hashlock.

2026

Security Audit

ENNI (DeFi)

March 2026

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

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 ENNI 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

ENNI is a decentralized, immutable lending and stablecoin protocol on Ethereum, designed without governance, admin keys, or upgradeability. Once deployed, the smart contracts operate autonomously as the final authority.

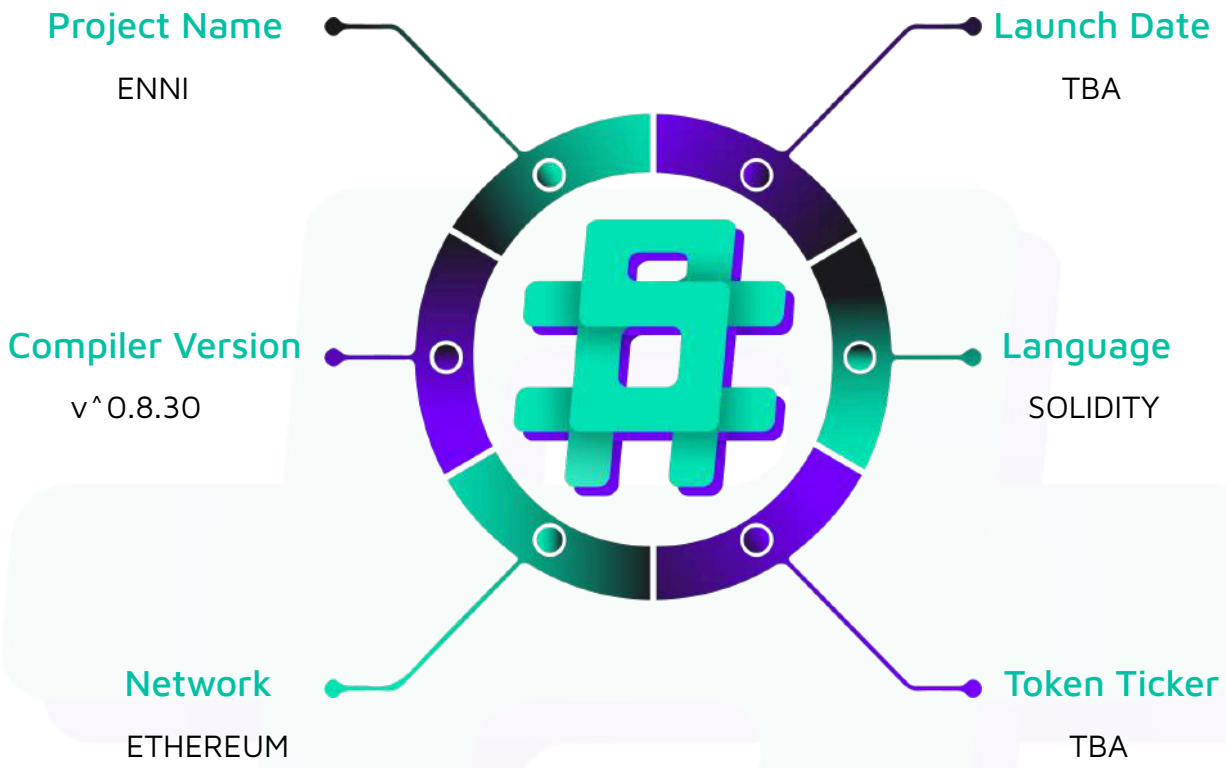
The protocol follows an overcollateralized CDP model, allowing users to mint or borrow stablecoins against collateral (e.g., WETH), with features such as 0% interest borrowing and optional direct minting backed by stable assets.

ENNI is currency-agnostic: each stablecoin is an independent deployment of the same core contracts, with its own collateral pool and oracle setup. New currencies can be added permissionlessly without requiring upgrades or governance.

