit = IERC20Upgradeable(CVX).balanceOf(address(this));
```

```
        if (toDeposit > 0) {
            CVX_VAULT.deposit(toDeposit);
        }
    }
```

Recommendation

Replace `now` with `block.timestamp` for clarity and compatibility with newer Solidity versions. Set a reasonable minimum output amount (`amountOutMin`) either through slippage tolerance or via an on-chain price oracle or TWAP mechanism to protect against adverse price movements.

Example:

```
uint256 minAmountOut = getMinOut(toSwap); // calculate based on oracle or slippage
tolerance
IUniswapRouterV2(SUSHI_ROUTER).swapExactTokensForTokens(
    toSwap,
    minAmountOut,
    path,
    address(this),
    block.timestamp
);
```

10. Lack of Slippage and Return Checks in Token Swaps and External Calls in `harvest`

Severity	Low
Finding ID	BH-2021-09-bvecvx-10
Target	bCVX/StrategyCvxHelper.sol
Function name	harvest

Description

The `harvest` function performs several external operations such as withdrawing from a reward pool, swapping tokens via SushiSwap, and staking. However, it does not validate return values or enforce slippage protection during the `_swapExactTokensForTokens` call. Without setting a minimum acceptable amount (`amountOutMin`), the contract may be vulnerable to MEV or price manipulation attacks. Additionally, the function assumes all external calls (e.g., `withdraw` , `stake`) succeed without verifying their effects, which could lead to silent failures and inconsistent state.

```
function harvest() external whenNotPaused returns (uint256 cvxHarvested) {
        _onlyAuthorizedActors();
        // 1. Harvest gains from positions
        _tendGainsFromPositions();

        uint256 stakedCvxCrv = cvxCrvRewardsPool.balanceOf(address(this));
        if (stakedCvxCrv > 0) {
            cvxCrvRewardsPool.withdraw(stakedCvxCrv, true);
        }

        // 2. Swap cvxCRV tokens to CVX
        uint256 cvxCrvBalance = cvxCrvToken.balanceOf(address(this));

        if (cvxCrvBalance > 0) {
            _swapExactTokensForTokens(sushiswap, cvxCrv, cvxCrvBalance,
getTokenSwapPath(cvxCrv, cvx));
        }

        // Track harvested + converted coin balance of want
        cvxHarvested = cvxToken.balanceOf(address(this));
        _processFee(cvx, cvxHarvested, performanceFeeGovernance,
```

```
IController(controller).rewards());

        // 3. Stake all CVX
        if (cvxHarvested > 0) {
            cvxRewardsPool.stake(cvxToken.balanceOf(address(this)));
        }

        emit Harvest(cvxHarvested, block.number);
        return cvxHarvested;
    }
```

Recommendation

Enhance robustness by: 1. Adding slippage protection to the token swap function (e.g., via oracle-based or configurable `amountOutMin`). 2. Validating that external contract interactions (e.g., staking, withdrawing) succeed and return expected results. 3. Emitting more granular events or adding internal checks for critical steps.

Example:

```
_swapExactTokensForTokens(sushiswap, cvxCrv, cvxCrvBalance, minOut);
require(cvxToken.balanceOf(address(this)) >= expectedAmount, "Low swap output");
```

11. Use of `approve` Instead of `safeApprove` May Cause Compatibility or Safety Issues

Severity	Low
Finding ID	BH-2021-09-bvecvx-11
Target	bCVX/StrategyCvxHelper.sol
Function name	initialize

Description

The contract uses the native ERC-20 `approve` function to grant token allowances:

```
cvxToken.approve(address(cvxRewardsPool), MAX_UINT_256);
```

While this may work with standard ERC-20 tokens, some widely used tokens (e.g., USDT) do not return a boolean value from `approve`, which can cause silent failures or undefined behavior. Using OpenZeppelin's `SafeERC20.safeApprove` adds proper return value checking and is the industry-recommended pattern to prevent compatibility and security issues.

```
function initialize(
        address _governance,
        address _strategist,
        address _controller,
        address _keeper,
        address _guardian,
        uint256;
        path[0] = cvxCrv;
        path[1] = weth;
        path[2] = cvx;
        _setTokenSwapPath(cvxCrv, cvx, path);

        // Approvals: Staking Pool
        cvxToken.approve(address(cvxRewardsPool), MAX_UINT_256);
    }
```

Recommendation

Replace all direct `approve` calls with `safeApprove` from OpenZeppelin's `SafeERC20` library. Ensure that the contract includes the correct import and using directive:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
using SafeERC20 for IERC20;

cvxToken.safeApprove(address(cvxRewardsPool), MAX_UINT_256);
```

12. Front-Running and Incentive Gaming Risk in Public `distribute` Function

Severity	Low
Finding ID	BH-2021-09-bvecvx-12
Target	veCVX/contracts/locker/CvxStakingProxy.sol
Function name	distribute

