Zellic

Ethena

Smart Contract Security Assessment

July 5, 2023

Prepared for:

Ethena Labs

Prepared by:

Katerina Belotskaia and Yuhang Wu

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment . . . 4

1.2 Non-goals and Limitations . . . 4

1.3 Results . . . 4

2 Introduction 6

2.1 About Ethena . . . 6

2.2 Methodology . . . 6

2.3 Scope . . . 7

2.4 Project Overview . . . 8

2.5 Project Timeline . . . 8

3 Detailed Findings 9

3.1 Inconsistencies in signers' and roles' management with implementation of access control . . . 9

3.2 Lack of input validation . . . 12

4 Discussion 13

4.1 Gas optimizations . . . 13

5 Threat Model 14

5.1 Module: AstEth.sol . . . 14

5.2 Module: EUSDAToken.sol . . . 16

5.3 Module: EUSDVariableDebtToken.sol . . . 17

5.4 Module: EthenaMinting.sol . . . 19

5.5 Module: EthenaStaking.sol 27
5.6 Module: StEthVariableDebtToken.sol 33

6 Audit Results **36**
6.1 Disclaimer 36

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Ethena Labs from June 26th to July 3th, 2023. During this engagement, Zellic reviewed Ethena's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an on-chain attacker drain the staking or rewarding funds?
- Could an on-chain attacker mint tokens using a faked signature?
- Could users lose the staking funds?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

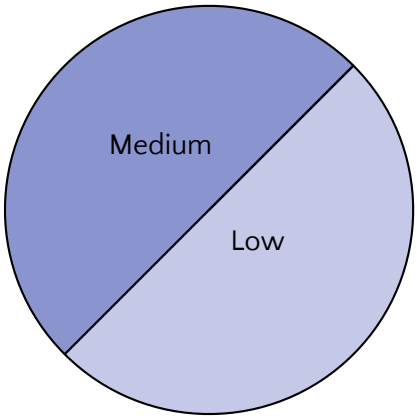
1.3 Results

During our assessment on the scoped Ethena contracts, we discovered two findings. No critical issues were found. One was of medium impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Ethena Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	0



2 Introduction

2.1 About Ethena

Ethena offers users the ability to mint/redeem a delta-neutral stablecoin, eUSD, that returns a sustainable yield to the holder. The yield is generated through staking Ethereum and the funding rate earned from the perpetual position hedging the delta. Users are able to mint eUSD, redeem eUSD, and stake/unstake their eUSD to receive their proportion of the generated yield.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Ethena Contracts

Repository	https://github.com/ethena-labs/ethena/protocols/eUSD/contracts/
Version	ethena: `d97c57917d95234787aad0db233c303d95bc9355`
Programs	• EthenaStaking • EthenaMinting • /lending/*
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 10 person-days. The assessment was conducted over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 26, 2023	Kick-off call
June 26, 2023	Start of primary review period
July 3, 2023	End of primary review period
July 12, 2023	Closing call

3 Detailed Findings

3.1 Inconsistencies in signers' and roles' management with implementation of access control

- **Target**: EthenaMinting
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

To execute the necessary transactions, the contract owner has the ability to assign accounts to one of two roles, Minter or Redeemer, with the `registerAllowedRole` function. This function sets the trusted signer address for the account and increments the `_roleActiveSignerCount`.

```
function registerAllowedRole(
  Role role,
  address account,
  address signer,
  bool allowed
) external override nonReentrant onlyOwner {
   ...
  _roleRegistry[role][account][signer] = allowed;
  _roleActiveSignerCount[role][account] += 1;
   ...
  }
```

Further, this value `_roleActiveSignerCount` is used to validate whether the account is assigned the appropriate role. To pass the verification, the account must have at least one confirmed signer.

```
function isMinter(address minter) public view returns (bool) {
  return _roleActiveSignerCount[Role.Minter][minter] ≥ 1;
}
```

