hashlock.

Security Audit

Petcoin AI (Token)

January 2026 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit Scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	11
Audit Resources	11
Dependencies	11
Severity Definitions	12
Status Definitions	13
Audit Findings	14
Centralisation	33
Conclusion	34
Our Methodology	35
Disclaimers	37
About Hashlock	38

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

The Petcoin AI team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

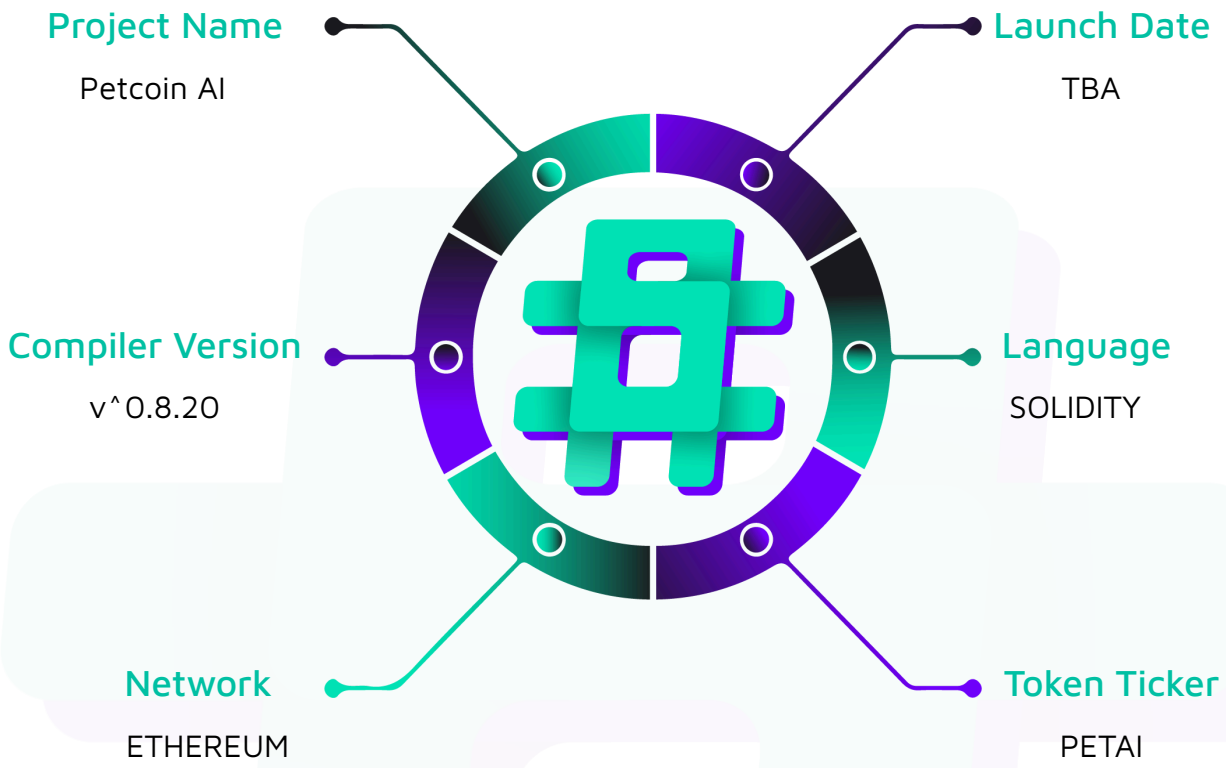
Project Context

Companion Network Unit (CNU) is the PetCoin AI utility token deployed on the Ethereum blockchain. The project combines blockchain technology with a mission-driven use case focused on supporting pet welfare and animal rescue organizations. CNU has a fixed total supply of 1 trillion tokens and implements custom tokenomics, including token burns, holder rewards, and automated charitable allocations through on-chain fee distribution mechanisms.

