BLOCKBITE

Security Audit

Enni

20. - 27.03.2026

Table of Contents

Introduction

Enni engaged BlockBite to do a detailed security analysis of their new CDP stable coin smart contracts.

The audit revealed 2 low-severity issues and 2 nitpicks; all of them are resolved or accepted by the project.

The following report describes all findings in details.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities complement but don't replace other vital measures to secure a project. The following sections will give an overview of the system, the issues uncovered and how they have been addressed.

Overview

Scope

The contract audit's scope comprised of three major components:

- Primarily, the Implementation security of the contract's source code
- Additionally, security of the overall design
- Additionally, resilience against economical attacks

The following contracts in the Enni Contracts repository were considered to be in scope:

- EnniCDP.sol
- EnniDirectMint.sol
- EnniMasterChef.sol
- EnniOracle.sol
- EnniToken.sol
- EnniVault.sol

Third-party dependencies or any other source files in the repository are not in scope.

The relevant commits are as following:

- 2ee7ea3bc9eb71124327b909f14998d4e06ed699 – Start of the audit
- 0028b4fa2e5cf73b1ae23b397e0aa519a4d7659f – Last reviewed revision

Functionality

ENNI is an immutable multi-currency CDP protocol on Ethereum. It allows users to deposit `WETH` as collateral and borrow stablecoins such as `enUSD` and `enCHF` at 0% interest. Additional currencies can be deployed using the same contracts with different constructor parameters.

The protocol consists of a set of core contracts that each handle a dedicated part of the system. The `EnniToken` contract is an ERC-20 token implementation with two minter slots and an optional fixed supply cap, and is reused for `ENNI`, `enUSD`, `enCHF`, and future tokens. The `EnniCDP` contract manages the collateralized debt positions for each currency, allowing users to lock `WETH` and mint the corresponding stablecoin against it. The `EnniOracle` contract provides the ETH price feed using Chainlink and Redstone, with an optional translation layer for non-USD currencies.

For `enUSD`, the protocol also includes the `EnniDirectMint` contract, which allows 1:1 minting and redemption (0.5% fee) between `USDC` or `USDT` and `enUSD`.

The `ENNI` token emissions are handled by the `EnniMasterChef` contract, which defines a 30-year emission schedule across up to 8 pools. Protocol revenue is directed to the `EnniVault` staking contract, where it is distributed directly to `ENNI` stakers.

Overall, the protocol is built around immutable contracts for collateralized borrowing, direct stablecoin minting, oracle-based pricing, and revenue distribution to token stakers.

Authorities

The system defines two distinct owner roles.

The first owner role is associated with the `EnniMasterChef` contract and is authorized to perform the following actions:

- `addPool()`: add a new pool to the reward distribution
- `setPool()`: update a pool’s allocation percentage

The second owner role is associated with the `EnniToken` contract, which is used for both the `ENNI` token and the protocol’s stablecoins. In this contract, the owner can transfer ownership to a different address and update the two minter slots.

According to the documentation, both ownership roles are intended to be renounced once the protocol is deployed and configured.

Findings

[LOW-1] Repay doesn't close a position

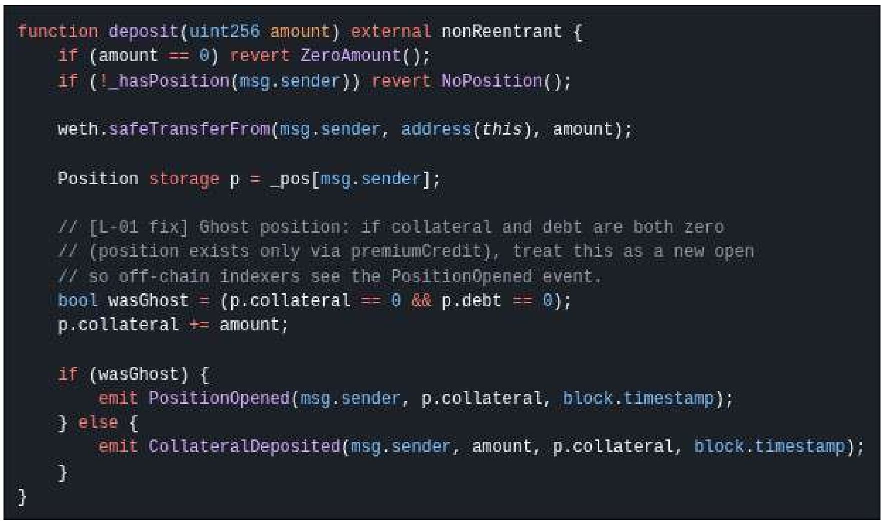
Description

`EnniCDP.sol` includes specific handling for ghost positions, defined as positions with no debt or collateral but with existing premium credit. In such cases, when a new deposit is made, the contract emits a `PositionOpened` event. This behavior ensures that off-chain indexers interpret the deposit as the creation of a new position and process it accordingly.

However, since no corresponding `PositionClosed` event is emitted when a position transitions into a ghost state, indexers may incorrectly assume that a user can maintain multiple open positions simultaneously, despite this being impossible within the system.

