BlockSAFU

ADVANCE MANUAL
SMART CONTRACT AUDIT

Project: BNOU Pool

Website: https://bitnou.com/

PASSED

BlockSAFU Score:

97

Contract Address:

0x152f09A78360A33a8364e7C09d435B73bCb0996e

DISCLAMER

BlockSAFU has completed this report to provide a summary of the Smart Contract functions, and any security, dependency, or cybersecurity vulnerabilities. This is often a constrained report on our discoveries based on our investigation and understanding of the current programming versions as of this report's date. To understand the full scope of our analysis, it is vital for you to at the date of this report. To understand the full scope of our analysis, you need to review the complete report. Although we have done our best in conducting our investigation and creating this report, it is vital to note that you should not depend on this report and cannot make any claim against BlockSAFU or its Subsidiaries and Team members on the premise of what has or has not been included in the report. Please remember to conduct your independent examinations before making any investment choices. We do not provide investment advice or in any way claim to determine if the project will be successful or not.

By perusing this report or any portion of it, you concur to the terms of this disclaimer. In the unlikely situation where you do not concur with the terms, you should immediately terminate reading this report, and erase and discard any duplicates of this report downloaded and/or printed by you. This report is given for data purposes as it were and on a non-reliance premise and does not constitute speculation counsel. No one should have any right to depend on the report or its substance, and BlockSAFU and its members (including holding companies, shareholders, backups, representatives, chiefs, officers, and other agents) BlockSAFU and its subsidiaries owe no obligation of care towards you or any other person, nor does BlockSAFU make any guarantee or representation to any individual on the precision or completeness of the report.

ABOUT THE AUDITOR:

BlockSAFU (BSAFU) is an Anti-Scam Token Utility that reviews Smart Contracts and Token information to Identify Rug Pull and Honey Pot scamming activity. BlockSAFUs Development Team consists of several Smart Contract creators, Auditors Developers, and Blockchain experts. BlockSAFU provides solutions, prevents, and hunts down scammers. BSAFU is a utility token with features Audit, KYC, Token Generators, and Bounty Scammers. It will enrich the crypto ecosystem.

OVERVIEW

Mint Function

- No mint functions.

Fees

- Buy 0% (No fees).
- Sell 0% (No fees).

Tx Amount

- Owner cannot set a max tx amount.

Transfer Pausable

- Owner cannot pause.

Blacklist

- Owner cannot blacklist.

Ownership

- Owner cannot take back ownership.

Proxy

- This contract has no proxy.

Anti Whale

- Owner cannot limit the number of wallet holdings.

Trading Cooldown

- Owner cannot set the selling time interval.

SMART CONTRACT REVIEW

Token Name	**BNOUPool**
Contract Address	0x152f09A78360A33a8364e7C09d435B73bCb0996e
Deployer Address	0x6b2a856A8954aa86eA66f9729597f4078D03e7a9
Owner Address	0x47a4ea43c6cf05e2541a76903f06d4b24fa4cc81
Gas Used for Buy	*will be updated after the DEX listing*
Gas Used for Sell	*will be updated after the DEX listing*
Contract Created	Sep-21-2022 11:43:59 PM +UTC
Initial Liquidity	*will be updated after the DEX listing*
Liquidity Status	Locked
Unlocked Date	*will be updated after the DEX listing*
Verified CA	Yes
Compiler	v0.8.15+commit.e14f2714
Optimization	Yes with 200 runs
Sol License	MIT License
Top 5 Holders	*will be updated after the DEX listing*
Other	default evmVersion

Team Review

The Bnou team has a nice website, their website is professionally built and the Smart contract is well developed, their social media is growing with over 910 people in their telegram group (count in audit date).

Official Website And Social Media

Website: https://bitnou.com/

Telegram Group:

https://t.me/bitnouofficial_english

https://t.me/bitnouofficial_French

https://t.me/bitnouofficial_Spanish

https://t.me/bitnouofficial_Creole

https://t.me/bitnouofficial_news

Discord: https://discord.com/invite/5Qb4bM7zYA

MANUAL CODE REVIEW

Minor-risk

0 minor-risk code issue found

Could be fixed, and will not bring problems.

Medium-risk

0 medium-risk code issues found

Should be fixed, could bring problems.

High-Risk

0 high-risk code issues found

Must be fixed, and will bring problems.

Critical-Risk

0 critical-risk code issues found

Must be fixed, and will bring problems.

EXTRA NOTES SMART CONTRACT

1. IERC20

```
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns
(uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to
`to`.
     *
     * Returns a boolean value indicating whether the operation
succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address to, uint256 amount) external returns
(bool);

    /**
     * @dev Returns the remaining number of tokens that `spender`
will be
     * allowed to spend on behalf of `owner` through
{transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are
called.
     */
    function allowance(address owner, address spender) external
view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the
```

```
caller's tokens.
     *
     * Returns a boolean value indicating whether the operation
succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this
method brings the risk
     * that someone may use both the old and the new allowance by
unfortunate
     * transaction ordering. One possible solution to mitigate this
race
     * condition is to first reduce the spender's allowance to 0
and set the
     * desired value afterwards:
     *
https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external
returns (bool);

    /**
     * @dev Moves `amount` tokens from `from` to `to` using the
     * allowance mechanism. `amount` is then deducted from the
caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation
succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account
(`from`) to
```

```
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to,
uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an
`owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender,
uint256 value);
}
```