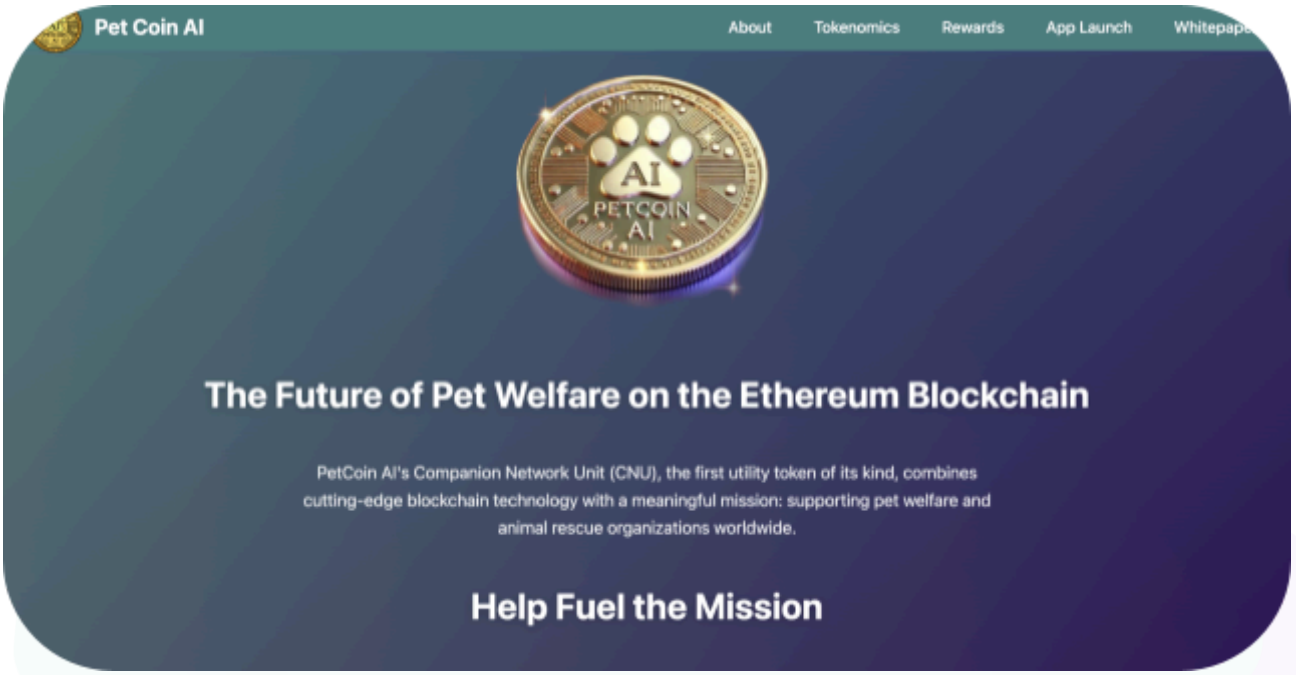
From a deployment perspective, no architectural or security limitations were identified that would restrict the protocol to Ethereum mainnet only. The contracts are suitable for deployment on EVM-compatible Layer 2 or alternative networks.

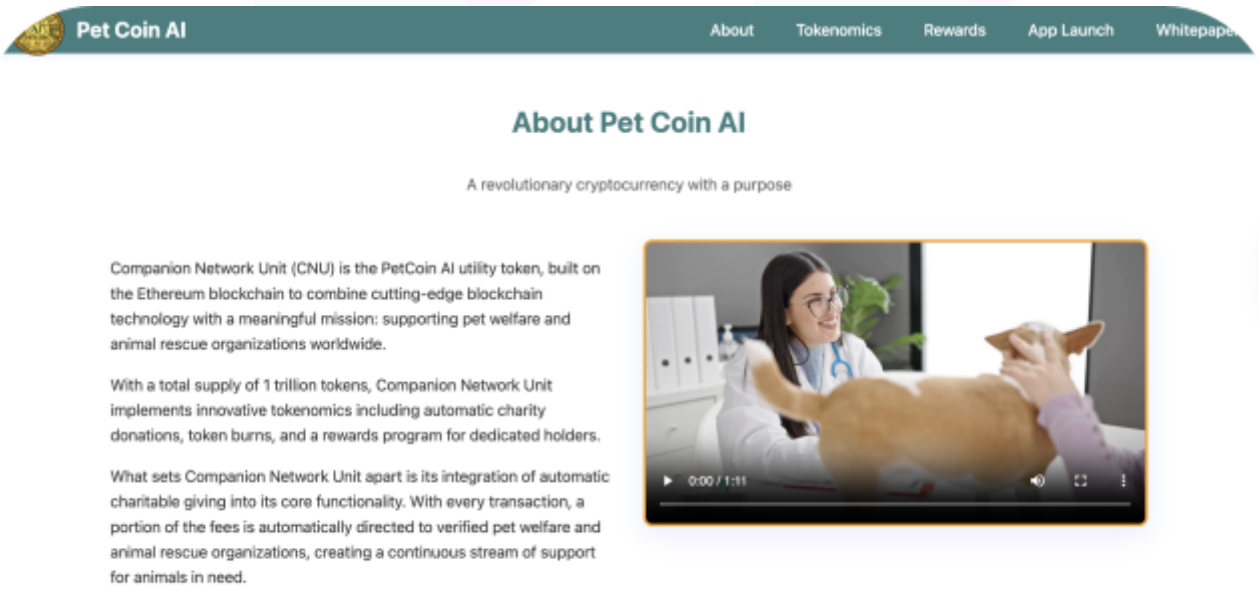
Project Name: Petcoin AI
Project Type: Token
Compiler Version: ^0.8.20
Website: https://petcoinai.info/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the Petcoin AI project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Petcoin AI Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**January, 2026**
Contract 1	AccessGating.sol
Contract 2	CharityVault.sol
Contract 3	StakingVault.sol
Contract 4	TreasuryVault.sol
Contract 5	CNU.sol
Contract 6	ICNUVaults.sol
Contract 7	UniswapV2PriceFeed.sol
Contract 8	VaultBase.sol
Audited GitHub Commit Hash	fea887c13d8ebaf06356bd8053df79e828276641
Fix Review GitHub Commit Hash	

