dForce Lending Protocol Review

Consensys Diligence

Date	March 2021
Lead Auditor	Alexander Wade
Co-auditors	Heiko Fisch

1 Executive Summary

From March to April 2021, ConsenSys Diligence engaged with dForce to review the dForce Lending Protocol: a pool-based lending platform and stable debt protocol.

We conducted this assessment from March 8 to April 9, 2021, and allocated 8 person-weeks over that period.

1.1 Scope

Our review focused on the commit hash `419c2a1e00a74a0590a2e05a959a7c053de4181c`.

For the most part, the Solidity files at the given commit were in scope. However, one major component was NOT in scope: the pricing oracle implementation, `PriceOracle.sol`.

A complete list of files in scope can be found in the Appendix.

2 Security Specification

This section outlines the system's primary actors and roles, and describes some of the risks inherent to the protocol.

2.1 Actors

The system's non-user roles are summarized below with their respective abilities. This is not an exhaustive list; consult dForce's documentation for more information.

Note that the **Owner** role is present in most of the system's components. Although this role is configurable on a per-component basis, this section lists the Owner as a single actor for simplicity. We have confirmed with dForce that the role is intended to be held by a single account.

- **Owner**: The Owner acts as the system administrator, and has a broad range of permissions that allow them to configure many parts of the system.
  - **Upgrade management**: The Owner may upgrade most of the system's components.
    - By changing the implementation address used by those components' proxy contracts, the Owner may completely change the logic used for the following contracts: `Controller`, `MSDController`, `RewardDistributor`, `MSD`, `iToken`, `iETH`, `iMSD`, and `MSDS`.
    - By updating system-wide variables, the Owner may completely change the logic used for the following contracts: `PriceOracle` and `RewardDistributor`.
  - **Interest rate configuration**: The Owner may adjust parameters for many features of the system, including interest rate calculations, borrow rates, supply rates, and more.
  - **iToken configuration**: The Owner may list new iTokens and define their initial configuration. The Owner may also update this configuration at any time. Configurable parameters include:
    - *Collateral factor*: The ratio of a user's collateral that can be borrowed against.
    - *Borrow factor*: A multiplier on the value of a user's borrowed assets. A higher borrowed asset value will restrict the amount that can be borrowed.
    - *Borrow capacity*: A limit on the amount of underlying assets that may be borrowed.
    - *Supply capacity*: A limit on the amount of underlying assets that may be deposited.
  - **Liquidation configuration**: The Owner may update the *close factor*, which determines the ratio of borrowed assets a liquidator may repay during liquidation. Also, the Owner may update the *liquidation incentive*, which acts as a multiplier on the assets used to repay a borrower's position.
  - **Withdrawal of iToken reserves**: The Owner may withdraw a portion of any iToken's underlying asset. This portion is determined by the iToken's *reserve ratio*, which may also be configured by the Owner.
  - **Withdrawal of MSD reserves**: The Owner may mint themselves MSD tokens according to the relative levels of debt and equity in the MSD asset. Debt and equity are calculated using the asset's *supply rate* and *borrow rate*, both of which may be updated by the Owner.
  - **Pause and unpause**: The Owner may pause and unpause the protocol indefinitely. Additionally, the Owner may pause and unpause individual features on specific assets supported by the protocol. Essentially, the Owner has all the permissions of the **Pause Guardian** (see below), but may also **unpause** features.
- **Pause Guardian**: The Pause Guardian is able to pause the protocol as a whole, as well as individual features, or individual assets. Only the Owner may unpause things.
  - **Pause features for specific iTokens**:
    - *Pause mints*: Pause minting actions on a given iToken.
    - *Pause redeems*: Pause redemptions for an iToken's underlying assets. This applies to both `redeem` and `redeemUnderlying`.
    - *Pause borrows*: Pause borrows on a given iToken.
  - **Pause features globally**:
    - *Pause seizes*: Pause all liquidation actions. This applies to both `liquidateBorrow` and `seize`.
    - *Pause transfers*: Pause all token transfers. This applies to `transfer` and `transferFrom` for all supported iTokens.

2.2 Risks

This section describes some of the risks inherent to the dForce lending protocol.

- **System Owner role may be a single point of failure.** The Owner role has permissions to execute a wide variety of actions in virtually every component of the system. The breadth and depth of the authority held by the Owner makes this role a target for attack by both malicious insiders and external parties. In the event the Owner role is compromised, an attacker would easily be able to drain user funds.

