# Inflation Attack on ERC4626

Goal: Use Certora to verify that OpenZeppelin's implementation of ERC4626 is *not* vulnerable to the inflation attack.

## First Step

Expressed a CVL rule describing the inflation attack. Rule `vulnerableToInflationAttack` shows a counter-example on an ERC4626 implementation (not the OpenZeppelin one), that is vulnerable to the inflation attack.

Running the same rule on the OpenZeppelin implementation of ERC4626 results in a timeout.

## Approach for Mitigation of Timeouts

- Issue: rule for the inflation attack is quite complicated.
- Idea: simplify the process by trying to fix a timeout on another rule and get back to rule for the inflation attack and hope that timeout has been resolved there as well.
- New rule chosen: `inverseMintWithdrawInFavourForVault`. Using the rule the solver finds a CEX on the implementation not from OpenZeppelin but times out when verifying the OpenZeppelin implementation.
- Below is a set of trials to resolve the timeouts. All tests have been performed independently, i.e., state has been reset between runs to see individual impact.
- The order the trials are presented is not the order in which I tested them. Initially, I followed the dump page output and the coloring scheme to understand which methods are affecting the timeouts. `withdraw` and `mint` were labeled as complex due to the branching structure of `_update`. `mulDiv` and the underlying non-linear math wasn't the first candidate I looked at.

## Summarization (1 of 2)

The function `mulDiv` of the Math library uses a lot of assembly code. Assumption, if we can support prover in reasoning about it, we hope to get rid of timeouts.

- Replace the entire function by `return require_uint256(x*y/denominator)` → No Timeout
- Replace by upper and lower limits, it holds that `x*y/denominator <= res && x*y/denominator > res - 1` → No Timeout
- Further relaxation of `require_uint256(x*y/denominator)` as `x == 0 => res == 0  && x <= denominator` (using domain knowledge: `shares <= totalSupply()` or `assets <= totalAssets()` ) → No timeout, but CEX not valid, as rule too imprecise.

## Summarization (2 of 2)

- Summarize `_update` Function of ERC20. The complexity of the `_update` function (>3 diamonds and many calls from `transfer`, `mint`, `burn` and `transferFrom`). The function modifies storage (transformation on `balances` mapping) but does not return any result. → Timeout

## Using Ghosts

- For the rule we are expressing, we know that `convertToAssets` and `convertToShares` are both called exactly once. Using a ghost we can store the parameters to the first call and summarize the result of the second call using the stored value. We know that the following equation holds `assets >= convertToAssets(convertToShares(assets))`, i.e., when the second call is made we can use the ghosted first call's result to define an upper limit. This way we assume to avoid calling `mulDiv` in general, as the calls are intercepted. → No timeout, but CEX not valid.

## Concretizing Inputs

- Hardcode values for addresses. → Timeout
- Hardcode `shares` to one specific concrete value. → Timeout

## Munging

- Observation: The `_update` Function of ERC20 is central to the code and complex. It has 3+ diamonds and combines logic from `transfer`, `mint`, `burn` and `transferFrom` in one function. By in-lining the code, one can reduce code complexity for the solver. For instance, branches `if(from == address(0))` can be eliminated as of knowledge that `from != 0`. This requires inter-procedural (but not cross-contract) reasoning over path conditions. Does the static analysis eliminate these cases? → (Led to a Timeout. No link available - I can reproduce if required)
- Replacing `safeTransfer`:278 calls by `transfer`. `safeTransfer` is a method from library `SafeTransferLib` and uses assembly code to perform the operations. → (Led to a Timeout. No link available - I can reproduce if required)

## Experimenting with solver options

Using options

```
-smt_hashingScheme plainInjectivity -solvers [yices,z3]
```

→ Timeout

→ In the end, the summary `require_uint256(x*y/denominator)` solved the issue. The additional SMT option wasn't required anymore.

# Non-timeout related comments

## Limited expressiveness of CVL

- Expressiveness for summation is restricted. For ERC 4626, the equation `sumOfBalance == totalSupply` is not expressible. A complicated rule is needed to express the formula (TODO: think about ideas to simplify...)

$$totalSupply = \sum_{a \in uniqueAddresses} balanceOf(a)$$

where $uniqueAddresses$ is the set of addresses participating in the transaction.

Is it possible to express a set and containment in CVL?

## Learnings on rule writing process

- Setting up a GitHub Action significantly helps to keep track of changes in the process: One commit equals one Certora run.
- *Improvement to the GitHub Action*: Certora could report back to GitHub. Then it'll be easier to reason which commit led to an error (failed verfication/timeout). Could be easily done, for instance, using SARIF.

## Open Questions / Other comments

- How does a good structure of the rule writing process look like?
  → *Will probably learn it in the the project, I'll be starting this week.*
- During the process I frequently applied combinations of the above trials. My assumption was that it could have easily been the case, that only *in combination* they solve the timeout issue. During the process of rule writing, how can one judge the best approach to take to not waste time?
- When using summaries that are too weak, how to avoid counter-examples of non-interest? → *...most certainly hard to guide the solver into a certain direction.*

TODO: Get back to inflation attack. Run with basic `mulDivSummary` doesn't timeout. Inspect results. If the CEX is correct, open Zeppelin would be vulnerable to the attack. First observation (needs second inspection): The CEX doesn't work, the global setup is incorrect as property `ERC20.totalsupply == sumOfBalances` is violated. Must be hard-coded as it's not possible to use a `requireInvariant`.