Accounts with a role can also add their own trusted `signer` accounts using the `registerAllowedSigner` function and remove them with the `removeRole` function, which

deletes the `_roleRegistry[role][msg.sender][signer]` and decrements `_roleActiveSignerCount` for the `msg.sender`.

```
function registerAllowedSigner(
  Role role,
  address signer,
  bool allowed
) external override nonReentrant onlyRoleOrOwner(role) {
  if (signer == address(0)) revert InvalidAddress();
  _roleRegistry[role][msg.sender][signer] = allowed;
  _roleActiveSignerCount[role][msg.sender] += 1;
  emit RoleRegistered(role, msg.sender, msg.sender, allowed);
}
```

```
function removeRole(Role role, address signer)
    public nonReentrant onlyRoleOrOwner(role) {
  if (!_roleRegistry[role][msg.sender][signer]) revert InvalidRole();
  delete _roleRegistry[role][msg.sender][signer];
  _roleActiveSignerCount[role][msg.sender] -= 1;
  emit RoleRemoved(role, signer);
}
```

Impact

Since functions `registerAllowedRole` and `registerAllowedSigner` do not have checks that the `signer` has already been added to the `account`, the `_roleActiveSignerCount` can be incremented repeatedly without adding a new `signer`. And in this case, during the `removeRole` function call, the `_roleActiveSignerCount` for the `account` will never reach zero; as a result, the account will always be assigned with this role, even if the account does not have allowed signers.

In addition, the `registerAllowedRole` and `registerAllowedSigner` functions get the input bool `allowed` argument, which determines whether the `signer` address will be set as trusted. In case `allowed` is equal to `false`, the `_roleActiveSignerCount` will still be increased, although the `signer` address was not actually added. So it will lead to the incorrect counting of `_roleActiveSignerCount`.

Recommendations

We recommend considering using the OpenZeppelin's AccessControl module because it provides a secure, audited, and standards-compliant way to manage roles

and permissions and reduces the potential security risks associated with creating a custom solution.

Remediation

Ethena Labs acknowledged this finding and implemented a fix in commit df19a0d3.

3.2 Lack of input validation

- **Target**: EthenaMinting
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The following functions lack input validation.

- The `addSupportedAsset` function lacks a check that the `_assets` is not zero address.
- The `addSupportedAsset` function lacks a check that `_asset` has been already added.
- The `removeSupportedAsset` function lacks a check that the `_supportedAssets` contains the `_asset` address.

Impact

If important input parameters are not checked, it can result in functionality issues and unnecessary gas usage and can even be the root cause of critical problems. It is crucial to properly validate input parameters to ensure the correct execution of a function and to prevent unintended consequences.

Recommendations

Consider adding require statements and necessary checks to the above functions.

Remediation

Ethena Labs acknowledged this finding and implemented a fix in commit df19a0d3.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Gas optimizations

Switch the function parameters from memory to calldata functions wherever possible.

For example, the functions `verifyRoute`, `verifyReceipt`, `verifyOrder`, `hashReceipt` and `hashOrder` receive arguments in memory; however, the arguments are not altered. Arguments that are received as memory need to first be copied to memory, which adds extra gas cost. We recommend switching to `calldata` to reduce gas cost.

Remove an unnecessary `nonReentrant` modifier from functions that are not vulnerable to a reentry attack.

The modifier can be removed for the following functions: `registerAllowedRole`, `registerAllowedSigner`, `setEUSD`, `addCustodyWallet`, `removeCustodyWallet`, `addSupportedAsset`, `removeSupportedAsset` and `removeRole`

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

> Please note that our threat model was based on commit d97c5791, which represents a specific snapshot of the codebase. Therefore, it's important to understand that the absence of certain tests in our report may not reflect the current state of the test suite.
>
> During the remediation phase, Ethena Labs took proactive steps to address the findings and improve the test suite by adding test cases where applicable in commits 4069f57e and 357cdd3e. This demonstrates their dedication to enhancing the code quality and overall reliability of the system, which is commendable.

5.1 Module: AstEth.sol

Function: `burn(address from, address receiverOfUnderlying, uint256 amount, uint256 index)`

This function burns tokens from a specified address and may transfer an equivalent amount of another asset to a different address.

