CERTORA

Formal Verification Report for Euler

This document describes the specification and verification of Euler's smart contracts using the Certora Prover. The work was undertaken from September 5, 2021 to October 31, 2021. The latest commit that was reviewed and run through the Certora Prover was 5c5eef86.

Our formal verification focused on the state of the markets and assets, and the methods reachable from the public interfaces of the EToken, DToken, and Markets modules. We also performed a manual audit of all of the modules.

The rules used for verification have been added to Euler's public repository; you can find them in the certora/ directory along with a README that explains how to rerun the verification.

All of the issues we discovered were promptly fixed by the Euler team prior to the initial release of the system.

Main Issues Discovered

Severity: High

Issue:	**Users can prevent their accounts from being liquidated**
Description:	Due to a require statement that is tested during liquidation, users can force all attempts to liquidate their accounts to fail and revert (details below).
Response:	This issue depends on a malicious asset being promoted out of borrow isolation. Euler has done the suggested hardening to block this specific attack, but governance should be aware the promotion of a malicious asset still has other serious security implications (see the Euler documentation), and assets must be thoroughly vetted before removing isolation or adding collateral factors.

Severity: High

Issue:	**"Exact output" swaps via Uniswap can leave Uniswap with allowance from Euler**
Description:	The return value from `IERC20.approve()` is ignored in some cases, allowing to reset Uniswap's allowance from Euler, they don't check the return value, which according to the ERC-20 Token Standard, is a boolean value indicating whether the operation succeeded (details).
Response:	For this problem to occur, an honest token would have to have an approve() method that fails, and that failure must be indicated with a return value of false (instead of reverting). Although we aren't aware of any such tokens, the Swap module now uses safeApprove() everywhere, as suggested.

Severity: High

Issue:	**Parameters for multihop swaps via Uniswap aren't validated**
Description:	The functions Swap.swapUniExactInput() and Swap.swapUniExactOutput() don't properly check params.path, making it possible to steal tokens from Euler (details below).
Response:	The Swap module depends on approvals being maintained properly. This attack requires approvals to be accidentally granted somehow (potentially as described in the previous issue). As suggested, Euler now makes additional checks on the Uniswap path to validate that the tokens in the path are as expected, although the primary security enforcement remains the approvals system.

Severity: Medium

Issue:	**“Exact output” swaps via Uniswap don’t support all ERC-20 tokens**
Description:	Many commonly used tokens do not return a boolean from IERC20.approve(); Euler methods will revert for these tokens (details).
Response:	This was also addressed by changing the Swap module to use safeApprove() everywhere (see above).

Severity: Medium

Issue:	**Parameters for swaps via 1inch aren’t validated**
Description:	The function swap1Inch() doesn’t check that params.payload matches params.underlyingIn, params.underlyingOut and params.amount. It also doesn’t check that params.payload specifies Euler’s address as both the account who gives away tokens to 1inch and the account who receives the tokens from 1inch.
Response:	Acknowledged. 1inch has a variety of methods and is used as a black box. To allow the users to use all 1inch’s functions, the code doesn’t enforce any specific format for params.payload. It relies on the approval mechanism of ERC-20 tokens to prevent any malicious swaps.

Severity: Medium

Issue:	**Exec.pTokenWrap() doesn't work with "deflationary" ERC-20 tokens**
Description:	Euler is intended to work with deflationary tokens, but Exec.pTokenWrap() will fail with these tokens (details).
Response:	PTokens can only be created for collateral assets. One of the criteria for collateral assets is that their balances are "well behaved", which excludes deflationary tokens.

Severity: Medium

Issue:	**BaseLogic.decreaseBorrow can also increase debt and emit a Borrow event**
Description:	In some cases, rounding error can cause decreaseBorrow to increase the borrow instead (details).
Response:	Acknowledged. This is a necessary consequence of the design. In order for off-chain systems to properly track the debt owed by an account, Borrow and Repay events are issued. However, interest is accrued second-by-second, which obviously cannot be tracked in real-time with events. To solve this, when an account's borrow is re-assessed (which it must be in order to increase or decrease a borrow), the accrued interest must be logged. To reduce gas usage and simplify the implementation, what is actually logged is the *change* in the borrow. So a repay operation for X units will actually result in a Repay event of only X-I units, where I was the interest that accrued. If the repay amount X is in fact smaller than I, then (counter-intuitively) a Borrow event will be issued instead.