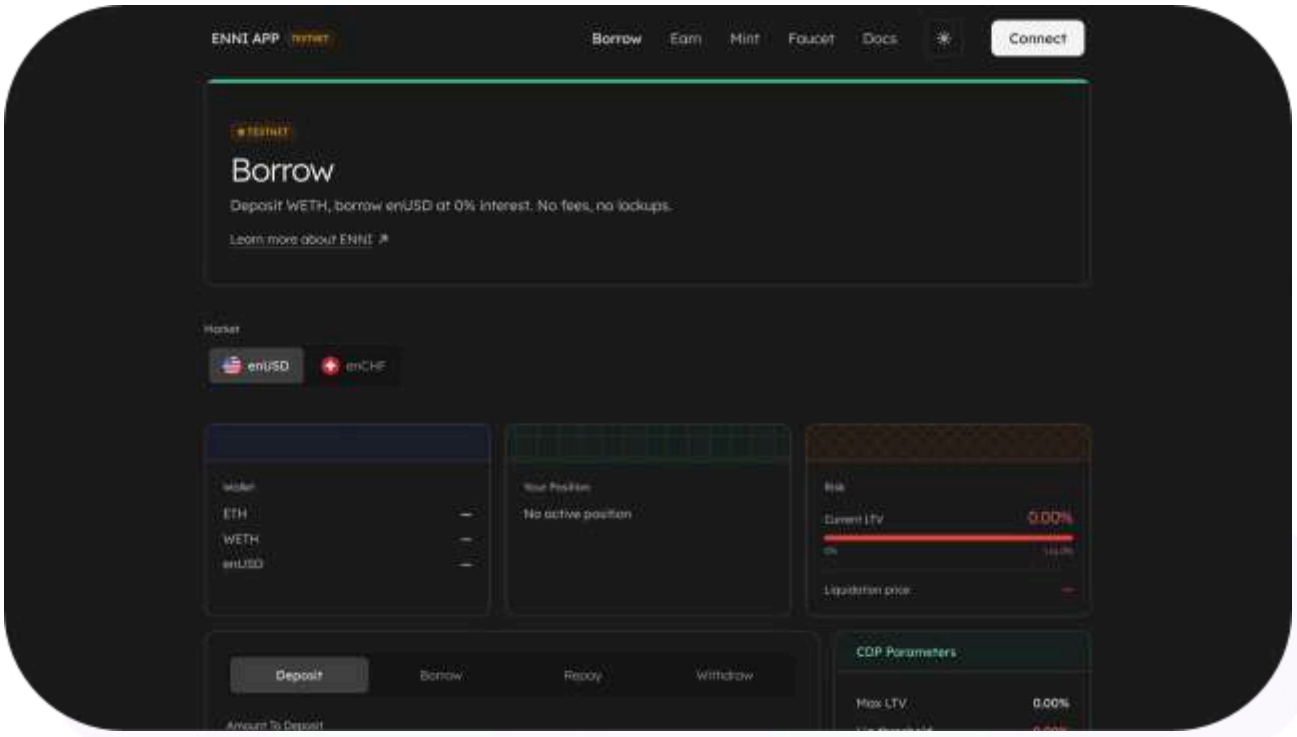
Project Name: ENNI
Project Type: DeFi
Compiler Version: ^0.8.30
Website: https://testnet.enni.ch/
Logo:

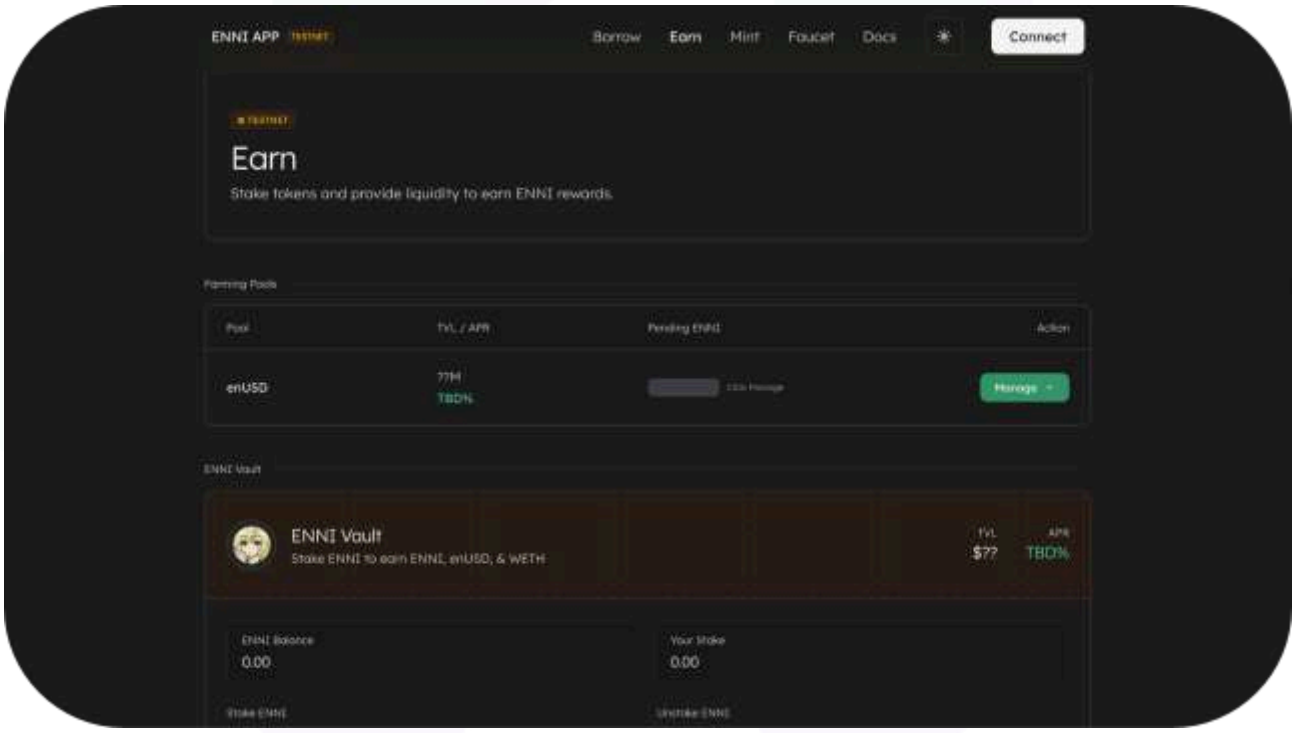
ENNI APP

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the ENNI 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	**ENNI Smart Contracts**
Network	**Ethereum**
Language	**Solidity**
Audit Date	**March, 2026**
Contract 1	EnniCDP.sol
Contract 2	EnniDirectMint.sol
Contract 3	EnniMasterChef.sol
Contract 4	EnniOracle.sol
Contract 5	EnniToken.sol
Contract 6	EnniVault.sol
Audited GitHub Commit Hash	ad2289ceed6485ba2f90d3f139e5d1be7c3c935e
Fix Review GitHub Commit Hash	2ee7ea3bc9eb71124327b909f14998d4e06ed699

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. 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 initially identified have now been resolved and acknowledged.

Hashlock found:

1 High severity vulnerability

9 Medium severity vulnerabilities

2 Low severity vulnerabilities

1 QA

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

EnniCDP.sol
Allows users to open and manage collateralized debt positions (WETH → stablecoin).
- Users can:
  - Open a position by depositing WETH collateral
  - Deposit more collateral
  - Withdraw collateral (if LTV stays within limits)
  - Borrow stablecoin (enUSD) against collateral
  - Repay debt (with minimum debt rules)
  - Claim premium credit earned by third-party buyouts
  - Close the position (withdraw collateral + premium once debt is repaid)
- Third parties can:
  - Buy out (repay part of another user's debt for a premium, if not liquidatable)
  - Liquidate under-collateralized positions (seize collateral, donate portion to rewards vault)
- System/Oracle:
  - Uses an external oracle for ETH/fiat price checks (stale price protections)
	Contract achieves this functionality.

EnniDirectMint.sol A simple on-ramp/off-ramp between USDC/USDT and enUSD.	**Contract achieves this functionality.**

- Users can:
  - Mint enUSD 1:1 by depositing USDC or USDT
  - Redeem enUSD 1:1 back to USDC or USDT (with a 0.5% fee)
- System:
- Fees are donated to a rewards vault	

EnniMasterChef.sol
A staking rewards contract (ENNI emissions to LP stakers).
- The owner can:
  - Add up to 8 LP pools
  - Set pool allocation points
- Users can:
  - Deposit LP tokens to earn ENNI
  - Withdraw LP tokens
  - Harvest earned ENNI rewards
  - Emergency withdraw (forfeit rewards)
- System:
  - Uses a time-based emission schedule (30 years, phased emission rates)
  - Caps total minting via MAX_CHEF_MINT
  - Tracks rewards per share and enforces global emission limits
	Contract achieves this functionality.

EnniOracle.sol
A price oracle wrapper that chooses the best available ETH/fiat price feed and caches a "last good" value.
- Provides:
  - peekPrice() and peekPriceWithTimestamp()
  - fetchPrice() to update cached price
	Contract achieves this functionality.

- Sources:
  - Primary: Chainlink ETH/USD
  - Fallback: Chronicle ETH/USD
  - Optional translator feed (e.g., JPY/USD) for non-USD fiat
- Behavior:
  - Returns cached last-good price if feeds are stale/unavailable	

EnniToken.sol
A basic ERC-20 + permit token with dual minter control and capped minting.
- The owner can:
  - Change minter1 / minter2
  - Transfer/renounce ownership
- Minters can:
  - Mint tokens (up to maxMintable, if set)
- Anyone can:
  - Burn their own tokens
  - Burn from others with allowance
	Contract achieves this functionality.

EnniVault.sol
A vault that pools ENNI and distributes rewards (ENNI, WETH, enUSD) pro rata to depositors.
- Users can:
  - Deposit ENNI (stakes it into MasterChef behind the scenes)
  - Withdraw ENNI (redeems from MasterChef)
  - Claim accumulated rewards (ENNI/WETH/enUSD)
- Donors can:
  - Donate WETH or enUSD to be distributed
	Contract achieves this functionality.

to stakers
- System:
  - Auto-harvests from MasterChef
  - Uses share accounting and "queued" reward handling when no stakers exist	

Code Quality

This audit scope involves the smart contracts of the ENNI 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 was 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 ENNI 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-severity issues should be resolved before deployment. While they do not typically lead to a complete loss of funds, they may result in partial loss of funds or unintended behavior under certain conditions.
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

High

[H-01] EnniCDP#liquidate - MIN_DEBT rule on liquidation creates permanently unliquidatable bad debt for small positions

Description

The `RemainingDebtRule` check at line 398 applies identically to both voluntary repayment and liquidation. For a position with debt near `MIN_DEBT` (e.g., 401e6) that crosses the 88% liquidation threshold, partial liquidation is blocked for nearly all amounts, and full liquidation becomes economically irrational once the position is sufficiently underwater. The protocol has no bad-debt socialization mechanism, so the position accumulates bad debt permanently.