Inputs

- `from`
  - **Constraints**: N/A.
  - **Impact**: This is the address of the account from which the tokens will be burned.
- `receiverOfUnderlying`
  - **Constraints**: N/A.
  - **Impact**: This is the address that might receive an equivalent amount of another underlying asset, if it is not the contract itself.
- `amount`

  - **Constraints**: N/A.
  - **Impact**: The amount getting burned.
- `index`
  - **Constraints**: N/A.
  - **Impact**: The variable debt index of the reserve.

Branches and code coverage (including function calls)

Intended branches

- Burns tokens and transfers the same amount of tokens to another address successfully.
  ☐ Test coverage

Negative behavior

- Revert due to invalid burn amount.
  ☐ Negative test

Function call analysis

- `burn` → `_burnScaled(from, receiverOfUnderlying, amount, index)`
  - **External/Internal?** Internal.
  - **Argument control?** `from`, `receiverOfUnderlying`, `amount`, and `index`.
  - **Impact**: Burn a scaled balance token.
- `burn` → `IERC20(_underlyingAsset).safeTransfer(receiverOfUnderlying, amount)`
  - **External/Internal?** External.
  - **Argument control?** `receiverOfUnderlying` and `amount`.
  - **Impact**: Transfer tokens to an address.

Function: `mint(address caller, address onBehalfOf, uint256 amount, uint256 index)`

This function mints tokens to a specified address and increases the staked Ether balance by a given amount.

Inputs

- `caller`
  - **Constraints**: N/A.
  - **Impact**: The address initiating the mint operation.
- `onBehalfOf`

  - **Constraints**: N/A.
  - **Impact**: The address that will receive the newly minted tokens.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: The initial amount to be minted, which is then scaled.
- `index`
  - **Constraints**: N/A.
  - **Impact**: The next liquidity index of the reserve.

Branches and code coverage (including function calls)

Intended branches

- Mints tokens successfully.
  ☑ Test coverage

Negative behavior

- Revert due to invalid mint amount.
  ☐ Negative test

Function call analysis

- `mint` → `_mintScaled(caller, onBehalfOf, amount, index)`
  - **External/Internal?** Internal.
  - **Argument control?** `caller`, `onBehalfOf`, `amount`, and `index`.
  - **Impact**: Mint a scaled balance token.

5.2 Module: EUSDAToken.sol

Function: `handleRepayment(address user, address onBehalfOf, uint256 amount)`

This function manages the repayment process by adjusting a user's debt balance or burning excess from an underlying asset.

Inputs

- `user`
  - **Constraints**: N/A.
  - **Impact**: The address of the user who is initiating the repayment process.
- `onBehalfOf`

  - **Constraints**: N/A.
  - **Impact**: This is the address of the account for which the repayment is being made.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: This is the quantity of tokens that are being repaid.

Branches and code coverage (including function calls)

Intended branches

- Repayment succeed.
  ☑ Test coverage

Function call analysis

- `handleRepayment` → `_eUSDVariableDebtToken.decreaseBalanceFromInterest(onBehalfOf, amount)`
  - **External/Internal?** External.
  - **Argument control?** `onBehalfOf` and `amount`.
  - **Impact**: Decrease the amount of interest accumulated by the user.

5.3 Module: EUSDVariableDebtToken.sol

Function: `burn(address from, uint256 amount, uint256 index)`

This function burns a specified amount of tokens from a given address and returns the updated total supply.

Inputs

- `from`
  - **Constraints**: N/A.
  - **Impact**: The address from which the tokens will be burned.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: The amount getting burned.
- `index`
  - **Constraints**: N/A.
  - **Impact**: The variable debt index of the reserve.

Branches and code coverage (including function calls)

Intended branches

- Burns tokens successfully.
  - ☐ Test coverage

Negative behavior

- Revert due to invalid burn amount.
  - ☐ Negative test

Function call analysis

- `burn` → `_burnScaled(from, address(0), amount, index)`
  - **External/Internal?** Internal.
  - **Argument control?** `from`, `amount`, and `index`.
  - **Impact**: Burn a scaled balance token.

Function: `mint(address user, address onBehalfOf, uint256 amount, uint256 index)`

This function mints tokens for a specified address, adjusts borrowing allowances if needed, and returns the success of the operation and updated total supply.