IERC20 Normal Base Template

2. BNOUPool Contract

```
contract BNOUPool is Ownable, Pausable {
    using SafeERC20 for IERC20;

    struct UserInfo {
        uint256 shares; // number of shares for a user.
        uint256 lastDepositedTime; // keep track of deposited time
for potential penalty.
        uint256 bnouAtLastUserAction; // keep track of bnou
deposited at the last user action.
        uint256 lastUserActionTime; // keep track of the last user
action time.
        uint256 lockStartTime; // lock start time.
        uint256 lockEndTime; // lock end time.
        uint256 userBoostedShare; // boost share, in order to give
the user higher reward. The user only enjoys the reward, so the
principal needs to be recorded as a debt.
        bool locked; //lock status.
        uint256 lockedAmount; // amount deposited during lock
period.
    }

    IERC20 public immutable token; // bnou token.

    IMasterChefV2 public immutable masterchefV2;

    address public boostContract; // boost contract used in
Masterchef.
    address public VBnou;

    mapping(address => UserInfo) public userInfo;
    mapping(address => bool) public freePerformanceFeeUsers; //
free performance fee users.
    mapping(address => bool) public freeWithdrawFeeUsers; // free
withdraw fee users.
    mapping(address => bool) public freeOverdueFeeUsers; // free
overdue fee users.

    uint256 public totalShares;
    address public admin;
    address public treasury;
```

```
    address public operator;
    uint256 public bnouPoolPID;
    uint256 public totalBoostDebt; // total boost debt.
    uint256 public totalLockedAmount; // total lock amount.

    uint256 public constant MAX_PERFORMANCE_FEE = 2000; // 20%
    uint256 public constant MAX_WITHDRAW_FEE = 500; // 5%
    uint256 public constant MAX_OVERDUE_FEE = 100 * 1e10; // 100%
    uint256 public constant MAX_WITHDRAW_FEE_PERIOD = 1 weeks; //
1 week
    uint256 public constant MIN_LOCK_DURATION = 1 weeks; // 1 week
    uint256 public constant MAX_LOCK_DURATION_LIMIT = 1000 days;
// 1000 days
    uint256 public constant BOOST_WEIGHT_LIMIT = 5000 * 1e10; //
5000%
    uint256 public constant PRECISION_FACTOR = 1e12; // precision
factor.
    uint256 public constant PRECISION_FACTOR_SHARE = 1e28; //
precision factor for share.
    uint256 public constant MIN_DEPOSIT_AMOUNT = 0.00001 ether;
    uint256 public constant MIN_WITHDRAW_AMOUNT = 0.00001 ether;
    uint256 public UNLOCK_FREE_DURATION = 1 weeks; // 1 week
    uint256 public MAX_LOCK_DURATION = 365 days; // 365 days
    uint256 public DURATION_FACTOR = 365 days; // 365 days, in
order to calculate user additional boost.
    uint256 public DURATION_FACTOR_OVERDUE = 180 days; // 180
days, in order to calculate overdue fee.
    uint256 public BOOST_WEIGHT = 100 * 1e10; // 100%

    uint256 public performanceFee = 200; // 2%
    uint256 public performanceFeeContract = 200; // 2%
    uint256 public withdrawFee = 10; // 0.1%
    uint256 public withdrawFeeContract = 10; // 0.1%
    uint256 public overdueFee = 100 * 1e10; // 100%
    uint256 public withdrawFeePeriod = 72 hours; // 3 days

    event Deposit(address indexed sender, uint256 amount, uint256
shares, uint256 duration, uint256 lastDepositedTime);
    event Withdraw(address indexed sender, uint256 amount, uint256
shares);
    event Harvest(address indexed sender, uint256 amount);
    event Pause();
```

```
    event Unpause();
    event Init();
    event Lock(
        address indexed sender,
        uint256 lockedAmount,
        uint256 shares,
        uint256 lockedDuration,
        uint256 blockTimestamp
    );
    event Unlock(address indexed sender, uint256 amount, uint256
blockTimestamp);
    event NewAdmin(address admin);
    event NewTreasury(address treasury);
    event NewOperator(address operator);
    event NewBoostContract(address boostContract);
    event NewVBnouContract(address VBnou);
    event FreeFeeUser(address indexed user, bool indexed free);
    event NewPerformanceFee(uint256 performanceFee);
    event NewPerformanceFeeContract(uint256
performanceFeeContract);
    event NewWithdrawFee(uint256 withdrawFee);
    event NewOverdueFee(uint256 overdueFee);
    event NewWithdrawFeeContract(uint256 withdrawFeeContract);
    event NewWithdrawFeePeriod(uint256 withdrawFeePeriod);
    event NewMaxLockDuration(uint256 maxLockDuration);
    event NewDurationFactor(uint256 durationFactor);
    event NewDurationFactorOverdue(uint256 durationFactorOverdue);
    event NewUnlockFreeDuration(uint256 unlockFreeDuration);
    event NewBoostWeight(uint256 boostWeight);

    constructor(
        IERC20 _token,
        IMasterChefV2 _masterchefV2,
        uint256 _pid,
        address _initializer
    ) {
        token = _token;
        masterchefV2 = _masterchefV2;
        bnouPoolPID = _pid;
        _transferOwnership(_initializer);
    }
```

```
    /**
     * @notice Deposits a dummy token to `MASTER_CHEF` MCV2.
     * It will transfer all the `dummyToken` in the tx sender
address.
     * @param dummyToken The address of the token to be deposited
into MCV2.
     */
    function init(IERC20 dummyToken) external onlyOwner {
        uint256 balance = dummyToken.balanceOf(msg.sender);
        require(balance != 0, "Balance must exceed 0");
        dummyToken.safeTransferFrom(msg.sender, address(this),
balance);
        dummyToken.approve(address(masterchefV2), balance);