Vulnerability Details

In `liquidate`, the remaining debt check enforces the same `MIN_DEBT` floor as borrower-side repayment:

```
function liquidate(address owner, uint256 repayAmount) external nonReentrant {

    // ...

    uint256 remaining = p.debt - repayAmount;

    if (remaining != 0 && remaining < MIN_DEBT) revert RemainingDebtRule();



    uint256 collateralSeized = Math.mulDiv(p.collateral, repayAmount, p.debt);

    // ...

    uint256 donation = Math.mulDiv(collateralSeized, DONATION_BPS, BPS); // 3%

    uint256 toLiquidator = collateralSeized - donation;

    // ...
```

```
}
```

For a position with debt = 401e6:

- Any partial `repayAmount` in the range [2, 1e6] leaves remaining in (0, `MIN_DEBT`) → reverts.
- `repayAmount` = 1 leaves remaining = 400,999,999 >= `MIN_DEBT` → passes, but yields ~0 collateral (economically useless).
- `repayAmount` = 401e6 (full) leaves remaining = 0 → passes the check.

However, when the position is deeply underwater (e.g., LTV > 97%), full liquidation becomes unprofitable:

```
Position: debt = 401e6, collateral value = $410 at 97.8% LTV
Full liquidation:
  collateralSeized = all collateral ≈ $410 worth of WETH
  donation = 3% × $410 = $12.30
  toLiquidator = $410 - $12.30 = $397.70 worth of WETH
  Liquidator pays: 401 stablecoins ($401)
  Liquidator receives: $397.70 WETH
  NET LOSS: $3.30 → no rational liquidator acts
```

Impact

Small positions near `MIN_DEBT` that become underwater beyond ~97% LTV are permanently unliquidatable. No rational liquidator will take a loss; partial liquidation is blocked by the `RemainingDebtRule`, and the protocol has no bad-debt socialization, auction mechanism, or backstop. These positions accumulate bad debt indefinitely, creating unbacked stablecoin supply that erodes the peg. The issue is amplified because MIN_DEBT = 400e6 is the minimum; many positions will cluster near this floor.

Recommendation

We recommend exempting liquidation calls from the `MIN_DEBT` remaining-debt constraint, allowing liquidators to partially clear any amount. This maximizes bad-debt resolution flexibility:

```
-    uint256 remaining = p.debt - repayAmount;
```

```
-     if (remaining != 0 && remaining < MIN_DEBT) revert RemainingDebtRule();
+     uint256 remaining = p.debt - repayAmount;
+     // MIN_DEBT is a borrower-side rule; liquidators must be free to
+     // partially clear any amount to maximize bad-debt resolution
```

Status

Resolved

Medium

[M-01] EnniMasterChef#_safeEnniTransfer - Silent underpayment permanently loses user rewards

Description

The `_safeEnniTransfer` function silently truncates reward payouts to the contract's available ENNI balance without reverting or emitting an event. Since rewardDebt is updated to reflect the full (untruncated) amount, the shortfall is permanently lost; users can never reclaim the difference.

Vulnerability Details

The `_safeEnniTransfer` function caps transfers at the contract's balance:

```
function _safeEnniTransfer(address to, uint256 amount) internal {

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

    if (amount > bal) amount = bal;

    if (amount > 0) IERC20(address(enni)).safeTransfer(to, amount);

}
```

After calling `_safeEnniTransfer`, the calling functions update `rewardDebt` to the full accumulated value, not the actually transferred amount. For example, in `withdraw`:

```
uint256 pending = accumulated > u.rewardDebt ? (accumulated - u.rewardDebt) : 0;

if (pending > 0) {

    _safeEnniTransfer(msg.sender, pending); // may send less than `pending`

}

// ...

u.rewardDebt = Math.mulDiv(u.amount, pool.accEnniPerShare, ACC_PRECISION); // full amount
```

Impact

This creates a first-come, first-served race condition during reward claiming. Early harvesters receive full payouts while late harvesters silently lose part or all of their rewards with no indication of the shortfall. The issue is especially dangerous near the `MAX_CHEF_MINT` cap, where the combination with issue H-02 can leave the contract with insufficient ENNI for pending claims.

Recommendation

We recommend reverting the transaction if the contract's balance is insufficient to cover the full reward amount. This ensures users are aware of the shortfall and can retry after more ENNI is minted. If silent truncation is desired for emergencies, emit an event logging the shortfall so users and monitoring tools can detect underpayments.

Status

Acknowledged

[M-02] EnniMasterChef#pendingEnni - View function overstates claimable rewards near mint cap

Description

The `pendingEnni` view function provides an inaccurate approximation of claimable rewards when mintedByChef is near `MAX_CHEF_MINT`, because it does not account for other pools consuming the remaining budget during massUpdatePools execution.