Inputs

- `user`
  - **Constraints**: N/A.
  - **Impact**: The address performing the mint.
- `onBehalfOf`
  - **Constraints**: N/A.
  - **Impact**: The address of the user that will receive the scaled tokens.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: The amount of tokens getting minted.
- `index`
  - **Constraints**: N/A.
  - **Impact**: The next liquidity index of the reserve.

Branches and code coverage (including function calls)

Intended branches

- Mints tokens successfully.
  ☑ Test coverage

Negative behavior

- Revert due to invalid burn amount.
  ☐ Negative test

Function call analysis

- `mint` → `_mintScaled(user, onBehalfOf, amount, index)`
  - **External/Internal?** Internal.
  - **Argument control?** `user`, `onBehalfOf`, `amount`, and `index`.
  - **Impact**: Mint a scaled balance token.

5.4 Module: EthenaMinting.sol

Function: `addCustodyWallet(address wallet)`

Add new trusted custody wallet address by owner of the contract.

Function: `addSupportedAsset(IERC20 _asset)`

Allows owner of the contract to add asset address to the list of trusted assets.

Inputs

- `_asset`
  - **Constraints**: No.
  - **Impact**: The address will be added to the `supportedAssets`. The users will be able to deposit the asset tokens after that.

Branches and code coverage (including function calls)

Intended branches

- `_asset` was added to `supportedAssets`.
  ☑ Test coverage

Negative behavior

- Caller is not an owner.
  ☐ Negative test
- `supportedAssets` already contains `_asset`.

☐ Negative test

Function call analysis

- `_supportedAssets.add(_asset)`
  - **External/Internal?** Internal.
  - **Argument control?** `_asset`.
  - **Impact**: Add new `_asset` address to the `_supportedAssets` — if already added, returns false.

Function: `mint(Receipt receipt, Signature takerSignature, Signature makerSignature)`

Allows owner or account with Minter role to mint eUSD tokens for `beneficiary`.

Inputs

- `receipt`
  - **Constraints**: `receipt.order.asset` should be `_supportedAssets`.
  - **Impact**: The approved and signed by minter data necessary to mint tokens.
- `takerSignature`
  - **Constraints**: There is a check that the `order` hash was signed by `benefactor`.
  - **Impact**: The benefactor signature of the order.
- `makerSignature`
  - **Constraints**: There is a check that the `receipt` hash was signed by Minter.
  - **Impact**: The minter signature of the receipt.

Branches and code coverage (including function calls)

Intended branches

- The expected amount of eUSD was minted for beneficiary.
  ☐ Test coverage
- The assets tokens properly transferred from benefactor to custody addresses.
  ☐ Test coverage

Negative behavior

- The caller is not the Minter or Owner.
  ☑ Negative test
- Mint the receipt with the same nonce twice.
  ☐ Negative test

Function call analysis

- `_transferCollateral → token.safeTransferFrom(benefactor, addresses[i], amountToTransfer)`
  - **External/Internal?** External.
  - **Argument control?** `benefactor`, `addresses[i]`, and `amountToTransfer`.
  - **Impact**: Transfer assets tokens from benefactor to custody addresses.
- `eUSD.mint(receipt.order.beneficiary, receipt.affirmedAmount);`
  - **External/Internal?** External.
  - **Argument control?** `receipt.order.beneficiary` and `receipt.affirmedAmount`.
  - **Impact**: Mint `affirmedAmount` of eUSD tokens to beneficiary.

Function: `redeem(Receipt receipt, Signature takerSignature, Signature makerSignature)`

Burn eUSD tokens from `benefactor` and transfer corresponding amount of assets tokens to `beneficiary`. Function can be called only by owner or `Redeemer`.

Inputs

- `receipt`
  - **Constraints**: N/A.
  - **Impact**: The approved and signed by Redeemer data necessary to perform transaction.
- `takerSignature`
  - **Constraints**: There is a check that the `order` hash was signed by `benefactor`.
  - **Impact**: The benefactor signature of the order.
- `makerSignature`
  - **Constraints**: There is a check that the `receipt` hash was signed by the Redeemer.
  - **Impact**: The Redeemer signature of receipt.

