[Euler Finance

](https://docs.euler.finance/)was hacked for

[approximately $200 million

](https://forum.euler.finance/t/special-announcement/900)on March 13th, 2023 due to a vulnerability in their EToken smart contract.

This attack was made possible due to a missing check on the liquidity status of the account upon donating funds to the protocol coupled with the ability to use loans as self-collateral and Euler's dynamic liquidation penalty. This meant that the account was able to become insolvent, allowing the attacker to liquidate themself and steal the contract balance.

-- Learn how to spot these vulnerabilities yourself by following our

[step-by-step guide

](https://www.cyfrin.io/blog/how-to-become-a-smart-contract-auditor-courses-and-resources)to getting started as a smart contract auditor.

This article will explore how this attack worked and the steps taken using a proof of concept and observing the state changes

using a proof of concept. It will provide a comprehensive explanation of this attack, how it was introduced, and how it could have been avoided.

# Euler Finance -Background

Euler Finance is a non-custodial, permissionless lending protocol on the Ethereum blockchain. It allows users to utilize their cryptocurrency assets to earn interest or hedge against market volatility.

To understand how the hacker stole ~$200M, it is important to understand how Euler works:

- When users deposit assets into the Euler protocol, they receive ETokens representing their collateral

. As these deposited tokens earn interest, the value of the ETokens increases compared to the underlying asset Almost any asset can be deposited on Euler provided a UniswapV3 pool exists

- Deposit tokens and receive ETokens representing the collateral
- Users can mint up to 10 times their collateral value in ETokens to gain leverage

but doing so will also mint the user DTokens representing their debt

.

- Euler allows users to use the loan directly as collateral.

This is known as layered leverage

: newly minted EToken can be used as collateral to borrow additional assets. This amplifies the potential profits but also increases the risk (decreases the health score - defined shortly).

- Layered leverage - minting multiple times provided health score is high enough
- All loans are technically overcollateralized

to ensure the borrower can repay the loan plus the interest.

- The health score is used to determine how close the borrower is to liquidation.

This is calculated as a ratio of the maximum amount the user can borrow (the maximum loan-to-value (LTV) ratio) to the amount that they have borrowed (their current LTV):healthScore = MaxLTV / currentLTV

- Euler Maximum LTV Values:

(a) For regular loans (the collateral token and loaned token are distinct): 75%

(b) For self-collateralized loans (the collateral and loaned tokens are the same): 95%

- If the user's health score decreases below 1, they can be liquidated.

Usually, liquidations occur for a fixed penalty. However, Euler implemented a soft liquidation mechanism

, where ****the penalty starts at 0% and increases by 1% for every 0.1 decrease in health score. This means that liquidators take on the debt at a discount, equal to the penalty percentage.

- Euler employed a [maximum penalty of 20%

](https://docs.euler.finance/euler-protocol/eulers-default-parameters#maximum-liquidation-discount).

The hack drained six different tokens using the same method: DAI, stETH, WBTC, and USDC.

Let's go through how this was done step by step.

## Euler Finance Attack Steps

For context, this was the attacker's balance before the exploit:

Attacker balance before exploit: 0 DAI

1. Borrowed 30 million DAI

using a flash loan. A flash loan is executed by smart contracts and enables users to borrow funds without needing collateral. These loans must be repaid in full within the same transaction. If it is not, the entire transaction including the loan reverts.

1. The attacker deployed two contracts

:

[(a) The Violator

](https://etherscan.io/address/0x583c21631c48d442b5c0e605d624f54a0b366c72): to perform the attack using the flash loan.

[(b) The Liquidator

](https://etherscan.io/address/0x583c21631c48d442b5c0e605d624f54a0b366c72): to liquidate the Violator's account.

-- Now, using the Violator contract:

1. Deposited 20 million DAI

to Euler using the [EToken::deposit

function](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/EToken.sol#L139). The attacker received ~19.5 million ETokens representing the collateral.

After depositing (violator):

Collateral (eDAI): 19568124.414447288391400399

Debt (dDAI): 0.000000000000000000000000

1. Borrowed 195.6 million ETokens and 200 million DTokens

using the [EToken::mint

function](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/EToken.sol#L206-L229), which allows users to borrow up to 10 times their deposit amount. This meant that the borrowed-to-collateral ratio gave an LTV of 93% (remember that the maximum was 95% for self-collateralized loans) and a health score of 1.02

- a fully collateralized loan.

After minting (violator):

Collateral (eDAI): 215249368.558920172305404396

Debt (dDAI): 200000000.000000000000000000000000

Health score: 1.040263157894736842

1. Repaid 10 million DAI

to Euler using the [DToken::repay

function](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/DToken.sol#L126-L148) meaning that ~10 million ETokens were burned (leaving the EToken balance the same). This decreases the debt compared to the collateral, increasing the health score.