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Vulnerable"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities we have identified have yet to be resolved or acknowledged.

Hashlock found:

2 Medium severity vulnerabilities

1 Low severity vulnerability

1 Gas Optimisations

4 QAs

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

AccessGating.sol
- Allows the operator to:
  - Set tier access threshold
  - Set the price feed contract
  - Set the max acceptable price to receive from the price feed
	Contract achieves this functionality.

CharityVault.sol - Allows the operator to: - Spend the funds toward a recipient	**Contract achieves this functionality.**

StakingVault.sol
- Allows the operator to:
  - Pause/unpause the staking behavior (claim unaffected)
  - Change the penalty for early withdraw
- Allows users to:
  - Stake their CNU
  - Claim their stake and reward after the staking time has passed
  - Withdraw their stake early, minus a small penalty
	Contract achieves this functionality.

TreasuryVault.sol - Allows the operator to: - Spend the funds toward a recipient (through functions `payClaim` and `withdraw`)	**Contract achieves this functionality.**

CNU.sol
- Allows the operator to:
  - Exclude/include an address from fee calculations when transferring tokens
  - Exclude/include an address from limit calculations when transferring tokens
  - Set the percentage of fees that will be burned, sent to the charity vault, and sent to the staking contract in each transfer (total fee cannot exceed 7%)
  - Change the Charity, Treasury and Staking vaults
  - Change the maximum amount of tokens allowed for an address (need to be between 10 million and 50 billion)
  - Change the maximum amount of tokens allowed to be transferred at once (need to be between 10 million and 10 billion)
	Contract achieves this functionality.

UniswapV2PriceFeed.sol
- Allows anyone to:
  - Update the TWAP price (the time elapsed should be at least 30 minutes)
  - Read the latest average price
	Contract does not calculate the price between the latest calculation and now.
	Contract may calculate the TWAP of CNU/USD instead of USD/CNU.

Code Quality

This audit scope involves the smart contracts of the Petcoin AI project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring is recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Petcoin AI project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] UniswapV2PriceFeed#getLatestPrice - Stale TWAP when reading the latest price

Description

The `getLatestPrice` function returns a potentially outdated TWAP price that was last computed during the most recent `update` call. It does not perform any on-the-fly recalculation using current pair data, leading to inaccurate prices if queried between updates (e.g., 29 minutes after the last update when `MIN_UPDATE_INTERVAL` is 30 minutes).

Vulnerability Details

The contract implements a TWAP oracle requiring manual `update()` calls at intervals of at least `MIN_UPDATE_INTERVAL` (1800 seconds). During `update()`, it computes `priceAverageUQ112x112 = (price0Cumulative - priceCumulativeLast) / timeElapsed`, where `price0Cumulative` is extrapolated to the current block timestamp via `currentCumulativePrices()`.