Severity: Low

Issue:	Missing initialization of the installer module address
Description:	A missing initialization results in higher gas costs in every call coming through the installer proxy (details).
Response:	This only has a gas impact when invoking the Installer module, which is a relatively rare operation and is paid for by governance when upgrading modules, and never by protocol users. Nevertheless, we've added the initialisation as suggested.

Severity: Low

Issue:	View functions in Markets have undefined behavior on invalid input
Description:	Several view functions in Markets behave inconsistently when their input pair is an invalid underlying token pair
Response:	This was originally by design, however after discussion with Certora we've decided to define this behaviour for methods where applicable (in the nat spec documentation), and by reverting with error messages elsewhere.

Severity: Recommendation

Issue:	Code cleanliness and gas optimizations
Description:	During our manual code review, we found several small details that could be improved (details).
Response:	We have implemented some of the suggested optimisations, where it had a measurable improvement and made sense to do so.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Summary of formal verification

Overview of Euler contracts

The Euler system defines three types of tokens. ETokens are held by lenders and can be used to reclaim the loaned underlying assets and interest. DTokens are held by borrowers, and can be burned when the debt is repaid. PTokens represent collateral that Euler is not allowed to lend out.

The Euler Architecture uses a module system to split the system functionality across several contracts. All state lives in the storage of the singleton Euler contract, but the code is stored in several module contracts. Euler uses delegatecalls to dispatch method calls to the appropriate modules.

For our verification efforts, we have written invariants and rules that describe the valid states and transitions for the state variables of the Euler contract (defined in the inherited Storage contract). Successful verification of a module M ensures that no call to any of the methods of M can violate the invariants.

Our rules and invariants focus on the “Markets and Assets” portion of the Euler state. We have verified those rules against the DToken, EToken, and Markets modules.

The remainder of this section describes the rules and invariants that we have checked.

Assumptions and simplifications made during verification

We made the following assumptions during our verification:

- We unroll loops twice. If a violation requires more than two iterations through a loop, it will not be detected.

- To avoid limitations around nonlinear arithmetic, we replaced underlyingDecimalsScaler with 1.

- We have not verified the correctness of the proxying or module dispatching.

- Our verification assumes that interest is accrued correctly; our verification removes the accrueInterest method.

- We have reduced MAX_ENTERED_MARKETS to 3.

- We assume that cross-module calls do not have side-effects. In particular, we assume that callInternalModule does not have any side-effects on the state of the contract.

- We assume that the decimal conversions in getCurrentOwedExact and roundUpOwed are correct.

- We assume that the underlying tokens are correct implementations of the ERC20 standard.

The complete list of modifications made for verification are contained in the file certora/applyHarness.patch. The simplifications are contained in the file certora/harness/BaseHarness.sol and in the methods block of certora/specs/common.spec.

Properties

This section gives detailed English descriptions of the properties we have verified; for the full details, see the rules in certora/spec/Markets.spec.

The status of each rule is indicated by one of the following icons:

CERTORA

✅ indicates the rule is formally verified on the latest reviewed commit, with the listed assumptions and simplifications.

❌ indicates the rule was violated under one of the tested versions of the code.

✍ indicates the rule is not yet formally specified.

⏱ indicates that some functions cannot be verified because the rules timed out

invariants on balances

[✅] eToken_supply_equality

For a given EToken, the totalBalance variable is the sum of the reserve balance and all user EToken balances.

[✍][1] dToken_supply_equality

For a given EToken, the totalBorrows variable is the sum of the outstanding DTokens for all users.

[✍][2] eToken_euler_supply

For a given underlying token, The underlying ERC20 balance of the Euler system is equal to the total supply of ETokens minus the total supply of DTokens.

structural invariants

[✅] underlying_eToken_equality

There is a one-to-one correspondence (bijection) between ETokens and underlying tokens, stored in the eTokenLookup.underlying and underlyingTokenLookup mappings.

[1] The tool is currently giving spurious counterexamples on DToken.repay and EToken.burn that should be ruled out by introducing and proving additional invariants bounding users' individual owed amounts by the total borrows. Unfortunately we were unable to complete the implementation of these invariants.

[2] There is an error in our implementation of this rule that is causing spurious counterexamples; we are unable to verify this rule.