        masterchefV2.deposit(bnouPoolPID, balance);
        emit Init();
    }

    /**
     * @notice Checks if the msg.sender is the admin address.
     */
    modifier onlyAdmin() {
        require(msg.sender == admin, "admin: wut?");
        _;
    }

    /**
     * @notice Checks if the msg.sender is either the bnou owner
address or the operator address.
     */
    modifier onlyOperatorOrBnouOwner(address _user) {
        require(msg.sender == _user || msg.sender == operator,
"Not operator or bnou owner");
        _;
    }

    /**
     * @notice Update user info in Boost Contract.
     * @param _user: User address
     */
    function updateBoostContractInfo(address _user) internal {
        if (boostContract != address(0)) {
            UserInfo storage user = userInfo[_user];
```

```
            uint256 lockDuration = user.lockEndTime -
user.lockStartTime;
            IBoostContract(boostContract).onBnouPoolUpdate(
                _user,
                user.lockedAmount,
                lockDuration,
                totalLockedAmount,
                DURATION_FACTOR
            );
        }
    }

    /**
     * @notice Update user share When need to unlock or charges a
fee.
     * @param _user: User address
     */
    function updateUserShare(address _user) internal {
        UserInfo storage user = userInfo[_user];
        if (user.shares > 0) {
            if (user.locked) {
                // Calculate the user's current token amount and
update related parameters.
                uint256 currentAmount = (balanceOf() *
(user.shares)) / totalShares - user.userBoostedShare;
                totalBoostDebt -= user.userBoostedShare;
                user.userBoostedShare = 0;
                totalShares -= user.shares;
                //Charge a overdue fee after the free duration has
expired.
                if (!freeOverdueFeeUsers[_user] &&
((user.lockEndTime + UNLOCK_FREE_DURATION) < block.timestamp)) {
                    uint256 earnAmount = currentAmount -
user.lockedAmount;
                    uint256 overdueDuration = block.timestamp -
user.lockEndTime - UNLOCK_FREE_DURATION;
                    if (overdueDuration > DURATION_FACTOR_OVERDUE)
{
                        overdueDuration = DURATION_FACTOR_OVERDUE;
                    }
                    // Rates are calculated based on the user's
overdue duration.
```

```
                    uint256 overdueWeight = (overdueDuration *
overdueFee) / DURATION_FACTOR_OVERDUE;
                    uint256 currentOverdueFee = (earnAmount *
overdueWeight) / PRECISION_FACTOR;
                    token.safeTransfer(treasury,
currentOverdueFee);
                    currentAmount -= currentOverdueFee;
                }
                // Recalculate the user's share.
                uint256 pool = balanceOf();
                uint256 currentShares;
                if (totalShares != 0) {
                    currentShares = (currentAmount * totalShares)
/ (pool - currentAmount);
                } else {
                    currentShares = currentAmount;
                }
                user.shares = currentShares;
                totalShares += currentShares;
                // After the lock duration, update related
parameters.
                if (user.lockEndTime < block.timestamp) {
                    user.locked = false;
                    user.lockStartTime = 0;
                    user.lockEndTime = 0;
                    totalLockedAmount -= user.lockedAmount;
                    user.lockedAmount = 0;
                    emit Unlock(_user, currentAmount,
block.timestamp);
                }
            } else if (!freePerformanceFeeUsers[_user]) {
                // Calculate Performance fee.
                uint256 totalAmount = (user.shares * balanceOf())
/ totalShares;
                totalShares -= user.shares;
                user.shares = 0;
                uint256 earnAmount = totalAmount -
user.bnouAtLastUserAction;
                uint256 feeRate = performanceFee;
                if (_isContract(_user)) {
                    feeRate = performanceFeeContract;
                }
```

```
                uint256 currentPerformanceFee = (earnAmount *
feeRate) / 10000;
                if (currentPerformanceFee > 0) {
                    token.safeTransfer(treasury,
currentPerformanceFee);
                    totalAmount -= currentPerformanceFee;
                }
                // Recalculate the user's share.
                uint256 pool = balanceOf();
                uint256 newShares;
                if (totalShares != 0) {
                    newShares = (totalAmount * totalShares) /
(pool - totalAmount);
                } else {
                    newShares = totalAmount;
                }
                user.shares = newShares;
                totalShares += newShares;
            }
        }
    }

    /**
     * @notice Unlock user bnou funds.
     * @dev Only possible when contract not paused.
     * @param _user: User address
     */
    function unlock(address _user) external
onlyOperatorOrBnouOwner(_user) whenNotPaused {
        UserInfo storage user = userInfo[_user];
        require(user.locked && user.lockEndTime < block.timestamp,
"Cannot unlock yet");
        depositOperation(0, 0, _user);
    }

    /**
     * @notice Deposit funds into the Bnou Pool.
     * @dev Only possible when contract not paused.