However, `getLatestPrice()` simply scales the stored `priceAverageUQ112x112` without checking staleness or recomputing:

```
return (priceAverageUQ112x112 * 1e18) / (2 ** 112);
```

This represents the average price over the prior interval only. If significant time has passed since `lastUpdateTimestamp` (but less than `MIN_UPDATE_INTERVAL`), the current spot price may have diverged substantially due to trades, yet `getLatestPrice()` ignores this.

No check exists in `getLatestPrice()` against `getTimeSinceUpdate()`, allowing consumers to rely on stale data.

Proof of Concept

1. Deploy contract and call `update()` at T=0, establishing initial TWAP.
2. Wait 29 minutes (1740 seconds); pair reserves change significantly due to trades.
3. Call `getTimeSinceUpdate()` returns 1740 < 1800, so `update()` reverts if attempted.
4. Call `getLatestPrice()` returns TWAP from T=0 interval, not reflecting current reserves/price at T=1740.

Note that this could also happen for intervals greater than 30 minutes if `update()` is yet to be called.

Impact

Consumers receive inaccurate/stale prices, potentially leading to incorrect access gating.

Recommendation

Recompute the correct TWAP by extrapolating the missing values like in `currentCumulativePrices()`.

Status

Unresolved

[M-02] UniswapV2PriceFeed#update - Incorrect token assumption

Description

In the constructor and in `update()`, the `UniswapV2PriceFeed.sol` contract blindly uses `price0` from the Uniswap V2 pair reserves without verifying token ordering, leading to potentially incorrect prices. Uniswap V2 pairs sort tokens by address (token0 < token1), so `reserves0` and `reserves1` correspond to fixed positions regardless of the intended price direction.

Vulnerability Details

In Uniswap V2, `getReserves()` returns `(reserve0, reserve1, timestamp)` where `reserve0` always belongs to the token with the lower address (`token0`). If the contract assumes `price0 = reserve1 / reserve0` represents a specific token pair price (e.g., PetCoin/USDC), it fails when the pair is initialized with reversed ordering (e.g., USDC/PetCoin if USDC address < PetCoin address).

Proof of Concept

Consider USDC (address 0xA) and PetCoin (address 0xB where 0xA < 0xB):
- Pair: token0=USDC, token1=PetCoin; reserves: reserve0=100 USDC, reserve1=20000 PetCoin.
- Correct USDC price: reserve1/reserve0 = 200 (PetCoin per USDC).
- If the contract wants PetCoin price in USDC: should compute reserve0/reserve1 = 0.005.
- But if it assumes `price0` as desired without checking, and code uses reserve1/reserve0, it returns the wrong value.

Impact

Critical oracle mispricing leads to incorrect tier registering for users.

Recommendation

When reading the average price, pass the token address in parameters to read and return the correct value. You will have to edit the `update` function to include `price1` and change `getLatestPrice` to select the correct value like here:

https://github.com/Uniswap/v2-periphery/blob/master/contracts/examples/ExampleOracleSimple.sol#L59-L66

Low

[L-01] UniswapV2PriceFeed - Arithmetic operations should be unchecked in uniswap V2 TWAP oracle

Description

The `UniswapV2PriceFeed` TWAP oracle performs arithmetic on Uniswap V2 cumulative prices and timestamps using Solidity 0.8 checked math, which will revert on overflow instead of wrapping. Uniswap V2's oracle design relies on modular (wrapping) arithmetic for both cumulative prices and the 32-bit timestamp, so these reverts will eventually break the oracle and make it unusable.

Vulnerability Details

The contract is compiled with Solidity `^0.8.20`, so all arithmetic on `uint256` and `uint32` is checked by default and reverts on overflow/underflow.

However, Uniswap V2 design assumes that:
- `price{0,1}CumulativeLast` are unbounded accumulators that are allowed to overflow and wrap around.
- The pair timestamp is a `uint32` that also wraps roughly every 136 years, and differences are computed modulo \(2^{32}\).