Branches and code coverage (including function calls)

Intended branches

- The expected amount of eUSD was burned from benefactor.
  - ☐ Test coverage
- The assets tokens properly transferred to beneficiary.
  - ☐ Test coverage

Negative behavior

- The caller is not Redeemer or Owner.
  ☑ Negative test
- Redeem the receipt with the same nonce twice.
  ☑ Negative test

Function call analysis

- `eUSD.burnFrom(receipt.order.benefactor, receipt.order.baseAmount);`
  - **External/Internal?** External.
  - **Argument control?** `receipt.order.benefactor` and `receipt.order.baseAmount`.
  - **Impact**: Burn the `baseAmount` of eUSD tokens from `benefactor`.

Function: `registerAllowedRole(Role role, address account, address signer, bool allowed)`

Allows owner of contract to grant role for an account and also add a trusted signer.

Inputs

- `role`
  - **Constraints**: The Minter or Redeemer role.
  - **Impact**: An account with a role will be able to mint and redeem eUSD tokens.
- `account`
  - **Constraints**: Account != `address(0)`.
  - **Impact**: The receiver of the role.
- `signer`
  - **Constraints**: Signer != `address(0)`.
  - **Impact**: The trusted signer for an account.
- `allowed`
  - **Constraints**: No.
  - **Impact**: The owner can set and remove the allowance for the signer.

Branches and code coverage (including function calls)

Intended branches

- The account has the role.
  ☐ Test coverage

Negative behavior

- The caller is not an owner.
  - ☐ Negative test
- The `account` already has a role.
  - ☐ Negative test
- The `account` with the role already has the signer.
  - ☐ Negative test

Function: `registerAllowedSigner(Role role, address signer, bool allowed)`

Allows the account that already has a role to add a trusted signer.

Inputs

- `role`
  - **Constraints**: The caller should have this `role`.
  - **Impact**: The caller will add to the new signer for this role.
- `signer`
  - **Constraints**: Signer != `address(0)`.
  - **Impact**: The address of a trusted signer.
- `allowed`
  - **Constraints**: No.
  - **Impact**: The account can set and remove the allowance for the signer.

Branches and code coverage (including function calls)

Intended branches

- The account has the signer.
  - ☐ Test coverage

Negative behavior

- The caller does not have the role.
  - ☐ Negative test
- The account already has a signer.
  - ☐ Negative test

Function: `removeCustodyWallet(address wallet)`

Remove custody wallet address by the owner of the contract from the trusted wallets' list.

Function: `removeRole(Role role, address signer)`

Allows the account with a role to remove the signer and decrease the counter of trusted signers.

Inputs

- `role`
  - **Constraints**: The caller should have the role.
  - **Impact**: N/A.
- `signer`
  - **Constraints**: `_roleRegistry[role][msg.sender][signer]` is true.
  - **Impact**: N/A.

Branches and code coverage (including function calls)

Intended branches

- `_roleRegistry[role][msg.sender][signer]` was deleted and `_roleActiveSignerCount[role][msg.sender]` was decreased.
  - ☐ Test coverage

Negative behavior

- `_roleRegistry[role][msg.sender][signer]` is false.
  - ☐ Negative test
- The caller has not the role.
  - ☐ Negative test

Function: `removeSupportedAsset(IERC20 _asset)`

Allows owner of the contract to remove asset address from the list of trusted assets.

Inputs

- `_asset`
  - **Constraints**: No.
  - **Impact**: The address of assets that will be removed from the supported assets list.

Branches and code coverage (including function calls)

Intended branches

- `_asset` was removed from `supportedAssets`.
  - ☐ Test coverage

Negative behavior

- Caller is not an owner.
  - ☐ Negative test
- `supportedAssets` does not contain `_asset`.
  - ☐ Negative test

Function call analysis

- `supportedAssets.remove(address(_asset))`
  - **External/Internal?** Internal.
  - **Argument control?** `_asset`.
  - **Impact**: Remove the `_asset` address from `supportedAssets`.

Function: `setEUSD(IeUSD _eUSD)`

Allows owner of the contract to change the eUSD token address.