     * @param _amount: number of tokens to deposit (in BNOU)
     * @param _lockDuration: Token lock duration
     */
    function deposit(uint256 _amount, uint256 _lockDuration)
```

```
external whenNotPaused {
        require(_amount > 0 || _lockDuration > 0, "Nothing to
deposit");
        depositOperation(_amount, _lockDuration, msg.sender);
    }

    /**
     * @notice The operation of deposite.
     * @param _amount: number of tokens to deposit (in BNOU)
     * @param _lockDuration: Token lock duration
     * @param _user: User address
     */
    function depositOperation(
        uint256 _amount,
        uint256 _lockDuration,
        address _user
    ) internal {
        UserInfo storage user = userInfo[_user];
        if (user.shares == 0 || _amount > 0) {
            require(_amount > MIN_DEPOSIT_AMOUNT, "Deposit amount
must be greater than MIN_DEPOSIT_AMOUNT");
        }
        // Calculate the total lock duration and check whether the
lock duration meets the conditions.
        uint256 totalLockDuration = _lockDuration;
        if (user.lockEndTime >= block.timestamp) {
            // Adding funds during the lock duration is equivalent
to re-locking the position, needs to update some variables.
            if (_amount > 0) {
                user.lockStartTime = block.timestamp;
                totalLockedAmount -= user.lockedAmount;
                user.lockedAmount = 0;
            }
            totalLockDuration += user.lockEndTime -
user.lockStartTime;
        }
        require(_lockDuration == 0 || totalLockDuration >=
MIN_LOCK_DURATION, "Minimum lock period is one week");
        require(totalLockDuration <= MAX_LOCK_DURATION, "Maximum
lock period exceeded");

        if (VBnou != address(0)) {
```

```
            IVBnou(VBnou).deposit(_user, _amount, _lockDuration);
        }

        // Harvest tokens from Masterchef.
        harvest();

        // Handle stock funds.
        if (totalShares == 0) {
            uint256 stockAmount = available();
            token.safeTransfer(treasury, stockAmount);
        }
        // Update user share.
        updateUserShare(_user);

        // Update lock duration.
        if (_lockDuration > 0) {
            if (user.lockEndTime < block.timestamp) {
                user.lockStartTime = block.timestamp;
                user.lockEndTime = block.timestamp +
_lockDuration;
            } else {
                user.lockEndTime += _lockDuration;
            }
            user.locked = true;
        }

        uint256 currentShares;
        uint256 currentAmount;
        uint256 userCurrentLockedBalance;
        uint256 pool = balanceOf();
        if (_amount > 0) {
            token.safeTransferFrom(_user, address(this), _amount);
            currentAmount = _amount;
        }

        // Calculate lock funds
        if (user.shares > 0 && user.locked) {
            userCurrentLockedBalance = (pool * user.shares) /
totalShares;
            currentAmount += userCurrentLockedBalance;
            totalShares -= user.shares;
            user.shares = 0;
```

```
            // Update lock amount
            if (user.lockStartTime == block.timestamp) {
                user.lockedAmount = userCurrentLockedBalance;
                totalLockedAmount += user.lockedAmount;
            }
        }
        if (totalShares != 0) {
            currentShares = (currentAmount * totalShares) / (pool
- userCurrentLockedBalance);
        } else {
            currentShares = currentAmount;
        }

        // Calculate the boost weight share.
        if (user.lockEndTime > user.lockStartTime) {
            // Calculate boost share.
            uint256 boostWeight = ((user.lockEndTime -
user.lockStartTime) * BOOST_WEIGHT) / DURATION_FACTOR;
            uint256 boostShares = (boostWeight * currentShares) /
PRECISION_FACTOR;
            currentShares += boostShares;
            user.shares += currentShares;

            // Calculate boost share , the user only enjoys the
reward, so the principal needs to be recorded as a debt.
            uint256 userBoostedShare = (boostWeight *
currentAmount) / PRECISION_FACTOR;
            user.userBoostedShare += userBoostedShare;
            totalBoostDebt += userBoostedShare;

            // Update lock amount.
            user.lockedAmount += _amount;
            totalLockedAmount += _amount;

            emit Lock(_user, user.lockedAmount, user.shares,
(user.lockEndTime - user.lockStartTime), block.timestamp);
        } else {
            user.shares += currentShares;
        }

        if (_amount > 0 || _lockDuration > 0) {
```

```
            user.lastDepositedTime = block.timestamp;
        }
        totalShares += currentShares;

        user.bnouAtLastUserAction = (user.shares * balanceOf()) /
totalShares - user.userBoostedShare;
        user.lastUserActionTime = block.timestamp;

        // Update user info in Boost Contract.
        updateBoostContractInfo(_user);

        emit Deposit(_user, _amount, currentShares, _lockDuration,
block.timestamp);
    }

    /**
     * @notice Withdraw funds from the Bnou Pool.
     * @param _amount: Number of amount to withdraw
     */
    function withdrawByAmount(uint256 _amount) public
whenNotPaused {
        require(_amount > MIN_WITHDRAW_AMOUNT, "Withdraw amount
must be greater than MIN_WITHDRAW_AMOUNT");
        withdrawOperation(0, _amount);
    }

    /**
     * @notice Withdraw funds from the Bnou Pool.