[✔] pToken_underlying_equality

There is a one-to-once correspondence (bijection) between PTokens and underlying tokens, stored in the pTokenLookup. and reversePTokenLookup mappings.

privilege invariants

[✔] userAssets_transactions_contained

With the exception of transfers, no transaction affects more than one user's balance.

Detailed description of discovered problems

Liquidation prevention

Due to a require statement that is tested during liquidation, users can force all attempts to liquidate their accounts to fail and revert.

To perform this attack, an attacker can perform the following steps:

1. Create a malicious ERC-20 token (denoted by M), create a pool on Uniswap for the pair M and RiskManager.referenceAsset, and activate a market for M on Euler using Markets.activateMarket().

2. Deposit a very large amount of M tokens into Euler using EToken.deposit(). The amount has to be very close to type(uint112).max. Now the totalBalances value of eM is very close to type(uint112).max.

3. Enter into M's market on Euler using Markets.enterMarket().

4. Use a different account to borrow M tokens from Euler using DToken.borrow(). The amount doesn't matter as long as it's more than zero. Now the totalBorrows value of dM is greater than zero.

Now, whenever someone tries to liquidate the attacker's account using Liquidation.liquidate(), this is what will happen:

1. On Liquidation.computeLiqOpp(), the function getAccountLiquidity() is invoked with the attacker's account. This function calls RiskManager.computeLiquidity() and eventually reaches initAssetCache() with M as the underlying.
2. On initAssetCache(), when Euler calls M.balanceOf(address(this)), the attacker can define the malicious M token to return a very large value that is very close to type(uint112).max.
3. When initAssetCache() computes newTotalBorrows, it must be greater than assetCache.totalBorrows because assetCache.totalBorrows is not zero. Therefore, feeAmount will be greater than zero and the function will enter the if statement that computes newTotalBalances.
4. Since both assetCache.totalBalances and assetCache.poolSize are very large values that are close to type(uint112).max, the value computed for newTotalBalances will be larger than type(uint112).max. It also means that newTotalBalances is greater than assetCache.totalBalances, so the function will enter the last if statement.
5. There will be a revert on encodeAmount() since newTotalBalances is greater than type(uint112).max (MAX_SANE_AMOUNT), meaning that the attacker's account cannot be liquidated.

Suggestions for mitigation: the function initAssetCache() should skip the last if statement if newTotalBalances is greater than type(uint112).max (MAX_SANE_AMOUNT) or if newTotalBorrows is greater than type(uint144).max (MAX_SANE_DEBT_AMOUNT). This additional condition should be added in order to avoid a similar issue with encodeDebtAmount(), even though borrowing such amount of M tokens isn't practical because it requires an enormous amount of collateral (assuming M has the default borrow factor).

Exact output swaps

"Exact output" swaps via Uniswap can leave Uniswap with allowance from Euler.

When the functions Swap.swapUniExactOutputSingle() and Swap.swapUniExactOutput() call IERC20.approve() to reset Uniswap's allowance from Euler, they don't check the return value, which according to the [ERC-20 Token Standard](), is a boolean value indicating whether the operation succeeded.

Suggested fix: Use Utils.safeApprove(), which reverts if the return value was false, instead of IERC20.approve().

Multihop Uniswap swaps

The functions Swap.swapUniExactInput() and Swap.swapUniExactOutput() don't check that params.path starts with params.underlyingIn and ends with params.underlyingOut.

Together with the [previous issue](), it is possible to steal tokens from Euler.

Let T1 be an ERC-20 token that can return false on approve(). An attacker can gain tokens from Euler as follows:

1. Create a malicious ERC-20 token (denoted by M), create a pool on Uniswap for the pair M and RiskManager.referenceAsset, and activate a market for M on Euler using Markets.activateMarket().

2. Swap T1 for another token using Swap.swapUniExactOutputSingle() or Swap.swapUniExactOutput() with a very large params.amountInMaximum. The params.amountOut can remain zero, its value doesn't matter. For the attack to continue, the last call to IERC20.approve() must failed (leaving Uniswap with a very high allowance). The return value of IERC20.approve() will be false, indicating the failure, but the code doesn't check it.

3. Swap M for another token T2 using Swap.swapUniExactInput() with a params.path that starts with T1 and ends with T2. Uniswap will take T1 tokens from Euler and give it T2 tokens in return. On finalizeSwap(), Euler will confirm that it now have swap.amountIn less M tokens and swap.amountOut more T2 tokens. The balance check for T2 will succeed because Uniswap transferred that amount of T2 tokens to Euler, and the balance check for M will also succeed, as the malicious token controls the return value of M.balanceOf().