Inputs

- `_eUSD`
  - **Constraints**: `_eUSD` $\neq$ `address(0)`.
  - **Impact**: The eUSD tokens that will be minted in exchange for trusted tokens.

Branches and code coverage (including function calls)

Negative behavior

- Caller is not an owner.
  - ☐ Negative test
- `_eUSD` is zero.
  - ☐ Negative test

Function: `transferToCustody(address wallet, address asset, uint256 amount)`

Allows owner or account with the Minter role to transfer tokens or native tokens from the current contract to the trusted custody wallet.

Inputs

- `wallet`
  - **Constraints**: the `CustodyWallet` should contain the `wallet` address.
  - **Impact**: The receiver of tokens.
- `asset`
  - **Constraints**: No.
  - **Impact**: The token address.
- `amount`
  - **Constraints**: No.
  - **Impact**: The amount will be transferred.

Branches and code coverage (including function calls)

Negative behavior

- The caller is not a Minter.
  - ☐ Negative test
- The caller is not an owner.
  - ☐ Negative test
- The wallet is not trusted.
  - ☐ Negative test

Function call analysis

- `wallet.call{value: amount}('')`
  - **External/Internal?** External.
  - **Argument control?** `wallet` and `amount`.
  - **Impact**: Transfer native tokens from the current contract to a trusted wallet.
- `IERC20(asset).safeTransfer(wallet, amount)`
  - **External/Internal?** External.
  - **Argument control?** `asset`, `wallet`, and `amount`.
  - **Impact**: Transfer asset tokens from the current contract to a trusted wallet account.

5.5 Module: EthenaStaking.sol

Function: `rescueTokens(IERC20 token, uint256 amount, address to)`

Allows the owner to rescue tokens accidentally sent to the contract.

Inputs

- `token`
  - **Constraints**: Cannot be the same address as of eUSD.
  - **Impact**: The token to be rescued.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: The amount of tokens to be rescued.
- `to`
  - **Constraints**: N/A.
  - **Impact**: The address to be rescued to.

Branches and code coverage (including function calls)

Intended branches

- Tokens are successfully rescued to an address.
  ☑ Test coverage

Negative behavior

- The contract call was reverted because the token address is the same as the eUSD address.
  ☐ Negative test

Function call analysis

- `rescueTokens` → `token.safeTransfer(to, amount)`
  - **External/Internal?** External.
  - **Argument control?** `to` and `amount`.
  - **Impact**: Safely transfer a certain `amount` of a `token` to the specific address `to`.

Function: `stakeWithPermit(uint256 eUSDIn, uint256 minStakedeUSDOut, address beneficiary, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

Allows users to stake their eUSD tokens using a permit signature.

Inputs

- `eUSDIn`
  - **Constraints**: Not zero — should be at least `MIN_INITIAL_STAKE` during the first stake.
  - **Impact**: The amount of eUSD tokens to stake.
- `minStakedeUSDOut`
  - **Constraints**: N/A.
  - **Impact**: The minimum amount of stakedeUSD tokens to receive.
- `beneficiary`
  - **Constraints**: N/A.
  - **Impact**: The beneficiary of the stakedeUSD tokens.
- `deadline`
  - **Constraints**: N/A.
  - **Impact**: The deadline for the permit signature.
- `v`
  - **Constraints**: Must be valid `secp256k1` signature from `owner` account over EIP-712–formatted function arguments.
  - **Impact**: The `v` value of the permit signature.
- `r`
  - **Constraints**: Must be valid `secp256k1` signature from `owner` account over EIP-712–formatted function arguments.
  - **Impact**: The `r` value of the permit signature.
- `s`
  - **Constraints**: Must be valid `secp256k1` signature from `owner` account over EIP712-formatted function arguments.
  - **Impact**: The `s` value of the permit signature.