     * @param _shares: Number of shares to withdraw
     */
    function withdraw(uint256 _shares) public whenNotPaused {
        require(_shares > 0, "Nothing to withdraw");
        withdrawOperation(_shares, 0);
    }

    /**
     * @notice The operation of withdraw.
     * @param _shares: Number of shares to withdraw
     * @param _amount: Number of amount to withdraw
     */
    function withdrawOperation(uint256 _shares, uint256 _amount)
internal {
```

```
        UserInfo storage user = userInfo[msg.sender];
        require(_shares <= user.shares, "Withdraw amount exceeds
balance");
        require(user.lockEndTime < block.timestamp, "Still in
lock");

        if (VBnou != address(0)) {
            IVBnou(VBnou).withdraw(msg.sender);
        }

        // Calculate the percent of withdraw shares, when
unlocking or calculating the Performance fee, the shares will be
updated.
        uint256 currentShare = _shares;
        uint256 sharesPercent = (_shares * PRECISION_FACTOR_SHARE)
/ user.shares;

        // Harvest token from MasterchefV2.
        harvest();

        // Update user share.
        updateUserShare(msg.sender);

        if (_shares == 0 && _amount > 0) {
            uint256 pool = balanceOf();
            currentShare = (_amount * totalShares) / pool; //
Calculate equivalent shares
            if (currentShare > user.shares) {
                currentShare = user.shares;
            }
        } else {
            currentShare = (sharesPercent * user.shares) /
PRECISION_FACTOR_SHARE;
        }
        uint256 currentAmount = (balanceOf() * currentShare) /
totalShares;
        user.shares -= currentShare;
        totalShares -= currentShare;

        // Calculate withdraw fee
        if (!freeWithdrawFeeUsers[msg.sender] && (block.timestamp
< user.lastDepositedTime + withdrawFeePeriod)) {
```

```
            uint256 feeRate = withdrawFee;
            if (_isContract(msg.sender)) {
                feeRate = withdrawFeeContract;
            }
            uint256 currentWithdrawFee = (currentAmount * feeRate)
/ 10000;
            token.safeTransfer(treasury, currentWithdrawFee);
            currentAmount -= currentWithdrawFee;
        }

        token.safeTransfer(msg.sender, currentAmount);

        if (user.shares > 0) {
            user.bnouAtLastUserAction = (user.shares *
balanceOf()) / totalShares;
        } else {
            user.bnouAtLastUserAction = 0;
        }

        user.lastUserActionTime = block.timestamp;

        // Update user info in Boost Contract.
        updateBoostContractInfo(msg.sender);

        emit Withdraw(msg.sender, currentAmount, currentShare);
    }

    /**
     * @notice Withdraw all funds for a user
     */
    function withdrawAll() external {
        withdraw(userInfo[msg.sender].shares);
    }

    /**
     * @notice Harvest pending BNOU tokens from MasterChef
     */
    function harvest() internal {
        uint256 pendingBnou =
masterchefV2.pendingBnou(bnouPoolPID, address(this));
        if (pendingBnou > 0) {
            uint256 balBefore = available();
```

```
            masterchefV2.withdraw(bnouPoolPID, 0);
            uint256 balAfter = available();
            emit Harvest(msg.sender, (balAfter - balBefore));
        }
    }

    /**
     * @notice Set admin address
     * @dev Only callable by the contract owner.
     */
    function setAdmin(address _admin) external onlyOwner {
        require(_admin != address(0), "Cannot be zero address");
        admin = _admin;
        emit NewAdmin(admin);
    }

    /**
     * @notice Set treasury address
     * @dev Only callable by the contract owner.
     */
    function setTreasury(address _treasury) external onlyOwner {
        require(_treasury != address(0), "Cannot be zero
address");
        treasury = _treasury;
        emit NewTreasury(treasury);
    }

    /**
     * @notice Set operator address
     * @dev Callable by the contract owner.
     */
    function setOperator(address _operator) external onlyOwner {
        require(_operator != address(0), "Cannot be zero
address");
        operator = _operator;
        emit NewOperator(operator);
    }

    /**
     * @notice Set Boost Contract address
     * @dev Callable by the contract admin.
     */
```

```
    function setBoostContract(address _boostContract) external
onlyAdmin {
        require(_boostContract != address(0), "Cannot be zero
address");
        boostContract = _boostContract;
        emit NewBoostContract(boostContract);
    }

    /**
     * @notice Set VBnou Contract address
     * @dev Callable by the contract admin.
     */
    function setVBnouContract(address _VBnou) external onlyAdmin {
        require(_VBnou != address(0), "Cannot be zero address");
        VBnou = _VBnou;
        emit NewVBnouContract(VBnou);
    }

    /**
     * @notice Set free performance fee address
     * @dev Only callable by the contract admin.
     * @param _user: User address
     * @param _free: true:free false:not free
     */
    function setFreePerformanceFeeUser(address _user, bool _free)
external onlyAdmin {
        require(_user != address(0), "Cannot be zero address");
        freePerformanceFeeUsers[_user] = _free;
        emit FreeFeeUser(_user, _free);
    }

    /**
     * @notice Set free overdue fee address
     * @dev Only callable by the contract admin.
     * @param _user: User address