Vulnerability Details

In `_pendingEnni`, the clamping logic uses a global approximation:

```
if (mintedByChef >= MAX_CHEF_MINT) r = 0;

else {

    uint256 remaining = MAX_CHEF_MINT - mintedByChef;

    if (r > remaining) r = remaining;

}
```

This clamping assumes the entire remaining budget is available for this single pool. In reality, when harvest, deposit, or withdraw is called, `_updatePool` triggers `massUpdatePools` (or the single pool update), and other pools may consume part of the remaining budget first (per H-02).

Impact

Frontend interfaces and external contracts relying on pendingEnni may display inflated reward amounts. Users may spend gas to harvest based on these inflated estimates only to receive significantly less (or nothing), especially if the transaction triggers massUpdatePools and lower-pid pools consume the remaining budget.

Recommendation

We recommend documenting this limitation clearly in the function's natspec. If accuracy is required, consider a view function that simulates the full `massUpdatePools` sequence to produce a more realistic estimate.

Status

Acknowledged

[M-03] EnniMasterChef#emergencyWithdraw - Forfeited rewards are permanently locked in the contract

Description

When a user calls `emergencyWithdraw`, their unclaimed ENNI rewards (already minted and held by the contract) are permanently locked. There is no sweep function or redistribution mechanism to recover these funds.

Vulnerability Details

The `emergencyWithdraw` function zeroes the user's accounting without harvesting or updating the pool:

```
function emergencyWithdraw(uint256 pid) external nonReentrant {

    // ...

    uint256 amt = u.amount;

    require(amt > 0, "nothing");



    u.amount = 0;

    u.rewardDebt = 0;



    pool.totalStaked -= amt;

    pool.lpToken.safeTransfer(msg.sender, amt);



    emit EmergencyWithdraw(msg.sender, pid, amt);

}
```

The forfeited ENNI remains in the contract's balance. While `_safeEnniTransfer` could theoretically use this surplus to cover future shortfalls (M-01), this creates a non-deterministic reward distribution where some users unpredictably receive rewards funded by other users' forfeitures.

Impact

ENNI tokens that were already minted and allocated become permanently unrecoverable by the protocol. Over the 30-year emission schedule, accumulated forfeitures could represent a meaningful amount of locked value. Additionally, the interaction with `_safeEnniTransfer` creates unpredictable reward flows.

Recommendation

We recommend adding an owner-callable function to sweep excess ENNI (above total pending obligations) from the contract, or implementing a mechanism to redistribute forfeited rewards back into the emission pool.

Status

Acknowledged

[M-04] EnniCDP#liquidate - Donation failure permanently locks WETH in immutable CDP contract

Description

When `rewardsVault.donateWETH(donation)` reverts inside the try/catch, the donation WETH is silently orphaned, already deducted from `p.collateral` but never transferred to the vault or the liquidator. The immutable CDP has no sweep function, no admin recovery, and no mechanism to reclaim these tokens.

Vulnerability Details

In `liquidate`, the donation flow deducts from the position before attempting the external call:

```
uint256 collateralSeized = Math.mulDiv(p.collateral, repayAmount, p.debt);

if (collateralSeized == 0) revert OracleBad();



uint256 donation = Math.mulDiv(collateralSeized, DONATION_BPS, BPS); // 3%

uint256 toLiquidator = collateralSeized - donation;                  // 97%



stableToken.safeTransferFrom(msg.sender, address(this), repayAmount);

stable.burn(repayAmount);



p.debt = remaining;

p.collateral -= collateralSeized; // full seized amount removed from position



// ---- donation is non-blocking ----

if (donation > 0) {

    try rewardsVault.donateWETH(donation) {} catch {} // if reverts → donation stays in
contract

}
```

```
weth.safeTransfer(msg.sender, toLiquidator); // only 97% leaves
```

When `donateWETH` reverts (vault paused, reentrancy guard triggered, gas limit, etc.):

1. collateralSeized is fully removed from the position (line 410)
2. toLiquidator (97%) is sent to the caller (line 417)
3. donation (3%) remains in the CDP contract balance
4. No state variable tracks this orphaned WETH
5. The contract is immutable with no sweep() or admin recovery function

Impact

Every failed donation permanently locks 3% of seized collateral in the CDP contract. If the rewards vault experiences any downtime or misconfiguration, WETH accumulates in the CDP with no recovery path. Over time, this represents a growing and irrecoverable loss of protocol value. For a $10M total liquidation volume, up to $300K could be permanently locked.

Recommendation

We recommend redirecting the donation to the liquidator when the vault call fails:

```
if (donation > 0) {
-     try rewardsVault.donateWETH(donation) {} catch {}
+     try rewardsVault.donateWETH(donation) {} catch {
+         toLiquidator += donation;
+     }
  }
```

Status

Resolved

[M-05] EnniCDP#ltvBps - isLiquidatable view returns false for positions with dust collateral and meaningful debt

Description

When `_ethToFiatWithPrice(p.collateral, price)` truncates to zero due to dust-level `collateral`, `ltvBps()` returns `(0, false)`, and `isLiquidatable()` returns `false`. However, the `liquidate()` function correctly identifies these same positions as liquidatable by assigning `ltvBefore = type(uint256).max`. This inconsistency causes off-chain liquidation bots to skip bad-debt positions.