Branches and code coverage (including function calls)

Intended branches

- Stake tokens successfully with signature.
  ☑ Test coverage

Negative behavior

- The contract call is reverted when the signature is expired.
  ☐ Negative test
- The contract call is reverted when the signer is not the owner or `address(0)`.
  ☑ Negative test

Function call analysis

- `stakeWithPermit` → `eUSD.permit(_msgSender(), address(this), eUSDIn, deadline, v, r, s)`
  - **External/Internal?** External.
  - **Argument control?** `eUSDIn`, `deadline`, `v`, `r`, and `s`.
  - **Impact**: Sets `eUSDIn` as allowance of `address(this)` account over `_msgSender()` account's WETH10 token, given `_msgSender()` account's signed approval.
- `stakeWithPermit` → `_stake(eUSDIn, minStakedeUSDOut, beneficiary)`
  - **External/Internal?** Internal.
  - **Argument control?** `eUSDIn`, `minStakedeUSDOut`, and `beneficiary`.
  - **Impact**: N/A.

Function: `stake(uint256 eUSDIn, uint256 minStakedeUSDOut, address beneficiary)`

Allows users to stake their eUSD tokens.

Inputs

- `eUSDIn`
  - **Constraints**: Not zero — should be at least `MIN_INITIAL_STAKE` during the first stake.
  - **Impact**: The amount of eUSD tokens to stake.
- `minStakedeUSDOut`
  - **Constraints**: N/A.
  - **Impact**: The minimum amount of stakedeUSD tokens to receive.
- `beneficiary`
  - **Constraints**: N/A.
  - **Impact**: The beneficiary of the stakedeUSD tokens.

Branches and code coverage (including function calls)

Intended branches

- Stake tokens successfully (first stake).
  ☑ Test coverage
- Stake tokens successfully (non-first stake).
  ☑ Test coverage

Negative behavior

- The contract call is reverted when `eUSDIn` is less than `MIN_INITIAL_STAKE` during the first stake.
  ☑ Negative test
- The contract call is reverted when allowance is insufficient.
  ☑ Negative test
- The contract call is reverted when `stakedeUSDOut` is less than `minStakedeUSDOut`.
  ☐ Negative test

Function call analysis

- `stake` → `_stake(eUSDIn, minStakedeUSDOut, beneficiary)`
  - **External/Internal?** Internal.
  - **Argument control?** `eUSDIn`, `minStakedeUSDOut`, and `beneficiary`.
  - **Impact**: N/A.

Function: `transferInRewards(uint256 amount)`

Allows the owner to transfer rewards from the controller contract into this contract.

Inputs

- `amount`
  - **Constraints**: Should not be zero.
  - **Impact**: The amount of rewards to transfer.

Branches and code coverage (including function calls)

Intended branches

- Owner could transfer rewards.
  ☑ Test coverage

Negative behavior

- The contract call is reverted when `amount` is zero.
  ☐ Negative test
- The contract call is reverted when the controller does not have enough tokens.
  ☐ Negative test

Function call analysis

- `transferInRewards` → `eUSD.transferFrom(msg.sender, address(this), amount)`
  - **External/Internal?** External.

  - **Argument control?** `amount`.
  - **Impact**: Transfer eUSD tokens from the controller to this contract.

Function: `unstakeWithPermit(uint256 steUSDIn, uint256 mineUSDOut, address beneficiary, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

Allows users to withdraw their staked eUSD tokens using a permit signature.

Inputs

- `steUSDIn`
  - **Constraints**: Should not be zero.
  - **Impact**: The amount of steUSD tokens to unstake.
- `mineUSDOut`
  - **Constraints**: N/A.
  - **Impact**: The minimum acceptable amount of eUSD tokens to receive.
- `beneficiary`
  - **Constraints**: N/A.
  - **Impact**: The beneficiary of the eUSD tokens.
- `deadline`
  - **Constraints**: N/A.
  - **Impact**: The deadline for the permit signature.
- `v`
  - **Constraints**: Must be valid `secp256k1` signature from the `owner` account over EIP-712–formatted function arguments.
  - **Impact**: The `v` value of the permit signature.
- `r`
  - **Constraints**: Must be valid `secp256k1` signature from the `owner` account over EIP-712–formatted function arguments.
  - **Impact**: The `r` value of the permit signature.
- `s`
  - **Constraints**: Must be valid `secp256k1` signature from the `owner` account over EIP-712–formatted function arguments.
  - **Impact**: The `s` value of the permit signature.