In `update()`:

- `uint32 timeElapsed = blockTimestamp - blockTimestampLast;` will revert if `blockTimestamp < blockTimestampLast` after a `uint32` wrap.
- `priceAverageUQ112x112 = (price0Cumulative - priceCumulativeLast) / timeElapsed;` will revert if `price0Cumulative < priceCumulativeLast` due to cumulative overflow.

```
   function update() external {

        (uint256 price0Cumulative, , uint32 blockTimestamp) = currentCumulativePrices();

@>      uint32 timeElapsed = blockTimestamp - blockTimestampLast;
```

```
        emit DebugTimeElapsed(blockTimestamp, blockTimestampLast, timeElapsed);

        require(timeElapsed >= MIN_UPDATE_INTERVAL, "UniswapV2PriceFeed: TOO_SOON");

@>      priceAverageUQ112x112 = (price0Cumulative - priceCumulativeLast) / timeElapsed;

        priceCumulativeLast = price0Cumulative;

        blockTimestampLast = blockTimestamp;

        lastUpdateTimestamp = blockTimestamp;

    }
```

In `currentCumulativePrices()`:

- `uint32 timeElapsed = blockTimestamp - blockTimestampLastPair;` has the same `uint32` wrap risk.

- The manual accumulation logic is reimplementing Uniswap's oracle behavior, but without the required unchecked modular arithmetic used in the official `UniswapV2OracleLibrary`.

```
   function currentCumulativePrices() internal view returns (uint256 price0Cumulative,
uint256 price1Cumulative, uint32 blockTimestamp) {

        price0Cumulative = pair.price0CumulativeLast();

        price1Cumulative = pair.price1CumulativeLast();

        (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLastPair) =
pair.getReserves();

@>      blockTimestamp = uint32(block.timestamp);

        if (blockTimestampLastPair != blockTimestamp) {

@>          uint32 timeElapsed = blockTimestamp - blockTimestampLastPair;

            require(reserve0 > 0 && reserve1 > 0, "UniswapV2PriceFeed: NO_RESERVES");

@>          uint256 price0Delta = uint256(UQ112x112.uqdiv(UQ112x112.encode(reserve1),
reserve0)) * uint256(timeElapsed);

@>          uint256 price1Delta = uint256(UQ112x112.uqdiv(UQ112x112.encode(reserve0),
reserve1)) * uint256(timeElapsed);

@>          price0Cumulative += price0Delta;
```

```
@>          price1Cumulative += price1Delta;

        }

    }
```

The correct behavior is to allow these values to overflow and treat them using modular arithmetic, which in Solidity 0.8 requires explicit `unchecked` blocks or using the official Uniswap V2 oracle library that already handles this correctly.

Impact

Long-running deployments will eventually hit cumulative or timestamp overflows, causing `update()` (and any dependent logic) to revert and effectively bricking the TWAP oracle.

Recommendation

Replace the custom `currentCumulativePrices()` with Uniswap's `UniswapV2OracleLibrary.currentCumulativePrices` (https://github.com/Uniswap/v2-periphery/blob/master/contracts/libraries/UniswapV2OracleLibrary.sol) and ensure all arithmetic on cumulative prices and timestamps uses `unchecked` modular arithmetic as in the official library.

Status

Unresolved

Gas

[G-01] Dead code that should be removed

Description

Under this finding is a list of several code sections where dead code is identified. Refactoring the suggestions will improve code readability and reduce gas consumption upon deploying the smart contracts and running the impacted functions, resulting in significant gas cost reductions, especially if the contracts are deployed on the mainnet.

Vulnerability Details

In `AccessGating.sol#L105`:

```
   function getUserStakedOwed(address user) public view returns (uint256 stakedOwed) {
        address[] memory vaults = ICNUVaults(address(cnuToken)).getStakingVaultHistory();
        for (uint256 i = 0; i < vaults.length; i++) {
            address vault = vaults[i];
@>          if (vault == address(0) || vault.code.length == 0) continue;
            try IStakingVaultOwed(vault).getUserOwed(user) returns (uint256 owed) {
                stakedOwed += owed;
            } catch {
                // Ignore incompatible vaults
            }
        }
    }
```

The line `if (vault == address(0) || vault.code.length == 0) continue;` can be removed since the token `CNU.sol` ensures both of those conditions are true when adding a new vault.