Source Code

https://github.com/enni-ch/contracts/blob/2ee7ea3bc9eb71124327b909f14998d4e06ed699/EnniCDP.sol#L239-L249



```
function deposit(uint256 amount) external nonReentrant {
    if (amount == 0) revert ZeroAmount();
    if (!_hasPosition(msg.sender)) revert NoPosition();

    weth.safeTransferFrom(msg.sender, address(this), amount);

    Position storage p = _pos[msg.sender];

    // [L-01 fix] Ghost position: if collateral and debt are both zero
    // (position exists only via premiumCredit), treat this as a new open
    // so off-chain indexers see the PositionOpened event.
    bool wasGhost = (p.collateral == 0 && p.debt == 0);
    p.collateral += amount;

    if (wasGhost) {
        emit PositionOpened(msg.sender, p.collateral, block.timestamp);
    } else {
        emit CollateralDeposited(msg.sender, amount, p.collateral, block.timestamp);
    }
}
```

Figure 1: EnniCDP deposit() function

Suggestion

1. Add an emit for `PositionClosed` if `debt == 0` and `collateral == 0` in `bayout()`, `_repay()`, `withdraw()` and `liquidate()`
2. Remove the special handling for ghost positions in `deposit()`

Resolution

The team acknowledged the issue.

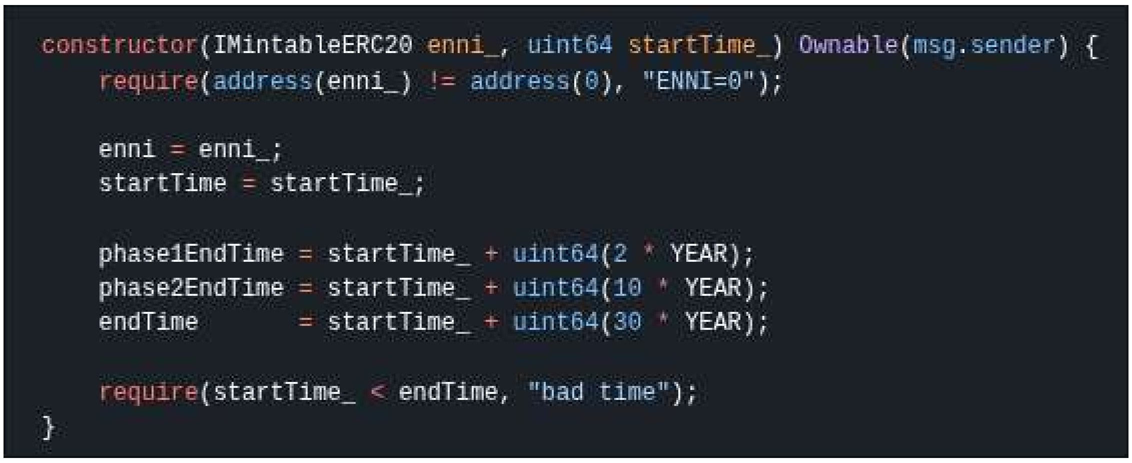
[LOW-2] Missing decimal check in EnniMasterChef

Description

`EnniMasterChef` assumes that the provided `IMintableERC20` token uses 18 decimals. While 18 decimals are common for ERC-20 tokens, this assumption is not universally valid, as demonstrated by the protocol's own stablecoins.

Source Code

https://github.com/enni-ch/contracts/blob/2ee7ea3bc9eb71124327b909f14998d4e06ed699/EnniMasterChef.sol#L98-L109



```
constructor(IMintableERC20 enni_, uint64 startTime_) Ownable(msg.sender) {
    require(address(enni_) != address(0), "ENNI=0");

    enni = enni_;
    startTime = startTime_;

    phase1EndTime = startTime_ + uint64(2 * YEAR);
    phase2EndTime = startTime_ + uint64(10 * YEAR);
    endTime       = startTime_ + uint64(30 * YEAR);

    require(startTime_ < endTime, "bad time");
}
```

Figure 2: EnniMasterChef constructor

Suggestion

Although this issue can be mitigated during deployment by ensuring the correct token configuration or redeploying the contract, it is recommended to enforce this assumption explicitly. This can be achieved by adding a validation check either in the deployment script or directly in the constructor to verify that the token uses 18 decimals.

Resolution

The team added a decimal check to the constructor with commit [0028b4fa2e5cf73b1ae23b397e0aa519a4d7659f]().

[NIT-1] Oracle cache is never updated

Description

`EnniOracleV1` maintains `lastGoodPrice` and `lastGoodUpdatedAt` as cached values intended to be used when underlying oracle calls revert or return stale data. However, the function responsible for updating this cache (`fetchPrice()`) is never invoked within the protocol, resulting in the contract consistently returning the values initialized during deployment.

Suggestion

Either integrate `fetchPrice()` into the protocol to ensure the cache is regularly updated, or remove the caching mechanism to avoid reliance on outdated data.

Resolution

The team acknowledged the issue.