Vulnerability Details

The view path and the state-changing path handle zero-collateral-fiat differently:

View path (`ltvBps` → `isLiquidatable`):

```
function ltvBps(address owner) public view returns (uint256 ltv, bool ok) {

    // ...

    uint256 collateralFiat = _ethToFiatWithPrice(p.collateral, price);

    if (collateralFiat == 0) return (0, false); // ← returns ok=false

    return (Math.mulDiv(p.debt, BPS, collateralFiat), true);

}



function isLiquidatable(address owner) external view returns (bool) {

    (uint256 ltv, bool ok) = ltvBps(owner);

    if (!ok) return false; // ← returns NOT liquidatable

    return ltv >= LIQ_LTV_BPS;

}
```

State-changing path (`liquidate`):

```
function liquidate(address owner, uint256 repayAmount) external nonReentrant {

    // ...
```

```
    uint256 collateralFiatBefore = _ethToFiatWithPrice(p.collateral, price);
    uint256 ltvBefore = collateralFiatBefore == 0
        ? type(uint256).max    // ← correctly treats as maximally underwater
        : Math.mulDiv(p.debt, BPS, collateralFiatBefore);
    if (ltvBefore < LIQ_LTV_BPS) revert NotLiquidatable(); // passes (max >= 8800)
    // ...
}
```

The dust collateral threshold at $2000/ETH: `_ethToFiatWithPrice` truncates to zero when `collateral * price / 1e18 / 1e12 == 0`, i.e., `collateral < 5e8 wei` (~$0.000001). These positions have near-zero collateral backing real debt, exactly the positions that most urgently need liquidation.

Impact

Off-chain liquidation bots and frontend interfaces relying on `isLiquidatable()` will skip positions with dust collateral and meaningful debt. These positions represent bad debt (unbacked stablecoin) that the protocol's own view function reports as "not liquidatable," even though the `liquidate()` function would accept the call. This delays or prevents bad-debt cleanup, degrading peg stability.

Recommendation

We recommend aligning the view function behavior with the state-changing function:

```
uint256 collateralFiat = _ethToFiatWithPrice(p.collateral, price);
- if (collateralFiat == 0) return (0, false);
+ if (collateralFiat == 0) return (type(uint256).max, true);
```

Status

Resolved

[M-06] EnniCDP#buyout - No slippage protection for buyout callers

Description

The `buyout()` function computes `collateralOut = _fiatToEthWithPrice(repayAmount, price)` at execution time with no caller-supplied minimum. Between transaction submission and inclusion, a legitimate Chainlink/Chronicle heartbeat update can change the price, causing the buyer to receive less WETH while paying the same repayAmount + premium in stablecoins.

Vulnerability Details

The buyout function determines collateral output solely from the oracle price at execution time:

```
function buyout(address owner, uint256 repayAmount) external nonReentrant {

    // ...

    uint256 price = _requireFreshPrice(); // price at execution, not submission



    // ... premium calculation ...



    uint256 collateralOut = _fiatToEthWithPrice(repayAmount, price);

    if (collateralOut == 0) revert OracleBad();

    if (collateralOut > p.collateral) revert InsufficientCollateral();

    // NO CHECK: if (collateralOut < minCollateralOut) revert ...



    stableToken.safeTransferFrom(msg.sender, address(this), totalPay);

    // ...

    weth.safeTransfer(msg.sender, collateralOut);

}
```

`_fiatToEthWithPrice` returns `mulDiv(fiatAmount, 1e30, price)`; a higher ETH price means less WETH per stablecoin. The buyer has no parameter to set a minimum acceptable collateral amount.

Concrete scenario:

- Buyer submits tx when ETH = $2,000: expects 0.5 ETH for 1,000 enUSD repayment
- Chainlink heartbeat fires between submission and inclusion: ETH = $2,100 (+5%)
- Execution: `collateralOut = mulDiv(1000e6, 1e30, 2100e18)` = 0.4762 ETH
- Buyer receives 0.0238 ETH less (~$50 loss on a $1,000 buyout) with no way to reject

Chainlink ETH/USD has a 0.5% deviation trigger and a 1-hour heartbeat. A 5% move during high volatility (2-3 heartbeat updates) is realistic.

Impact

Buyout callers are exposed to unprotected price slippage. During volatile market conditions, legitimate oracle updates between transaction submission and execution can result in the buyer receiving materially less collateral than expected. Unlike AMM swaps, which universally offer minAmountOut parameters, this function provides no protection mechanism.