Overall, the attacker received a significant amount of T2 tokens, that can be withdrawn from Euler, and Euler lost the same worth of T1 tokens.

Suggested fix: Decode underlyingIn and underlyingOut from params.path, instead of receiving them as additional arguments.

Exact output unsupported for some tokens

"Exact output" swaps via Uniswap don't support all ERC-20 tokens.

The functions Swap.swapUniExactOutputSingle() and Swap.swapUniExactOutput() call IERC20.approve() that is defined to return a boolean. However, there are legitimate non-standards-compliant tokens like USDT that don't return a return value on approve(). All IERC20.approve() calls to such tokens will revert because of the missing return value.

Suggested fix: Use Utils.safeApprove(), which support all ERC-20 tokens, instead of IERC20.approve().

Deflationary PTokens

Exec.pTokenWrap() doesn't work with "deflationary" ERC-20 tokens.

The function Exec.pTokenWrap() requires that exactly `amount` underlying tokens will be transferred to the corresponding pToken address, when requesting to transfer amount underlying tokens. However, it violates this section from [Euler's Architecture]:

> "We try to work as well as possible with "deflationary" tokens. These are tokens where when you request a transfer for X, fewer than X tokens are actually transferred."

Users can still wrap their "deflationary" tokens with pTokens by calling PToken.wrap() directly.

DecreaseBorrow can increase borrow

BaseLogic.decreaseBorrow can also increase debt and emit a Borrow event.

The function BaseLogic.decreaseBorrow sets the new owed (debt) to owedRemaining, although it can also be larger than the original owed due to the rounding up. This scenario will also emit a Borrow event at the end because the debt has increased.

For example, if a user calls DToken.repay() with amount=0 (perhaps accidentally), his debt can be increased and the transaction will emit both RequestRepay and Borrow events.

Suggested fix: Add a require statement that owedRemaining cannot be less than the original owed to prevent the debt from increasing.

Missing installer module initialization

Euler's constructor doesn't set trustedSenders[installerProxy].moduleImpl to installerModule. This results in higher gas costs in every call coming through the installer proxy.

Suggested fix: Set trustedSenders[installerProxy].moduleImpl to installerModule.

Markets undefined behavior

The view functions in Markets have undefined behavior on invalid input.

The view functions Markets.underlyingToAssetConfigUnresolved(), Markets.eTokenToUnderlying(), Markets.eTokenToDToken(), Markets.interestRateModel(), Markets.interestRate(), Markets.interestAccumulator(), Markets.reserveFee() and Markets.getPricingConfig() in the Markets module don't deal with the case where their input is an invalid underlying token / eToken address. Besides Markets.interestAccumulator(), they all return zero values. The function Markets.interestAccumulator() is an exception because it reverts unexpectedly, with no error message, due to division by zero in initAssetCache(), when computing newTotalBorrows.

Suggested fix: In case the input of these functions is an invalid underlying token / eToken address, revert with an informative error message.

Code cleanliness and gas optimizations

Issue:	**Unnecessary casting in Markets.doActivateMarket()**
Description:	The function Markets.doActivateMarket() contains an unnecessary casting from address to address.
Mitigation/Fix:	Remove this unnecessary casting.

Severity: Recommendation

Issue:	**Local variable in Installer.installModules() and Exec.batchDispatch() shadows a state variable**
Description:	The local variable moduleId in the functions Installer.installModules() and Exec.batchDispatch() shadows the state variable moduleId inherited from BaseModule.
Mitigation/Fix:	Rename this local variable.

Severity: Recommendation

Issue:	**_getEnteredMarketIndex() can be inlined to skip checks**
Description:	The function _getEnteredMarketIndex() can be replaced with cheaper accountLookup[account].firstMarketEntered or marketsEntered[account][index] lines of code. Especially when looping over all the markets an account has entered.
Mitigation/Fix:	save gas by deleting this function and inlining its implementation where it was used earlier (in the functions BaseLogic.doEnterMarket() and BaseLogic.doExitMarket()).

Severity: Recommendation

Issue:	**Early exit optimization in BaseLogic.getEnteredMarketsArray()**
Description:	The function BaseLogic.getEnteredMarketsArray() can immediately return a 0-length addresses array if numMarketsEntered is zero. There is no reason to do any other operations in this case.
Mitigation/Fix:	Save gas by immediately returning a 0-length addresses array in this case, instead of doing other operations for no reason.