[NIT-2] use `_spendAllowance()` in `burnFrom()`

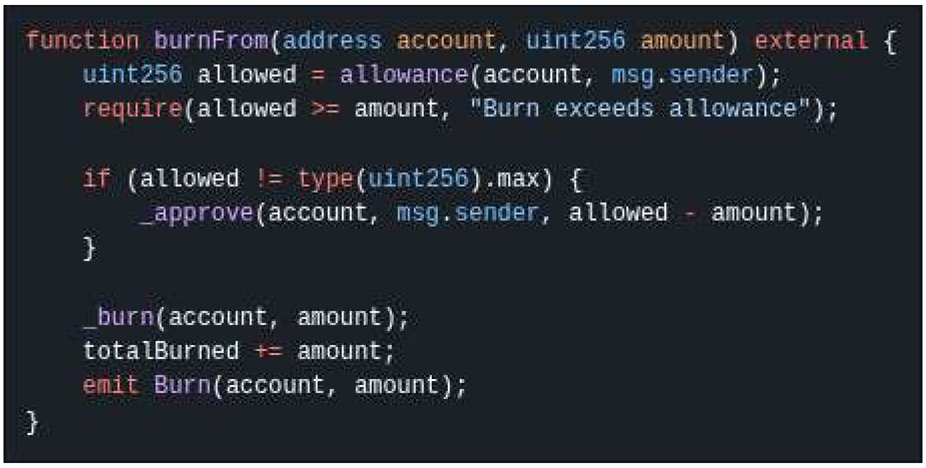
Description

OpenZeppelin's ERC-20 implementation uses `_spendAllowance()` to handle allowance consumption consistently, including preserving infinite allowances (`type(uint256).max`) and reverting when the available allowance is insufficient.

While the current implementation of `EnniToken.burnFrom()` preserves infinite allowance behavior and correctly checks whether sufficient allowance is available, it duplicates this logic instead of relying on the standard OpenZeppelin mechanism.

Source Code

https://github.com/enni-ch/contracts/blob/2ee7ea3bc9eb71124327b909f14998d4e06ed699/EnniToken.sol#L102-L113

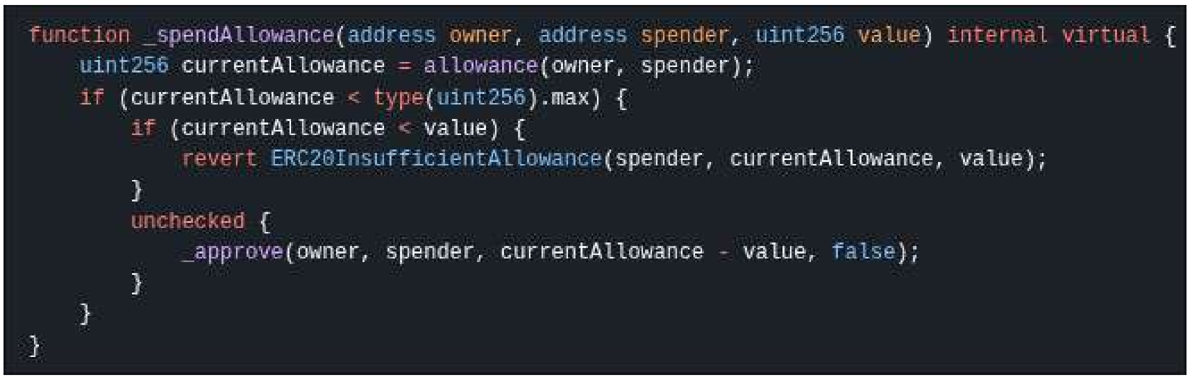


```
function burnFrom(address account, uint256 amount) external {
    uint256 allowed = allowance(account, msg.sender);
    require(allowed >= amount, "Burn exceeds allowance");

    if (allowed != type(uint256).max) {
        _approve(account, msg.sender, allowed - amount);
    }

    _burn(account, amount);
    totalBurned += amount;
    emit Burn(account, amount);
}
```

Figure 3: EnniToken.burnFrom() implementation



```
function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance < type(uint256).max) {
        if (currentAllowance < value) {
            revert ERC20InsufficientAllowance(spender, currentAllowance, value);
        }
        unchecked {
            _approve(owner, spender, currentAllowance - value, false);
        }
    }
}
```

Figure 4: Openzeppelins _spendAllowance() implementation

Suggestion

Although the current implementation behaves correctly, it is recommended to align `burnFrom()` with OpenZeppelin's `_spendAllowance()` logic for consistency, maintainability, and reduced risk of future deviations from the standard behavior.

Resolution

The team added the suggested change with commit 0028b4fa2e5cf73b1ae23b397e0aa519a4d7659f.

Techniques

A comprehensive examination has been performed utilizing Manual Review and Static Analysis techniques. The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors
- Assessing the code base to ensure compliance with current best practices and industry standards
- Ensuring contract logic meets the specifications and intentions of the client
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- Thorough line-by-line manual review of the entire code base

Disclaimer

The smart contracts provided by the client for audit purposes have been thoroughly analysed in compliance with the global best practices to date w.r.t Cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of tokens, their sale or any other aspect of the project. Crypto assets/tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty.

No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service, or other assets. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond Solidity that could present security risks.