33AUDITS & CO.

Coinwell Audit Report

Introduction

A security review of the Coinwell protocol was conducted, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Samuel as the Security Researcher.

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 vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About Coinwell

This audit is being performed on the core protocol contracts for Coinwell. The contracts in scope include `PriceFeed`, `CWToken`, `VaultManager`, `StabilityPool`, `BorrowerOperations`, `PCV`, and related components. The Coinwell protocol is a decentralized lending and borrowing system that allows users to create collateralized debt positions (vaults) by depositing collateral and borrowing CWUSD tokens. The protocol includes a stability pool for liquidation protection, fee mechanisms, and oracle price feeds for collateral valuation.

Commit: [7ce3c00fc6d58d90db8f450b941860b8cab456e4] - Scope: `PriceFeed`, `CWToken`, `VaultManager`, `StabilityPool`, `BorrowerOperations`, `PCV`, and related components

Audit Methodology

This audit focused primarily on the changes made post-fork from the original protocol, which had already undergone extensive security audits. Our team conducted a targeted review of:

1. **Post-Fork Modifications**: All changes made to the original codebase after the fork, including new contracts and modifications to existing ones
2. **Deployment Chain Differences**: Adaptations made for the target deployment chain (PulseChain), including oracle integrations and token implementations
3. **New Functionality**: Any new features or mechanisms introduced in the fork that were not present in the original audited protocol
4. **Integration Points**: How the modified components interact with the existing, previously audited core protocol logic

The findings presented in this report represent vulnerabilities and issues identified specifically in the post-fork changes and deployment adaptations, rather than issues in the original, heavily audited

protocol codebase.

Severity Definitions

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[C-01]	Fee Logic In CWToken Incorrect	Critical	Fixed
[M-01]	Tellor Oracle Staleness Not Checked Dynamic	Medium	Acknowledged
[M-02]	Missing Slippage Protection In BufferPool.sol Swaps Enables Value Extraction	Medium	Fixed
[M-03]	Permanent Reward Loss Due To Incorrect Accounting Of Actual Amount Transferred	Medium	Acknowledged
[L-01]	Users Viewing Pending Rewards Will See Incorrect Reward Amounts	Low	Fixed

Critical

[C-01] Fee Logic In CWToken Incorrect

Description

The fee logic in CWToken does not work as intended and will revert due to the two step process of first calculating feeRate and next calculating feeAmount because each step will use the 18 decimals from amount, which will result in a value with too many decimals and break the entire fee logic system.

Follow This Example: Including the usage of SafeMath library functions

Step 1 Calculate Fee Rate

```
feeRate = amount.mul(totalBuyFee).div(feeDenominator.mul(10));
```

```
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) return 0;
    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");
    return c;
}
```

- c = 50e18 * 100 = 5000e18
- Check: 5000e18 / 50e18 = 100 ✓
- Returns: 5000e18

1b. feeDenominator.mul(10)

- 10000 * 10 = 100000

1c. div(5000e18, 100000)

```
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b > 0, "SafeMath: division by zero");
    return a / b;
}
```

- feeRate = 5000e18 / 100000 = 0.05e18
- Result: feeRate = 50000000000000000 (0.05 tokens with 18 decimals)

Step 2: Calculate feeAmount

```
feeAmount = amount.mul(feeRate).div(FEE_DENOM);
```

2a. amount.mul(feeRate)

- Using the same mul function as above.
- a = 50e18
- b = 0.05e18
- c = a _ b = 50e18 _ .05e18

- c = 2500000000000000000000000000000000000 (this is 2.5e36)
- the require check in `mul` will succeed
- 2.5e26 / 50e18 = 0.05e18
- So the overflow check passes. The multiplication creates a huge number but it's valid.

2b. div(2.5e36, 10000)

```
- feeAmount = 2.5 e36 / 10000 = 2.5e32
```

Step 3: Calculate remaining amount

```
return amount.sub(feeAmount);
```

```
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a, "SafeMath: subtraction overflow");
    return a - b;
}

- a = 50e18 = 50000000000000000000
- b = 2.5e32 = 250000000000000000000000000000000

Check: b <= a?
250000000000000000000000000000000 <= 50000000000000000000?
NO! This is FALSE!

require(false, "SafeMath: subtraction overflow");
→ REVERTS!
```

Impact

- **Critical Impact**: The fee logic is broken, and currently reverts when it hits .sub. The feeRate is not calculated as an actual rate, but has the same amount of decimals as CWToken (18). This makes the calculation of feeAmount too large and causes the revert. Without the revert it would also lead to an incorrect feeAmount even if it would pass the overflow check.
- **High Likelihood**: Affects all token transfers that trigger fee logic

Recommendations

The addition of first getting the feeRate and then trying to calculate feeAmount is unnecessary. There is already the totalBuyFee and totalSellFee - which are the percentages of fee for each transaction, use these directly with the FEE_DENOM to calculate that exact percentage of fee.