Severity: Recommendation

Issue:	**Early exit optimization in DToken.repay() and EToken.burn()**
Description:	The functions DToken.repay() and EToken.burn() can immediately return if owed is zero. There is no reason to do any other operations in this case.
Mitigation/Fix:	Save gas by immediately returning in this case, instead of doing other operations for no reason.

Severity: **Recommendation**

Issue:	**Unnecessary array boundaries checks**
Description:	When loading an array element more than once, there is no reason to check again that the index doesn't exceed the array limits.
Mitigation/Fix:	Save gas by caching the array element in a local variable instead of loading it again.

Severity: **Recommendation**

Issue:	**Trivial require statement in Governance.convertReserves()**
Description:	The function Governance.convertReserves() requires amount to be less than or equal to assetStorage.reserveBalance, right after an if statement that sets amount to assetStorage.reserveBalance. Therefore, the require statement will always pass if the previous if statement was entered.
Mitigation/Fix:	Save gas by surrounding that require statement with an else { ... } block.

Severity: **Recommendation**

Issue:	**Expensive require statement in Swap.initSwap()**
Description:	The function Swap.initSwap() requires that assetStorageIn.underlying != address(0) and assetStorageOut.underlying != address(0), to ensure these underlying tokens has markets on Euler. These require statement are expensive since they executes two SLOADs.
Mitigation/Fix:	Save gas by replacing these require statements with the equivalent requirements that swap.eTokenIn != address(0) and swap.eTokenOut != address(0).

Severity: **Recommendation**

<table>
<tr><th>Issue:</th><th>Unnecessary checked arithmetics in PToken.transferFrom()</th></tr>
<tr><td>Description:</td><td>
The arithmetic allowances[from][msg.sender] -= amount in PToken.transferFrom() perfoms an underflow check which is unnecessary because the function requires that allowances[from][msg.sender] >= amount
The arithmetic balances[from] -= amount in PToken.transferFrom() perfoms an underflow check which is unnecessary because the function requires that balances[from] >= amount
The arithmetic balances[recipient] += amount in PToken.transferFrom() perfoms an overflow check which is unnecessary because:

If from = recipient, then balances[recipient] + amount = (balances[from] - amount) + amount = balances[from] (the function requires that balances[from] >= amount). We know that balances[from] fits into uint and therefore balances[recipient] + amount also fits.
Otherwise, balances[recipient] + amount <= balances[recipient] + balances[from] <= totalBalances (the function requires that balances[from] >= amount). We know that totalBalances fits into uint and therefore balances[recipient] + amount also fits.

</td></tr>
<tr><td>Mitigation/Fix:</td><td>Save gas by surrounding these arithmetics with an unchecked { ... } block.</td></tr>
</table>

Severity: Recommendation

Issue:	Unnecessary checked arithmetics in PToken.claimSurplus()
Description:	• The arithmetic uint amount = currBalance - totalBalances in PToken.claimSurplus() perfoms an underflow check which is unnecessary because the function requires that currBalance > totalBalances • The arithmetic balances[who] += amount in PToken.claimSurplus() perfoms an overflow check which is unnecessary because the overflow check at totalBalances += amount guarantees that totalBalances + amount fits into uint and we know that balances[who] <= totalBalances.
Mitigation/Fix:	Save gas by surrounding these arithmetics with an unchecked { ... } block.

Severity: Recommendation

Issue:	Unnecessary checked arithmetics in PToken.doUnwrap()
Description:	• The arithmetic totalBalances -= amount in PToken.doUnwrap() perfoms an underflow check which is unnecessary because the function requires that balances[who] >= amount and we know that totalBalances >= balances[who]. • The arithmetic balances[who] -= amount in PToken.doUnwrap() perfoms an underflow check which is unnecessary because the function requires that balances[who] >= amount.
Mitigation/Fix:	Save gas by surrounding these arithmetics with an unchecked { ... } block.

Severity: Recommendation

Issue:	**Unnecessary checked arithmetic in Governance.convertReserves()**
Description:	The arithmetic assetCache.reserveBalance - uint96(amount) in Governance.convertReserves() perfoms an underflow check which is unnecessary because the function requires that amount <= assetStorage.reserveBalance.
Mitigation/Fix:	Save gas by surrounding this arithmetic with an unchecked { ... } block.