     * @param _free: true:free false:not free
     */
    function setOverdueFeeUser(address _user, bool _free) external
onlyAdmin {
        require(_user != address(0), "Cannot be zero address");
        freeOverdueFeeUsers[_user] = _free;
        emit FreeFeeUser(_user, _free);
```

```
    }

    /**
     * @notice Set free withdraw fee address
     * @dev Only callable by the contract admin.
     * @param _user: User address
     * @param _free: true:free false:not free
     */
    function setWithdrawFeeUser(address _user, bool _free)
external onlyAdmin {
        require(_user != address(0), "Cannot be zero address");
        freeWithdrawFeeUsers[_user] = _free;
        emit FreeFeeUser(_user, _free);
    }

    /**
     * @notice Set performance fee
     * @dev Only callable by the contract admin.
     */
    function setPerformanceFee(uint256 _performanceFee) external
onlyAdmin {
        require(_performanceFee <= MAX_PERFORMANCE_FEE,
"performanceFee cannot be more than MAX_PERFORMANCE_FEE");
        performanceFee = _performanceFee;
        emit NewPerformanceFee(performanceFee);
    }

    /**
     * @notice Set performance fee for contract
     * @dev Only callable by the contract admin.
     */
    function setPerformanceFeeContract(uint256
_performanceFeeContract) external onlyAdmin {
        require(
            _performanceFeeContract <= MAX_PERFORMANCE_FEE,
            "performanceFee cannot be more than
MAX_PERFORMANCE_FEE"
        );
        performanceFeeContract = _performanceFeeContract;
        emit NewPerformanceFeeContract(performanceFeeContract);
    }
```

```
    /**
     * @notice Set withdraw fee
     * @dev Only callable by the contract admin.
     */
    function setWithdrawFee(uint256 _withdrawFee) external
onlyAdmin {
        require(_withdrawFee <= MAX_WITHDRAW_FEE, "withdrawFee
cannot be more than MAX_WITHDRAW_FEE");
        withdrawFee = _withdrawFee;
        emit NewWithdrawFee(withdrawFee);
    }

    /**
     * @notice Set overdue fee
     * @dev Only callable by the contract admin.
     */
    function setOverdueFee(uint256 _overdueFee) external onlyAdmin
{
        require(_overdueFee <= MAX_OVERDUE_FEE, "overdueFee cannot
be more than MAX_OVERDUE_FEE");
        overdueFee = _overdueFee;
        emit NewOverdueFee(_overdueFee);
    }

    /**
     * @notice Set withdraw fee for contract
     * @dev Only callable by the contract admin.
     */
    function setWithdrawFeeContract(uint256 _withdrawFeeContract)
external onlyAdmin {
        require(_withdrawFeeContract <= MAX_WITHDRAW_FEE,
"withdrawFee cannot be more than MAX_WITHDRAW_FEE");
        withdrawFeeContract = _withdrawFeeContract;
        emit NewWithdrawFeeContract(withdrawFeeContract);
    }

    /**
     * @notice Set withdraw fee period
     * @dev Only callable by the contract admin.
     */
    function setWithdrawFeePeriod(uint256 _withdrawFeePeriod)
external onlyAdmin {
```

```
        require(
            _withdrawFeePeriod <= MAX_WITHDRAW_FEE_PERIOD,
            "withdrawFeePeriod cannot be more than
MAX_WITHDRAW_FEE_PERIOD"
        );
        withdrawFeePeriod = _withdrawFeePeriod;
        emit NewWithdrawFeePeriod(withdrawFeePeriod);
    }

    /**
     * @notice Set MAX_LOCK_DURATION
     * @dev Only callable by the contract admin.
     */
    function setMaxLockDuration(uint256 _maxLockDuration) external
onlyAdmin {
        require(
            _maxLockDuration <= MAX_LOCK_DURATION_LIMIT,
            "MAX_LOCK_DURATION cannot be more than
MAX_LOCK_DURATION_LIMIT"
        );
        MAX_LOCK_DURATION = _maxLockDuration;
        emit NewMaxLockDuration(_maxLockDuration);
    }

    /**
     * @notice Set DURATION_FACTOR
     * @dev Only callable by the contract admin.
     */
    function setDurationFactor(uint256 _durationFactor) external
onlyAdmin {
        require(_durationFactor > 0, "DURATION_FACTOR cannot be
zero");
        DURATION_FACTOR = _durationFactor;
        emit NewDurationFactor(_durationFactor);
    }

    /**
     * @notice Set DURATION_FACTOR_OVERDUE
     * @dev Only callable by the contract admin.
     */
    function setDurationFactorOverdue(uint256
_durationFactorOverdue) external onlyAdmin {
```

```
        require(_durationFactorOverdue > 0,
"DURATION_FACTOR_OVERDUE cannot be zero");
        DURATION_FACTOR_OVERDUE = _durationFactorOverdue;
        emit NewDurationFactorOverdue(_durationFactorOverdue);
    }

    /**
     * @notice Set UNLOCK_FREE_DURATION
     * @dev Only callable by the contract admin.
     */
    function setUnlockFreeDuration(uint256 _unlockFreeDuration)
external onlyAdmin {
        require(_unlockFreeDuration > 0, "UNLOCK_FREE_DURATION
cannot be zero");
        UNLOCK_FREE_DURATION = _unlockFreeDuration;
        emit NewUnlockFreeDuration(_unlockFreeDuration);
    }

    /**
     * @notice Set BOOST_WEIGHT
     * @dev Only callable by the contract admin.
     */
    function setBoostWeight(uint256 _boostWeight) external
onlyAdmin {
        require(_boostWeight <= BOOST_WEIGHT_LIMIT, "BOOST_WEIGHT
cannot be more than BOOST_WEIGHT_LIMIT");
        BOOST_WEIGHT = _boostWeight;
        emit NewBoostWeight(_boostWeight);
    }

    /**
     * @notice Withdraw unexpected tokens sent to the Bnou Pool
     */
    function inCaseTokensGetStuck(address _token) external
onlyAdmin {
        require(_token != address(token), "Token cannot be same as
deposit token");