After repaying (violator):

Collateral (eDAI): 215249368.558920172305404396

Debt (dDAI): 190000000.000000000000000000000000

Health score: 1.089473684210526315

1. Borrowed 195.6 million eDAI and 200 million ETokens

using the EToken::mint

function again. This made the attacker's position more precarious by decreasing their health score. Increasing the amount borrowed also allowed the attacker to maximize their profits.

After minting (violator):

Collateral (eDAI): 410930612.703393056219408393

Debt (dDAI): 390000000.000000000000000000000000

Health score: 1.020647773279352226

1. Donated 100 million EToken

to Euler using the [EToken::donateToReserves

](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/EToken.sol#L359-L387) function.

Vulnerability: Lack of liquidity checks on the donateToReserves

function.

The donateToReserve()

function allows users to deposit funds into the reserve. Crucially, this function does not check that the user’s health score remained > 1 after the donation:

The Violator was able to force their self-collateralized leverage position to become under-collateralized by donating ETokens to the reserve. In other words, they donated collateral to the reserve that was being used to secure loans. Their DToken (debt) balance remained unchanged

, thus decreasing the health score and creating bad debt. If the Violator had not minted a second time, they would have had to donate more tokens to decrease their health score enough to enable a liquidation with the maximum 20% penalty (corresponding to a health score < 0.8). The hacker’s Liquidator contract could then successfully liquidate the Violator contract and withdraw from the protocol, earning the maximum 20% penalty.

After donating (violator):

Collateral (eDAI): 310930612.703393056219408393

Debt (dDAI): 390000000.000000000000000000000000

Health score: 0.750978643164551262

-- Now using the Liquidator Contract:

1. Checks the liquidation

using the [Liquidation::checkLiquidation

](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/Liquidation.sol#L178-L189) function to obtain the yield

and repay

values (corresponding to the collateral and debt transferred to the liquidator).

The following code snippet from the computeLiqOpp()

function, called in checkLiquidation()

, ensures that the amount of EToken to be sent to the liquidator does not exceed the borrower's available collateral. If the collateral does not satisfy the expected repayment yield, the remaining collateral is used by default. This means that liquidators only ever incur debt that equals the collateral they acquire at the discount dictated by the soft-liquidation mechanism

.

However, this check is based on the assumption that the violator's collateral can never be lower than the debt.

1. Liquidated the Violator

using the [Liquidation::liquidate

function](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/Liquidation.sol#L198-L220).

- The violator's health score dropped below 1

and the soft-liquidation mechanism was triggered.

- The yield cannot exceed the available collateral

and the discount equal to the penalty fee of 20% needs to be maintained, as enforced in computeLiqOpp()

. Since the Violator's EToken balance exceeded their DToken balance, the entire balance of ETokens was transferred to the Liquidator while a portion of the DTokens remained in the Violator's account. This ensured that the discount was applied and the health score of the Liquidator was maintained; however, this created bad debt that will never be repaid.

- This meant the Liquidator's profit entirely covered their debt

, as the value of the collateral obtained after liquidation was greater than the value of the debt. Thus, the liquidator could successfully withdraw the obtained funds without the need for any additional collateral.

- The Liquidator contract received 259 million DTokens and 311 million ETokens

from the Violator.

After liquidating (liquidator):

Collateral (eDAI): 310930612.703393056219408392

Debt (dDAI): 259319058.477209877830400000000000000

After liquidating (violator):

Collateral (eDAI): 0.000000000000000001

Debt (dDAI): 135765628.943911884480000000000000000

EULER balance: 38904507 DAI

1. Withdrew the liquidated funds

using [EToken::withdraw

](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/EToken.sol#L180-L200).

The exchange rate for the EToken to the underlying token was skewed due to the total borrows in the system being

artificially increased, meaning that the attacker could withdraw more DAI for their ETokens. As the attacker already had more ETokens than DTokens, the Liquidator was able to withdraw the total contract balance of ~38.9m DAI by burning ~38m of their ETokens.

After withdrawing (liquidator):

Collateral (eDAI): 272866200.699670845275982401 // The entire vault was drained - not enough funds in pool to withdraw fully

Debt (dDAI): 259319058.477209877830400000000000000

EULER balance: 0 DAI

1. Repaid the flash loan

using the profits (30 million DAI for the loan plus 27k DAI in interest) leaving ~8.88 million DAI profit.

Attacker balance after exploit: 8877507 DAI

1. Repeated the attack on other liquidity pools

, resulting in a net profit of $197 million.