Description

The `distribute` function is externally callable and rewards the caller with a portion of `cvxCrv` (`callIncentive`). While this design encourages public execution of reward distribution, it exposes the protocol to front-running and incentive gaming risks. Malicious actors or bots can monitor the mempool and front-run legitimate calls to repeatedly extract small incentive payouts. This behavior can lead to inefficient reward distribution, unnecessary gas usage, and reduced returns for actual stakeholders.

```
function distribute() external {
        //claim rewards
        IConvexRewards(cvxStaking).getReward(false);

        //convert any crv that was directly added
        uint256 crvBal = IERC20(crv).balanceOf(address(this));
        if (crvBal > 0) {
            ICrvDepositor(crvDeposit).deposit(crvBal, true);
        }

        //distribute cvxcrv
        uint256 cvxCrvBal = IERC20(cvxCrv).balanceOf(address(this));

        if (cvxCrvBal > 0) {
            uint256 incentiveAmount = cvxCrvBal.mul(callIncentive).div(denominator);
            cvxCrvBal = cvxCrvBal.sub(incentiveAmount);

            //send incentives
            IERC20(cvxCrv).safeTransfer(msg.sender,incentiveAmount);

            //update rewards
            ICvxLocker(rewards).notifyRewardAmount(cvxCrv, cvxCrvBal);
```

```
            emit RewardsDistributed(cvxCrv, cvxCrvBal);
        }
    }
```

Recommendation

Mitigate front-running and gaming risks by adding: 1. A cooldown mechanism (e.g., require at least X seconds between calls). 2. A minimum reward threshold to prevent dust-level incentive claims. 3. Optionally restrict access to a keeper system if public execution is not essential.

Example cooldown:

```
uint256 public lastDistribute;
uint256 public distributeCooldown = 300;
require(block.timestamp >= lastDistribute + distributeCooldown, "cooldown active");
lastDistribute = block.timestamp;
```

Example incentive threshold:

```
require(incentiveAmount >= MIN_INCENTIVE_THRESHOLD, "incentive too small");
```

13. Front-Running and Lack of Access Control in `distributeOther` May Allow Incentive Farming

Severity	Low
Finding ID	BH-2021-09-bvecvx-13
Target	veCVX/contracts/locker/CvxStakingProxy.sol
Function name	distributeOther

Description

The `distributeOther` function is externally callable and permits any user to trigger the distribution of arbitrary ERC20 tokens (except `crv` and `cvxCrv`) held by the contract. The caller receives a portion of the token balance as an incentive (`callIncentive`). This public design, while useful for decentralization, can be exploited by bots that repeatedly monitor and front-run deposits to extract small incentive rewards (incentive farming). There are no cooldowns, minimum thresholds, or role restrictions to prevent abuse or inefficiency.

```
function distributeOther(IERC20 _token) external {
        require( address(_token) != crv && address(_token) != cvxCrv, "not
allowed");

        uint256 bal = _token.balanceOf(address(this));

        if (bal > 0) {
            uint256 incentiveAmount = bal.mul(callIncentive).div(denominator);
            bal = bal.sub(incentiveAmount);

            //send incentives
            _token.safeTransfer(msg.sender,incentiveAmount);

            //approve
            _token.safeApprove(rewards, 0);
            _token.safeApprove(rewards, uint256(-1));

            //update rewards
            ICvxLocker(rewards).notifyRewardAmount(address(_token), bal);

            emit RewardsDistributed(address(_token), bal);
```

```
        }
    }
```

Recommendation

Enhance the function's safety and efficiency by: 1. Enforcing a minimum balance threshold to prevent gas-wasting calls and micro-incentives. 2. Adding an optional cooldown period between successive calls to throttle repeated attempts. 3. Optionally restricting callable tokens to a predefined allowlist to prevent misuse.

Example:

```
require(bal >= MIN_REWARD_AMOUNT, "reward too small");
require(block.timestamp >= lastDistributeOther[_token] + cooldown, "cooldown
active");
```

14. Function `reinvest` returns zero regardless of actual reinvestment

Severity	Low
Finding ID	BH-2021-09-bvecvx-14
Target	veCVX/contracts/veCVXStrategy.sol
Function name	reinvest

Description

The `reinvest` function is declared to return a `uint256 reinvested` value but does not assign any value to it throughout the function body. As a result, it always returns 0 regardless of the amount actually redeposited into veCVX. This can mislead external callers or tools that rely on the returned value to track reinvestment activity.

```
function reinvest() external whenNotPaused returns (uint256 reinvested) {
        _onlyGovernance();

        if (processLocksOnReinvest) {
            // Withdraw all we can
            LOCKER.processExpiredLocks(false);
        }

        // Redeposit all into veCVX
        uint256 toDeposit = IERC20Upgradeable(CVX).balanceOf(address(this));

        // Redeposit into veCVX
        LOCKER.lock(address(this), toDeposit, LOCKER.maximumBoostPayment());
    }
```

Recommendation

Assign the value of `toDeposit` to the `reinvested` variable before returning. For example: `reinvested = toDeposit;`