For example:

```
feeAmount = amount.mul(totalBuyFee).div(feeDenominator);
amountAfterFee = amount.sub(feeAmount);
```

Status

Fixed - The developer has corrected the fee calculation logic to use the proper fee percentages directly.

Medium

[M-01] Tellor Oracle Staleness Not Checked

Description

The PriceFeed contract does not validate the staleness of Tellor oracle data. The Chainlink oracle is hardcoded to always return true during its frozen check so the system falls back to using the Tellor oracle - but there is no staleness check. This allows extremely stale price data to be accepted as current, potentially causing severe mispricing across the protocol.

There is a check to revert if the price data is in the future, but no check for staleness.

```
// contracts/PriceFeed.sol:345-347
function _chainlinkIsFrozen() internal pure returns (bool) {
    return true;  // Always frozen!
}

// contracts/PriceFeed.sol:378-380
function _tellorIsFrozen() internal pure returns (bool) {
    return false;  // Never frozen!
}

// contracts/PriceFeed.sol:367-376
function _tellorIsBroken(TellorResponse memory _response) internal view
returns (bool) {
    if (!_response.success) {return true;}
    if (_response.timestamp == 0 || _response.timestamp >
block.timestamp) {return true;}
    if (_response.value == 0) {return true;}
    // NO STALENESS CHECK: Missing check for (block.timestamp -
_response.timestamp > MAX_TIMEOUT)
    return false;
}
```

Impact

The `_tellorIsBroken()` function only validates:

- Call succeeded (success == true)
- Timestamp is not zero or in the future
- Price value is not zero

But it is missing validation that `_response.timestamp` is recent enough to be considered fresh data - which allows logic to continue and execute even when price data is outdated and incorrect, potentially causing harm to the protocol or users.

- **Medium Impact**: Extremely stale price data can be accepted as current, leading to severe mispricing
- **Medium Likelihood**: Affects all price-dependent operations when Tellor oracle is used

Recommendations

Add staleness check for the Tellor oracle in `_tellorIsBroken()`:

```
function _tellorIsBroken(TellorResponse memory _response) internal view
returns (bool) {
    if (!_response.success) {return true;}
    if (_response.timestamp == 0 || _response.timestamp >
block.timestamp) {return true;}
    if (_response.value == 0) {return true;}

    // Add staleness check
    uint256 constant TELLOR_TIMEOUT = 3600; // 1 hour or any staleness
check that your team deems appropriate
    if (block.timestamp - _response.timestamp > TELLOR_TIMEOUT) {return
true;}

    return false;
}
```

Team Response

The development team noted that they plan to run their own Tellor fork and will be reporting prices consistently from their backend. They intend to block external reporters to control reporting and keep operational costs down through selective price updates. The prices should never become stale under their operational model.

Status

Acknowledged - The development team has acknowledged this finding and explained their operational model. They plan to run their own Tellor fork with controlled reporting and consistent backend price updates. While implementing staleness checks would provide additional defense-in-depth, the team has determined it is not necessary given their operational controls.

[M-02] Missing Slippage Protection In BufferPool.sol Swaps Enables Value Extraction

Description

Both `swapCWUSDForCollateral` and `swapCollateralForCWUSD` compute outputs solely from the oracle price and a fee, without user-provided slippage bounds or a deadline. There is no minOut/maxIn constraint and no deadline to limit execution timing. This exposes users to:

- MEV/sandwich attacks between approval and execution
- Execution at stale/oracle-lagged prices

```
function swapCWUSDForCollateral(uint256 _amountCWUSD) external override
returns (uint256 collateralSent) {
    // ...
    uint256 price = priceFeed.fetchPrice();
    // ...
    tokenCWUSD.transferFrom(msg.sender, address(this), _amountCWUSD);
    uint256 grossCollateral = (_amountCWUSD * DECIMAL_PRECISION) /
price;
    uint256 fee =
vaultManager.getRedemptionFeeWithDecay(grossCollateral);
    uint256 collateralToUser = grossCollateral - fee;
}

function swapCollateralForCWUSD(uint256 _collateralAmount) external
payable override returns (uint256 cwusdSent) {
    // ...
    uint256 price = priceFeed.fetchPrice();
    // ...
    uint256 grossCWUSD = (_collateralAmount * price) /
DECIMAL_PRECISION;
    uint256 fee =
vaultManager.getRedemptionFeeWithDecay(_collateralAmount);
    uint256 feeCWUSDEquivalent = (fee * price) / DECIMAL_PRECISION;
    uint256 cwusdToUser = grossCWUSD - feeCWUSDEquivalent;
}
```

Impact

- **Medium Impact**: Users can be sandwiched and receive materially worse execution than expected
- **Medium Likelihood**: Execution can occur at stale or manipulated oracle prices, causing unintended value transfer. Lack of timing bounds enables miners/validators or bots to delay or reorder for profit

Recommendations

Add user slippage and timing guards:

```
swapCWUSDForCollateral(amountCWUSD, minCollateralOut, deadline)
swapCollateralForCWUSD(collateralAmount, minCWUSDOut, deadline)
```

Enforce:

```
require(block.timestamp <= deadline, "BufferPool: expired");
require(collateralToUser >= minCollateralOut, "BufferPool: slippage");
require(cwusdToUser >= minCWUSDOut, "BufferPool: slippage");
```

Status

Fixed - The developer has added slippage protection and deadline constraints to the swap functions.

[M-03] Permanent Reward Loss Due To Incorrect Accounting Of Actual Amount Transferred

Description

The `safeCWTransfer` function sends less than the calculated pending rewards when the contract has insufficient CW balance, but the `rewardDebt` is updated as if the full amount was transferred. This causes users to permanently lose rewards they are owed. Users permanently lose rewards when the MasterChef contract has insufficient CW balance to pay out calculated pending rewards, as the accounting is not handled properly to update users account based on actual amount sent out. The lost rewards can never be claimed again.

```
// In deposit(), withdraw(), enterStaking(), leaveStaking()
uint256 pending =
user.amount.mul(pool.accCWPerShare).div(1e12).sub(user.rewardDebt);
safeCWTransfer(msg.sender, pending);  // May send less than pending
user.rewardDebt = user.amount.mul(pool.accCWPerShare).div(1e12);  //
Updated as if full amount was sent

function safeCWTransfer(address _to, uint256 _amount) internal {
    uint256 CWBal = CW.balanceOf(address(this));
    if (_amount > CWBal) {
        CW.transfer(_to, CWBal);  // Sends less than owed
    } else {
        CW.transfer(_to, _amount);
    }
}
```

Impact

The `rewardDebt` is fully updated to current levels, regardless of if the user was able to receive their full amount of pending rewards or not. There is no mechanism to store any rewards the user should get, but cannot at the moment because the MasterChef does not have enough balance. So the user effectively will have their account updated fully and will lose the rights to that missing reward amount.

- **Medium Impact**: Users permanently lose rewards when the contract has insufficient balance
- **Medium Likelihood**: This issue will only occur when the supply cap is reached and new emissions stop

Example Scenario:

Initial State:

- `user.amount = 1000e18` (1000 tokens staked)
- `pool.accCWPerShare = 1000e12`
- `user.rewardDebt = 500e6` (500 CW already credited)
- Contract CW balance = `100e18` (100 CW available)

User calls deposit():

1. Calculate pending: `pending = 1000e18 * 1000e12 / 1e12 - 500e6 = 500e6` (500 CW owed)
2. safeCWTransfer: Sends only `100e18` (100 CW) due to insufficient balance
3. Update rewardDebt: `user.rewardDebt = 1000e18 * 1000e12 / 1e12 = 1000e6` (credited for 500 CW)
4. User loses: `400e6` (400 CW) **forever**

Recommendations

The remaining amount of rewards the user did not receive can be stored for later retrieval. However, this is an edge case that will only occur many years in the future when the supply cap is reached and new emissions stop. This will be expected behavior by users at that time of the farm's life, as rewards may cut off when the supply cap is reached and users will be claiming very often to get as much as they can before that happens.