Recommendation

We recommend adding a `minCollateralOut` parameter:

```
- function buyout(address owner, uint256 repayAmount) external nonReentrant {

+ function buyout(address owner, uint256 repayAmount, uint256 minCollateralOut) external
nonReentrant {

      // ...

      uint256 collateralOut = _fiatToEthWithPrice(repayAmount, price);

      if (collateralOut == 0) revert OracleBad();

      if (collateralOut > p.collateral) revert InsufficientCollateral();

+     if (collateralOut < minCollateralOut) revert InsufficientCollateral();
```

Status

Acknowledged

[M-07] EnniOracleV1#peekPriceWithTimestamp - Stale cached price accepted for up to 24 hours when both feeds fail

Description

When both Chainlink and Chronicle feeds are unavailable, `peekPriceWithTimestamp()` falls back to (`lastGoodPrice`, `lastGoodUpdatedAt`). The CDP's `ORACLE_MAX_AGE = 24 hours` check allows this cached price through for up to 24 hours after both feeds go down, enabling all CDP operations at a potentially significantly stale price during volatile markets.

Vulnerability Details

The oracle's fallback path returns the last cached price without any indication that it is a fallback:

```
function peekPriceWithTimestamp() external view returns (uint256 price, uint256 updatedAt) {
    (bool ok, uint256 p, uint256 ts, ) = _readBest();
    if (ok) return (p, ts);

    uint256 cached = lastGoodPrice;        // ← returns stale cached price
    uint256 cachedTs = lastGoodUpdatedAt;   // ← with original timestamp
    if (cached == 0) revert OracleUnavailable();
     return (cached, cachedTs);                // ← no way for caller to know this is a fallback
}
```

The CDP's `_readPrice` then checks freshness against this original timestamp:

```
function _readPrice() internal view returns (uint256 price, uint256 updatedAt, bool isFresh) {
    (price, updatedAt) = oracle.peekPriceWithTimestamp();
    // ...
```

```
    isFresh = (block.timestamp - updatedAt) <= ORACLE_MAX_AGE; // 24 hours

}
```

Combined timeline:

```
T=0h:  Both feeds working. lastGoodPrice = $2500, lastGoodUpdatedAt = T=0
T=1h:  Both feeds go down (stale or reverting)
T=5h:  User calls borrow()
       → oracle returns ($2500, T=0)
       → CDP: (T=5h - T=0) = 5h <= 24h → fresh ✓
       → User borrows against $2500 valuation
       → But real ETH price may now be $1800 (28% crash)

T=23h: Still accepted. Real price could be anywhere.
T=25h: Finally rejected. 24+ hours of stale price exposure.
```

Impact

During a dual-feed outage (Chainlink + Chronicle), the protocol continues operating on a price that can be up to 24 hours old. In volatile markets, ETH can move 20-30% in 24 hours. Users can borrow against inflated collateral valuations, and liquidatable positions remain protected by the stale high price. The 24-hour window is especially dangerous because it exceeds Chainlink's heartbeat (1 hour) by 24x, meaning the price is far more stale than any single-feed staleness check would permit.

Recommendation

We recommend reverting when both feeds are unavailable instead of falling back to a cached price, or significantly reducing `ORACLE_MAX_AGE`:

```
function peekPriceWithTimestamp() external view returns (uint256 price, uint256
updatedAt) {

      (bool ok, uint256 p, uint256 ts, ) = _readBest();

      if (ok) return (p, ts);



-     uint256 cached = lastGoodPrice;
```

```
-     uint256 cachedTs = lastGoodUpdatedAt;
-     if (cached == 0) revert OracleUnavailable();
-     return (cached, cachedTs);
+     revert OracleUnavailable();
  }
```

If a cached fallback is desired for liveness, reduce `ORACLE_MAX_AGE` to 2-4 hours to limit stale-price exposure.

Status

Acknowledged

[M-08] EnniDirectMint#_redeem - Fee donation failure permanently locks enUSD in the immutable DirectMint contract

Description

The `_redeem` function donates the redemption fee to the rewards vault via try `rewardsVault.donateEnUSD(fee) {} catch {}`. When the donation reverts, the fee amount of `enUSD` is neither donated nor burned; it remains permanently stranded in the `DirectMint` contract with no recovery mechanism. This is the same empty-catch pattern as M-04, but affects enUSD in a different contract.

Vulnerability Details

In `_redeem`, the fee is separated from the redeemed amount, but only the net is burned:

```
function _redeem(IERC20Metadata stable, uint256 amount) internal {
    require(amount > 0, "Zero amount");

    uint256 fee = (amount * REDEEM_FEE_BPS) / BPS;  // 0.5% fee
    uint256 net = amount - fee;

    require(stable.balanceOf(address(this)) >= net, "Insufficient liquidity");

    IERC20Metadata(address(enUSD)).safeTransferFrom(msg.sender, address(this), amount);

    // Donate fee to vault, non-blocking to guarantee liveness
    if (fee > 0) {
        try rewardsVault.donateEnUSD(fee) {} catch {} // ← if reverts, fee stays here
    }

    enUSD.burn(net);          // ← only burns net, NOT fee

    stable.safeTransfer(msg.sender, net);
```

When donateEnUSD reverts, `the` contract received amount enUSD from the user (line 115)

1. fee donation fails silently (line 119)
2. net is burned (line 122), but the fee is not

3. fee enUSD remains in the DirectMint contract balance
4. No sweep function, no admin recovery, contract is immutable

For a 1000 enUSD redemption: fee = 5 enUSD stranded per failed donation. Over 1000 such events: 5000 enUSD ($5000) permanently locked.