### Causes

There are two critical reasons the attack was able to happen:

1. Lack of health check after donation

: Failure to check whether the user was in a state of undercollateralization after donating funds to the reserve address resulted in the soft liquidation mechanism being able to be triggered. This alongside self-collateralized layered leverage enabled the attacker to self-liquidate with the maximum penalty of 20%.

1. The Liquidator's profit exceeded their debt

: The Violator's health score dropped below 1 when the soft liquidation logic was triggered due to the balance of ETokens exceeding the balance of DTokens post-donation. Since the computeLiqOpp()

function ensured that the Liquidator's health score was above 1 while maintaining the 20% discount on the ETokens, bad debt was able to be locked in the Violator contract. This allowed the Liquidator's profit to entirely cover their debt, as the value of the collateral obtained after liquidation was greater than the value of the debt. Thus, the Liquidator could successfully extract the obtained funds without the need for any additional collateral.

This combination of factors enabled the attack to drain the contract's funds.

## Proof of Concept: Replicating the Hack

The following code, written using Foundry, is a proof of concept for this hack and recreates the steps described above:

The Violator

contract:

The Liquidator

contract:

The following MarketsView

contract extends the [Euler Markets

contract](https://github.com/euler-xyz/euler-contracts/blob/dfaa7788b17ac7c2a826a3ed242d7181998a778f/contracts/modules/Markets.sol) to expose private state variables.

-- A full proof of concept including all the interfaces used can be viewed on

[GitHub

](https://github.com/ciaranightingale/euler).

## How the bug was introduced

The donateToReserves()

function was added to the protocol as a remediation for a previous bug.

The exchange rate for how many ETokens a user receives for their underlying tokens is calculated as:

An attacker was able to exploit this by minting 1 wei of ETokens, then sending x

tokens to the protocol to artificially inflate the exchange rate (ETokenSupply

was small and underlyingBalance

was large, resulting in a large value for exchangeRate

).

This meant that the first lender received 0 ETokens due to floor rounding, meaning that the attacker was able to withdraw all the tokens (including the lender's) since they owned the total supply.

To mitigate this, Euler added an initial total supply of ETokens and a reserve of 1 million wei, meaning that the first lender contributed a minor amount of tokens to the reserve thereby making it an economically infeasible attack.

This remediation was effective for future ETokens but for existing ones where the underlying token reserve was < 1 million wei, a donateToReserves()

function was added to enable governance to increase the minimum reserve. This fixed the deposit bug but enabled the attacker to steal almost $200 million.

-- For an in-depth explanation of how this bug worked, refer to the

[Euler "Exchange Rate Manipulation" blog

](https://www.euler.finance/blog/exchange-rate-manipulation-in-erc4626-vaults).

## Lessons Learned & Key Takeaways

1. Invariant Testing:

This attack could have been avoided if the health score was tested post-donation - the core invariant being that the health score never goes below 1 unless the value of the underlying changes. New logic and functions added to an existing codebase, such as the donateToReserves()

function, should be thoroughly tested in the context of the entire protocol.

-- Learn more about how fuzz invariant testing can help to spot these vulnerabilities by

[reading this article

](https://www.cyfrin.io/blog/fuzz-invariant-tests).

1. Comprehensive Auditing

: multiple firms had previously audited Euler Finance; however, the donateToReserves()

function was only audited once. The protocol was audited again after the code change; however, the function was out of scope. Comprehensive audits are crucial to ensure that modification of the protocol does not create vulnerabilities in the context of the entire protocol. The donateToReserves()

function was never considered when used in the context of lenders, only for the use case where the EToken reserves needed to be increased.

## Summary

The absence of a health check in the donateToReserves()

function only posed a problem when combined with the implementation of soft (dynamic) liquidation.

The act of donating self-collateralized, layered leverage, reducing the health score below one, combined with the soft-liquidation mechanism, enabled the attacker to self-liquidate at the maximum 20% discount. This resulted in a significant profit for the attacker as the value of the debt was less than the collateral post-liquidation.

This attack could have been mitigated by employing invariant testing alongside an auditing process that considered code changes in the context of the wider protocol and from multiple different entry points.

Getting your protocol audited significantly decreases the probability of an attack like this happening.

- To learn smart contract security and development, visit

[Cyfrin Updraft

](https://updraft.cyfrin.io/).

- To request a security review for your smart contract,

[reach out to us here

](https://cyfrin.typeform.com/to/yDUg5DK3?typeform-source=0dwqu1zc3qs.typeform.com).

### References

This proof of concept in this article was adapted from the

[DefiHackLabs repository

](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Euler_exp.sol).

- Attack transaction
- Omniscia post-mortem
- Slowmist analysis