Documentation should clearly outline what will happen when the farm emissions end and the supply cap is reached.

Status

Accepted - This is an edge case that will only occur when the supply cap is reached and new emissions stop. The development team has determined this to be expected behavior and will document it appropriately in the whitepaper and documentation.

Low

[L-01] Users Viewing Pending Rewards Will See Incorrect Reward Amounts

Description

The `pendingCW()` function calculates pending rewards using uncapped token amounts while `updatePool()` uses capped amounts due to the token supply limit. This creates a permanent inconsistency where users see inflated pending rewards that can never be claimed.

Users see significantly more pending rewards than they can actually claim when the token cap is reached, causing accounting inconsistency and user trust issues.

The `pendingCW()` function uses the calculated reward amount directly, while `updatePool()` applies the token cap through `_safeMint()`:

```
// In updatePool() - uses capped amount
uint256 CWReward =
baseEmission.mul(pool.allocPoint).div(totalAllocPoint);
uint256 minted = _safeMint(address(this), CWReward);  // Capped by token
supply
pool.accCWPerShare =
pool.accCWPerShare.add(minted.mul(1e12).div(lpSupply));

// In pendingCW() - uses uncapped amount
uint256 CWReward =
baseEmission.mul(pool.allocPoint).div(totalAllocPoint);
accCWPerShare = accCWPerShare.add(CWReward.mul(1e12).div(lpSupply));  //
No capping!
```

Example Scenario

Initial State:

- CW.totalSupply() = 99,999,000e18 (near cap)
- CW.maxSupply() = 100,000,029e18 (cap)
- pool.accCWPerShare = 1000e12
- user.amount = 1000e18 (1000 tokens staked)
- user.rewardDebt = 1000e6
- lpSupply = 10,000e18

New Rewards:

- baseEmission = 1000e18
- CWReward = 100e18 (calculated)

Here is where the updatePool will take into consideration the cap, so it will use 29e18 for reward calculations, while pendingCW will use the original 100e18.

In `updatePool()` (actual behavior):

```
uint256 minted = _safeMint(address(this), 100e18);  // Returns 29e18
(capped)
pool.accCWPerShare = 1000e12 + (29e18 * 1e12 / 10,000e18);
pool.accCWPerShare = 1002.9e12
```

In `pendingCW()` (user sees):

```
accCWPerShare = 1000e12 + (100e18 * 1e12 / 10,000e18);
accCWPerShare = 1010e12
```

The result will be two different rewards amounts, the actual rewards that a user can claim, and the inflated reward amount that a user is shown.

What `pendingCW()` shows:

```
pending = user.amount.mul(accCWPerShare).div(1e12).sub(user.rewardDebt);
pending = 1000e18 * 1010e12 / 1e12 - 1000e6;
pending = 10 CW tokens
```

What `updatePool()` actually updates to:

```
// After updatePool() runs, accCWPerShare = 1002.9e12
pending = 1000e18 * 1002.9e12 / 1e12 - 1000e6;
pending = 2.9 CW tokens
```

Impact

- **Low Impact**: There will be a disconnect between the rewards that a user can actually claim because of the cap limit, and the rewards that a user is shown that they have available in pendingCW
- **Low Likelihood**: There is no material loss as pendingCW is used only as a view function, but it can provide a bad user experience, especially if they receive much less than they expect

Recommendations

For more accuracy, apply the same capping logic as in `_safeMint`, to get the actual amount of rewards. The actual minting does not have to happen, but running the same logic as `_safeMint` which enforces the cap will make sure there is consistency here.

For Example:

```

```

```
// Apply the same capping logic as _safeMint
uint256 supply = CW.totalSupply();
uint256 cap = CW.maxSupply();
if (supply < cap) {
    if (supply + CWReward > cap) {
        CWReward = cap - supply;
    }
    accCWPerShare = accCWPerShare.add(CWReward.mul(1e12).div(lpSupply));
}
```

Status

Fixed - The developer has updated the pendingCW() function to include cap logic for consistency.

Conclusion

This focused audit of post-fork changes found targeted issues in deployment adaptations and oracle validation, not in the original audited core. The team responded quickly and fixed most items; overall security remains sound with limited follow-ups recommended.

This report was prepared by 33Audits & Co and represents our independent security assessment of the Coinwell protocol smart contracts.