Branches and code coverage (including function calls)

Intended branches

- Unstake tokens successfully with signature.
  ☑ Test coverage

Negative behavior

- The contract call is reverted when the signature is expired.
  - ☐ Negative test
- The contract call is reverted when the signer is not the owner or `address(0)`.
  - ☑ Negative test

Function call analysis

- `unstakeWithPermit` → `stakedeUSD.permit(_msgSender(), address(this), steUSDIn, deadline, v, r, s)`
  - **External/Internal?** External.
  - **Argument control?** `steUSDIn`, `deadline`, `v`, `r`, and `s`.
  - **Impact**: Sets `steUSDIn` as allowance of the `address(this)` account over `_msgSender()` account's WETH10 token, given `_msgSender()` account's signed approval.
- `unstakeWithPermit` → `_unstake(steUSDIn, mineUSDOut, beneficiary)`
  - **External/Internal?** Internal.
  - **Argument control?** `steUSDIn`, `mineUSDOut`, and `beneficiary`.
  - **Impact**: N/A.

Function: `unstake(uint256 steUSDIn, uint256 mineUSDOut, address beneficiary)`

Allows users to withdraw their staked eUSD tokens.

Inputs

- `steUSDIn`
  - **Constraints**: Should not be zero.
  - **Impact**: The amount of steUSD tokens to unstake.
- `mineUSDOut`
  - **Constraints**: N/A.
  - **Impact**: The minimum acceptable amount of eUSD tokens to receive.
- `beneficiary`
  - **Constraints**: N/A.
  - **Impact**: The beneficiary of the eUSD tokens.

Branches and code coverage (including function calls)

Intended branches

- Unstake tokens successfully.
  ☑ Test coverage

Negative behavior

- The contract call is reverted when `eUSDOut` is less than `mineUSDOut`.
  ☐ Negative test

Function call analysis

- `unstake` → `_unstake(steUSDIn, mineUSDOut, beneficiary)`
  - **External/Internal?** Internal.
  - **Argument control?** `steUSDIn`, `mineUSDOut`, and `beneficiary`.
  - **Impact**: N/A.

5.6 Module: StEthVariableDebtToken.sol

Function: `burn(address from, uint256 amount, uint256 index)`

This function burns a specified amount of tokens from a given address and returns the updated total supply.

Inputs

- `from`
  - **Constraints**: N/A.
  - **Impact**: The address from which the tokens will be burned.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: The amount getting burned.
- `index`
  - **Constraints**: N/A.
  - **Impact**: The variable debt index of the reserve.

Branches and code coverage (including function calls)

Intended branches

- Burns tokens successfully.
  ☐ Test coverage

Negative behavior

- Revert due to invalid burn amount.
  ☐ Negative test

Function call analysis

- `burn` → `_burnScaled(from, address(0), amount, index)`
  - **External/Internal?** Internal.
  - **Argument control?** `from`, `amount`, and `index`.
  - **Impact**: Burn a scaled balance token.

Function: `mint(address user, address onBehalfOf, uint256 amount, uint256 index)`

This function mints tokens for a specified address, adjusts borrowing allowances if needed, and returns the success of the operation and updated total supply.

Inputs

- `user`
  - **Constraints**: N/A.
  - **Impact**: The address performing the mint.
- `onBehalfOf`
  - **Constraints**: N/A.
  - **Impact**: The address of the user that will receive the scaled tokens.
- `amount`
  - **Constraints**: N/A.
  - **Impact**: The amount of tokens getting minted.
- `index`
  - **Constraints**: N/A.
  - **Impact**: The next liquidity index of the reserve.

Branches and code coverage (including function calls)

Intended branches

- Mints tokens successfully.
  ☑ Test coverage

Negative behavior

- Revert due to invalid burn amount.
  ☐ Negative test

Function call analysis

- `mint` → `_mintScaled(user, onBehalfOf, amount, index)`
  - **External/Internal?** Internal.
  - **Argument control?** `user`, `onBehalfOf`, `amount`, and `index`.
  - **Impact**: Mint a scaled balance token.

6 Audit Results

At the time of our audit, the audited code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Ethena contracts, we discovered two findings. No critical issues were found. One was of medium impact and one was of low impact. Ethena Labs acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.