Impact

Every failed fee donation permanently locks 0.5% of the redeemed enUSD in the DirectMint contract. The stranded enUSD inflates the total supply (not burned) while being economically dead (not circulating). This creates a persistent, growing discrepancy between enUSD total supply and actual backing, degrading peg integrity over time.

Recommendation

We recommend burning the fee when the donation fails, ensuring no enUSD is orphaned:

```
if (fee > 0) {
-     try rewardsVault.donateEnUSD(fee) {} catch {}
+     try rewardsVault.donateEnUSD(fee) {} catch {
+         enUSD.burn(fee);
+     }
  }
  enUSD.burn(net);
```

Status

Resolved

[M-09] EnniMasterChef#_updatePool - Empty pools with non-zero allocPoint permanently waste emissions

Description

When a pool has a non-zero allocPoint but zero `totalStaked`, the pool's share of global emissions is permanently lost. The `lastRewardTime` advances, but no ENNI is minted, and the lost emissions are not redistributed to other active pools.

Vulnerability Details

In `_updatePool`, when `lpSupply == 0`, the function simply advances `lastRewardTime` and returns without minting:

```
function _updatePool(uint256 pid) internal {

    PoolInfo storage pool = poolInfo[pid];



    uint64 toTime = _clampToGlobal(uint64(block.timestamp));

    if (toTime <= pool.lastRewardTime) return;



    uint256 lpSupply = pool.totalStaked;

    if (lpSupply == 0 || pool.allocPoint == 0 || totalAllocPoint == 0) {

        pool.lastRewardTime = toTime; // time advances, nothing minted

        return;

    }

    // ...

}
```

Meanwhile, `_poolRewardBetween` still divides global emissions by `totalAllocPoint`, which includes the empty pool's allocation:

```
function _poolRewardBetween(uint256 pid, uint64 from, uint64 to) internal view returns
(uint256) {
```

```
    // ...

    uint256 emission = _globalEmissionBetween(from, to);

    // ...

    return Math.mulDiv(emission, poolAlloc, _totalAlloc);

}
```

This means active pools receive a reduced share because totalAllocPoint includes the empty pool, but the empty pool's share is never minted; it is permanently lost.

Impact

If a pool with 50% allocPoint has no stakers for one month during Phase 1, approximately 83,333 ENNI (~166,666 ENNI/month * 50%) are permanently unearnable. This directly dilutes rewards for all active stakers across the protocol. The issue is amplified when addPool is called before the LP token is available on a DEX, the new pool immediately dilutes all existing pools, and its emission share vanishes until someone deposits.

Recommendation

We recommend setting the allocPoint to zero for pools that are not yet active or have no stakers, or implementing a mechanism that redistributes the empty pool's emission share proportionally to pools that have active stakers.

Status

Acknowledged

Low

[L-01] EnniCDP#deposit - Ghost position via premiumCredit bypasses open() guard and skips PositionOpened event

Description

After a full buyout leaves an owner with debt=0, collateral=0, premiumCredit>0, the `_hasPosition()` check returns true (due to `premiumCredit`), blocking `open()` with `PositionAlreadyExists`. However, `deposit()` passes the same check and allows adding collateral, after which `borrow()` reactivates a full position, all without emitting a PositionOpened event.

Impact

Off-chain systems (liquidation bots, dashboards, analytics) that track positions via PositionOpened events will not know this position exists. While the on-chain functionality is correct and no funds are at risk, the missing event creates a blind spot in position monitoring. This is particularly concerning for liquidation bots that may miss positions needing liquidation.

Recommendation

We recommend changing `deposit()` to check for an active position based on collateral or debt rather than the broad `_hasPosition`.

Status

Resolved

[L-02] EnniToken#transferOwnership- Single-step ownership transfer risks permanent loss of admin control

Description

The `transferOwnership` function immediately transfers the owner role to a new address in a single step with no confirmation from the recipient. If the new address is incorrect (typo, wrong checksum, contract without proper handling), admin control over minter management is permanently and irrecoverably lost.

Impact

If ownership is transferred to an incorrect address, the protocol permanently loses the ability to update minter roles. For the ENNI governance token (which has a hard supply cap and dual-minter system), this means the minter configuration is frozen forever. While the token continues to function, any future need to rotate minters (compromise, migration, new contracts) becomes impossible.

Recommendation

We recommend implementing a two-step ownership transfer pattern.

Status

Acknowledged

QA

[Q-01] EnniToken#constructor - No validation on zero decimals allows deployment of an unusable token

Description

The EnniToken constructor accepts `decimals_ = 0` without validation. While this is a deployment-time configuration concern rather than a runtime vulnerability, deploying a token with zero decimals would make it incompatible with the CDP system (which requires 6-decimal stablecoins) and most DeFi integrations.

Recommendation

We recommend adding a validation and reverting if the decimal is equal to zero.

Status

Acknowledged

Centralisation

The ENNI 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 ENNI project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. 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