        uint256 amount = IERC20(_token).balanceOf(address(this));
        IERC20(_token).safeTransfer(msg.sender, amount);
    }
```

```
    /**
     * @notice Trigger stopped state
     * @dev Only possible when contract not paused.
     */
    function pause() external onlyAdmin whenNotPaused {
        _pause();
        emit Pause();
    }

    /**
     * @notice Return to normal state
     * @dev Only possible when contract is paused.
     */
    function unpause() external onlyAdmin whenPaused {
        _unpause();
        emit Unpause();
    }

    /**
     * @notice Calculate Performance fee.
     * @param _user: User address
     * @return Returns Performance fee.
     */
    function calculatePerformanceFee(address _user) public view
returns (uint256) {
        UserInfo storage user = userInfo[_user];
        if (user.shares > 0 && !user.locked &&
!freePerformanceFeeUsers[_user]) {
            uint256 pool = balanceOf() +
calculateTotalPendingBnouRewards();
            uint256 totalAmount = (user.shares * pool) /
totalShares;
            uint256 earnAmount = totalAmount -
user.bnouAtLastUserAction;
            uint256 feeRate = performanceFee;
            if (_isContract(_user)) {
                feeRate = performanceFeeContract;
            }
            uint256 currentPerformanceFee = (earnAmount * feeRate)
/ 10000;
            return currentPerformanceFee;
        }
```

```
        return 0;
    }

    /**
     * @notice Calculate overdue fee.
     * @param _user: User address
     * @return Returns Overdue fee.
     */
    function calculateOverdueFee(address _user) public view
returns (uint256) {
        UserInfo storage user = userInfo[_user];
        if (
            user.shares > 0 &&
            user.locked &&
            !freeOverdueFeeUsers[_user] &&
            ((user.lockEndTime + UNLOCK_FREE_DURATION) <
block.timestamp)
        ) {
            uint256 pool = balanceOf() +
calculateTotalPendingBnouRewards();
            uint256 currentAmount = (pool * (user.shares)) /
totalShares - user.userBoostedShare;
            uint256 earnAmount = currentAmount -
user.lockedAmount;
            uint256 overdueDuration = block.timestamp -
user.lockEndTime - UNLOCK_FREE_DURATION;
            if (overdueDuration > DURATION_FACTOR_OVERDUE) {
                overdueDuration = DURATION_FACTOR_OVERDUE;
            }
            // Rates are calculated based on the user's overdue
duration.
            uint256 overdueWeight = (overdueDuration * overdueFee)
/ DURATION_FACTOR_OVERDUE;
            uint256 currentOverdueFee = (earnAmount *
overdueWeight) / PRECISION_FACTOR;
            return currentOverdueFee;
        }
        return 0;
    }

    /**
     * @notice Calculate Performance Fee Or Overdue Fee
```

```
     * @param _user: User address
     * @return Returns  Performance Fee Or Overdue Fee.
     */
    function calculatePerformanceFeeOrOverdueFee(address _user)
internal view returns (uint256) {
        return calculatePerformanceFee(_user) +
calculateOverdueFee(_user);
    }

    /**
     * @notice Calculate withdraw fee.
     * @param _user: User address
     * @param _shares: Number of shares to withdraw
     * @return Returns Withdraw fee.
     */
    function calculateWithdrawFee(address _user, uint256 _shares)
public view returns (uint256) {
        UserInfo storage user = userInfo[_user];
        if (user.shares < _shares) {
            _shares = user.shares;
        }
        if (!freeWithdrawFeeUsers[msg.sender] && (block.timestamp
< user.lastDepositedTime + withdrawFeePeriod)) {
            uint256 pool = balanceOf() +
calculateTotalPendingBnouRewards();
            uint256 sharesPercent = (_shares * PRECISION_FACTOR) /
user.shares;
            uint256 currentTotalAmount = (pool * (user.shares)) /
                totalShares -
                user.userBoostedShare -
                calculatePerformanceFeeOrOverdueFee(_user);
            uint256 currentAmount = (currentTotalAmount *
sharesPercent) / PRECISION_FACTOR;
            uint256 feeRate = withdrawFee;
            if (_isContract(msg.sender)) {
                feeRate = withdrawFeeContract;
            }
            uint256 currentWithdrawFee = (currentAmount * feeRate)
/ 10000;
            return currentWithdrawFee;
        }
        return 0;
```

```
    }

    /**
     * @notice Calculates the total pending rewards that can be
harvested
     * @return Returns total pending bnou rewards
     */
    function calculateTotalPendingBnouRewards() public view
returns (uint256) {
        uint256 amount = masterchefV2.pendingBnou(bnouPoolPID,
address(this));
        return amount;
    }

    function getPricePerFullShare() external view returns
(uint256) {
        return totalShares == 0 ? 1e18 : (((balanceOf() +
calculateTotalPendingBnouRewards()) * (1e18)) / totalShares);
    }

    /**
     * @notice Current pool available balance
     * @dev The contract puts 100% of the tokens to work.
     */
    function available() public view returns (uint256) {
        return token.balanceOf(address(this));
    }
..
    function balanceOf() public view returns (uint256) {
        return token.balanceOf(address(this)) + totalBoostDebt;
    }


..
    function _isContract(address addr) internal view returns
(bool) {
        uint256 size;
        assembly {
            size := extcodesize(addr)
        }
        return size > 0;
    }
}
```