In `CharityVault`:

- Line 13 defines a mapping `feeForwarders` that is not used and can be removed.
- The function `spend` can be simplified for the following reasons:

```
   function spend(address recipient, uint256 amount, string calldata memo) external
onlyOwner nonReentrant {

@>      require(recipient != address(0), "Invalid recipient"); // @audit `token.transfer`
already reverts for this reason

        require(amount > 0, "Amount must be > 0");

        IERC20 token = IERC20(cnuToken);

@>      uint256 balance = token.balanceOf(address(this)); // @audit Same as next line

@>      require(amount <= balance, "Insufficient balance"); // @audit `token.transfer`
already reverts for this reason

        require(token.transfer(recipient, amount), "Transfer failed");

        emit CharitySpent(recipient, amount, memo);

    }
```

In `TreasuryVault`, the functions `payClaim` and `withdraw` have a first line reverting if the `to` is `address(0)`. This check is already done during ERC20 transfer and the lines can be removed:

```
   function payClaim(address recipient, uint256 amount, string calldata memo) external
onlyOwner nonReentrant {

@>      require(recipient != address(0), "Invalid recipient"); // @audit `token.transfer`
already reverts for this reason

        require(amount > 0, "Amount must be > 0");

        require(cnuToken.transfer(recipient, amount), "Claim transfer failed");

        emit ClaimPaid(recipient, amount, memo);

    }
```

```
    function withdraw(address to, uint256 amount, string calldata memo) external
onlyOwner nonReentrant {
@>      require(to != address(0), "Invalid recipient"); // @audit `token.transfer`
already reverts for this reason
        require(amount > 0, "Amount must be > 0");
        require(cnuToken.transfer(to, amount), "Withdraw transfer failed");
        emit TreasuryWithdrawn(to, amount, memo);
    }
```

In `UniswapV2PriceFeed.sol`, the variable `lastUpdateTimestamp` is useless. Consider refactoring the code to remove this variable and use `blockTimestampLast` instead which will have the same value. Also, the event `DebugTimeElapse` sounds like it was created for debugging purposes and should be removed if it isn't useful off-chain.

Impact

Unoptimized code leads to reduced readability and increased gas consumption.

Recommendation

Refactor the code as recommended in the vulnerability details section.

Status

Unresolved

QA

[Q-01] StakingVault#getUserSummary - getUserSummary return values mismatch intended behavior

Description

The `getUserSummary` function returns `totalStakedAmount` (current active principal), `totalRewardsEarned` (total rewards to be earned for all active stakes), and `claimableNow` (principal + rewards for matured stakes only), but the notice implies it should show total staked amount and total rewards earned.

Vulnerability Details

The `getUserSummary` function returns two different values whose semantics do not match the notice description:

- `totalStakedAmount` is the current active principal across all unclaimed stakes (`userTotalStaked[user]`)

- `totalRewardsEarned` is actually the total rewards configured for all current active stakes (including future, not-yet-matured rewards via `userTotalRewards[user]`)

This means that instead of exposing "total staked and total rewards earned" as implied, the function conflates future rewards with already-earned ones.

Impact

Depending on what is the real desired use of the function, the impact may be either a bad natspec misleading API that causes frontends/dApps to display incorrect reward breakdowns, or a wrong value results for `totalStakedAmount` and `totalRewardsEarned`.

Recommendation

Rename the variables and edit the natspec if the code behaves as expected, or edit the function to return the actual total amount staked and rewards earned for a user.

Status

Unresolved

[Q-02] StakingVault#getTierParams - The staking reward percentage differs from the whitepaper

Description

The staking reward rates in `StakingVault.sol` do not match the percentages documented in the whitepaper, creating a discrepancy between user expectations and actual rewards.

Vulnerability Details

Whitepaper rates:

- 30 days = 2%

- 90 days = 5%

- 180 days = 10%

- 365 days = 15%

Implemented rates (in `getTierParams()`):

```
if (tier == Tier.THIRTY) return (30 days, 100);             // 1%

if (tier == Tier.NINETY) return (90 days, 300);             // 3%

if (tier == Tier.ONE_EIGHTY) return (180 days, 700);        // 7%

if (tier == Tier.THREE_SIXTY_FIVE) return (365 days, 1500); // 15%
```

The code implementation appears correct as the whitepaper values show illogical progression (90-day and 180-day tiers have identical rewards, and 365-day rewards are lower than shorter tiers).

Impact

Users relying on whitepaper documentation will receive lower rewards than expected for 30-day (50% less), 90-day (40% less), and 180-day (30% less) stakes. This creates trust issues and potential user dissatisfaction despite the implementation being economically sound.

Recommendation

Update the whitepaper to reflect the actual implemented rates: 1%, 3%, 7%, and 15% for the respective tiers.