READ CONTRACT (ONLY NEED TO KNOW)

1. BOOST_WEIGHT
1000000000000 uint256
(Function for read boost weight)

2. BOOST_WEIGHT_LIMIT
50000000000000 address
(Function for read boost weight limit)

3. DURATION_FACTOR
31536000 address
(Function for read duration factor)

4. owner
0x47a4ea43c6cf05e2541a76903f06d4b24fa4cc81 address
(Function for read owner address)

5. VBnou
0x0000000000000000000000000000000000000000 address
(Function for read vbnou address)

6. admin
0x47a4ea43c6cf05e2541a76903f06d4b24fa4cc81 address
(Function for read admin address)

WRITE CONTRACT

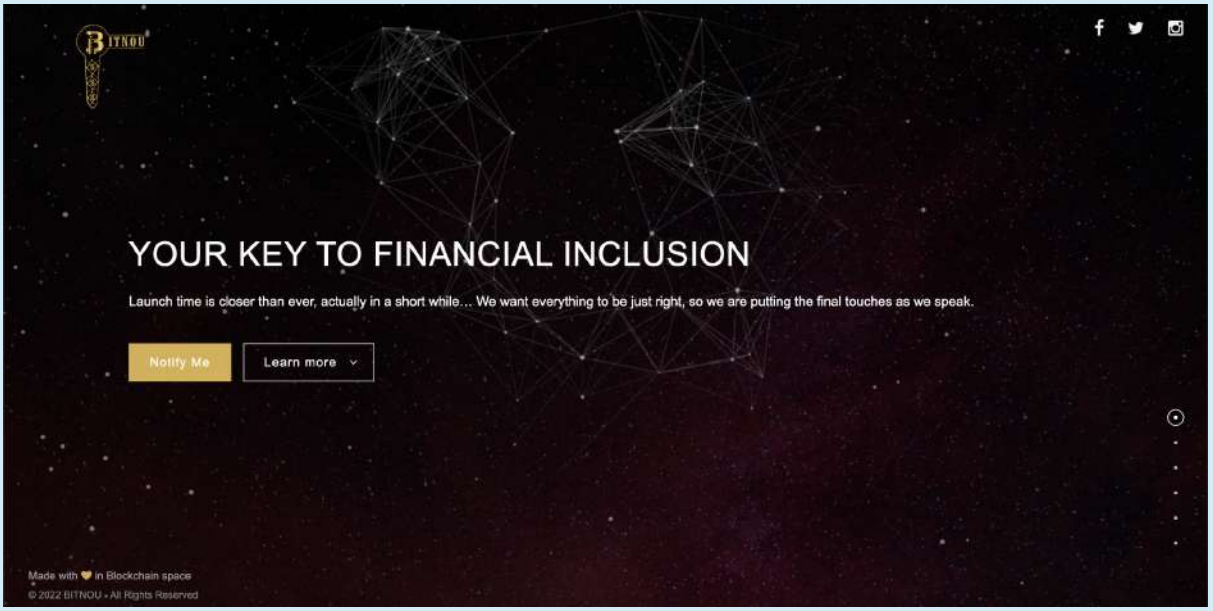
1. renounceOwnership
(Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner)

2. transferOwnership
newOwner (address)
(Its function is to change the owner)

3. setAdmin
_admin (address)
(Its function for set admin address)

4. setVBnouContract
_VBnou (address)
(its function for set bnou contract address)

WEBSITE REVIEW



- **Mobile Friendly**
- **Contains no code error**
- **SSL Secured (By Let's Encrypt SSL)**

Web-Tech stack: Apache, Bootstrap, Animate css

Domain .com (Hostgator) - Tracked by whois

First Contentful Paint:	1.6s
Fully Loaded Time	5.6s
Performance	46%
Accessibility	79%
Best Practices	58%
SEO	80%

RUG-PULL REVIEW

Based on the available information analyzed by us, we come to the following conclusions:

- Locked Liquidity (Locked by pinksale)

  (*Will be updated after DEX listing*)

- TOP 5 Holder.

  (*Will be updated after DEX listing*)

- The Team KYC by Blocksafu

HONEYPOT REVIEW

- Ability to sell.
- The owner is not able to pause the contract.
- The owner can’t set fees

Note: Please check the disclaimer above and note, that the audit makes no statements or warranties on the business model, investment attractiveness, or code sustainability. The report is provided for the only contract mentioned in the report and does not include any other potential contracts deployed by the project owner.