Status

Unresolved

[Q-03] UniswapV2PriceFeed#constructor - Unreachable try-catch blocks in UniswapV2PriceFeed constructor

Description

The `UniswapV2PriceFeed` contract constructor contains try-catch blocks that wrap external calls to the Uniswap V2 pair contract. Under the expected deployment scenario where the Uniswap pair is deployed before the price feed, these catch blocks will never execute, resulting in dead code that serves no functional purpose.

Vulnerability Details

In the expected scenario where the price feed is deployed after the uniswap pair, the static calls to `pair.price0CumulativeLast()` and `pair.getReserves()` will never revert. For this reason, these lines should be removed.

In the case where the price feed will be deployed before the pair, and the address passed in the constructor is the pre-computed address of the pair deployment, then the try-catch blocks are indeed needed, but the price feed will report a price of 0 starting from `block.timestamp`, which will result in an incorrect value of the price for the first update. If this second scenario is desired, consider updating the code to start saving values for the price feed after the pair is deployed.

Impact

Dead code or incorrect price feed, depending on the deployment order.

Recommendation

Remove the try catch blocks and deploy the pair before the price feed

```
constructor(address _pair) Ownable(msg.sender) {

    pair = IUniswapV2Pair(_pair);

    priceCumulativeLast = pair.price0CumulativeLast();

    (,,blockTimestampLast) = pair.getReserves();

}
```

Status

Unresolved

[Q-04] CNU#function - Users can exceed max CNU by interacting with an exempted address

Description

In `CNU.sol`, transfers to addresses exempted from fees or limits also exempt the sender by default for that transfer. This allows non-exempt users to bypass max limits by sending to an exempted address, such as a staking vault.

Vulnerability Details

The contract logic incorrectly applies exemptions bidirectionally: if the recipient (`to`) is whitelisted (e.g., no fees or limits), the sender (`from`) skips checks for fees and limits. Regular users will bypass their own exemptions by interacting with exempted users or contracts, like staking vaults.

Proof of Concept

This test shows how a user can exceed the max balance. You can assume `await token.transfer(user1, maxWalletSize);` is actually the user receiving tokens through an exchange or from another user.

```
    it("Allows user to have more than maxWalletSize", async () => {
      const maxWalletSize = await token.maxWalletSize();
      const rewardReserve = maxWalletSize * 100n / 10000n;
      await token.transfer(stakingVault, rewardReserve);
      await token.transfer(user1, maxWalletSize);
      await token.connect(user1).approve(stakingVault, maxWalletSize);
      await stakingVault.connect(user1).stake(maxWalletSize, 1); // Tier 1 = 30 days
      await token.transfer(user1, maxWalletSize);

      console.log(`User1 balance before earlyWithdraw: ${ethers.formatUnits(await
token.balanceOf(user1), 18)} CNU`);
      await stakingVault.connect(user1).earlyWithdraw(0);
```

```
      console.log(`User1 balance after earlyWithdraw:  ${ethers.formatUnits(await
token.balanceOf(user1), 18)} CNU`);

      console.log(`Max wallet size:
${ethers.formatUnits(maxWalletSize, 18)} CNU`);

      console.log(`Difference:                          ${ethers.formatUnits(await
token.balanceOf(user1) - maxWalletSize, 18)} CNU over limit`);

    });
```

Impact

Non-exempt users can bypass max limits and fees.

Recommendation

If bypassing fees or max limits this way is an issue, consider refactoring the code to check fee exemption from the sender only, and checking the limit exemption from the receiver only:

```
 function _isFeeExempt(address from, address to) internal view returns (bool) {

      return

          isExcludedFromFees[from]

-         || isExcludedFromFees[to]

          || from == address(0)

          || to == address(0);

  }


- function _isLimitExempt(address from, address to) internal view returns (bool) {

+ function _isLimitExempt(address to) internal view returns (bool) {

-      return isExcludedFromLimits[from]

-         || isExcludedFromLimits[to];

+      return isExcludedFromLimits[to];

  }
```

This way, a user claiming from the staking vault will not be able to exceed their max wallet holdings. If the fee exemption is also necessary when staking, then more refactoring should be needed to enable it.

Status

Unresolved

Centralisation

The Petcoin AI project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Petcoin AI project seems to have a sound and well-tested code base, however, our findings need to be resolved to achieve full security. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au