Note: dForce informed us that they intend to transfer the Owner role to a multisig or DAO at some point in the future.

- **Unpredictability due to front running or timing.** Actions taken by the Owner role do not come with a delay. From a user's perspective, upgrades and updates may come without warning. This means that the Owner could use front running to make malicious changes just ahead of incoming transactions. Alternatively, well intentioned changes may result in negative effects for users due to unfortunate timing.

  For example, the Owner may choose to update the collateral factor for an iToken. Whether intentionally or not, this update may be performed just prior to a user's "borrow" transaction, and may result in their taking a position with less collateral value than expected. This example may be extended to every Owner permission outlined above (see: 2.1 Actors).

  In general, users of the system can't be sure what the behavior of a function call will be, because behavior can change at any time.

  There are two broad strategies for addressing this:

  - *Let the user lock things down from changing.* For example, allow them to specify what version of a module or contract they expect, and if that version is no longer current, revert.
  - *Use a time lock to give users advance notice of changes.* For example, ensure that upgrades and updates require two steps with a mandatory delay between them. This ensures that changes to system configuration are broadcasted well in advance, and allows users to react in time.

- **Some loans can't be liquidated with a profit.** Loans that become under-collateralized, which usually happens as a result of price movements, can be partially liquidated. That means a liquidator can repay a fraction of the loan and gets an equivalent part of the loan's collateral in return, plus a reward for their service. More specifically, the globally set `closeFactor` determines the maximum fraction of a loan that can be liquidated, and the amount of collateral given to the liquidator is `liquidationIncentive * v`, where `liquidationIncentive` is also globally set and `v` denotes the current value of the repaid amount.

  Executing a liquidation requires a considerable amount of gas; how much exactly depends on several factors, most notably the number of loans taken and the number of assets used as collateral. (See also issue 4.3.) For a particular liquidation to be profitable, the money spent on gas plus the value `v` of the repaid amount must be smaller than `liquidationIncentive * v`. Since `v` can't be greater than `collateralFactor * l`, where `l` denotes the value of the loan, a liquidator must spend less than `collateralFactor * l * (liquidationIncentive - 1)` to be profitable. If, for example, we assume `collateralFactor = 0.5`, `liquidationIncentive = 1.05` and a loan worth $4000, the maximum liquidation amount is $2000 and the liquidator's budget for gas is $100.

  Given that liquidations are (or can be) quite expensive in terms of gas consumption, in times of high gas prices even fairly sizable loans, possibly worth several thousand dollars, can't be liquidated with a profit. From a borrower's perspective, this means that relatively large loans can be taken that will only be liquidated if someone is willing to take a loss. To make things worse, a borrower can drive up the amount of gas needed for a liquidation of their loan by adding more collateral and/or borrow positions with possibly small value.

  Possible mitigations include increasing the parameters `closeFactor` and/or `liquidationIncentive`, which can be done by the Owner. However, this affects *all* under-collateralized loans, also ones that could be liquidated with a profit even without the parameter change. So it might be worth integrating a rising `closeFactor` and/or `liquidationIncentive` for smaller loans into the protocol; this also eliminates the need for an Owner intervention and makes the system's behavior therefore more predictable.
  Nevertheless, these measures can only be a mitigation, not a complete solution; they "move the numbers" but, fundamentally, the problem remains: If the collateral is not even sufficient to cover the gas costs of a liquidation, no parameter adjustments can help.

  We discussed this with dForce, and their plan (apart from possible parameter changes) is to liquidate any such position themselves and take the loss. If there aren't too many of these loans, that will probably work; in fact, the very announcement to do so might actually prevent that a substantial amount of such loans is taken in the first place because their appeal would lie in a low(er) risk of being liquidated in case of under-collateralization. On the other hand, *if* a huge amount of such loans is taken nevertheless — for example because many users believe dForce won't or won't even be able to follow through with liquidating all of them — that could become a self-fulfilling prophecy, leading to many under-collateralized loans in the system that no one is willing to liquidate.

  We would therefore recommend investigating modifications of the system to more robustly deal with situations as outlined above, although that might prove to be a challenging task. Possible building blocks could be close factors and liquidation incentives that dynamically adapt to the loan size, as outlined above. In addition to that, it would probably be necessary to prevent "small" loans (or positions in general), both initially and as a result of other operations on a bigger loan. Situations as described in the next item might add further complications.

- **Risk of uncollateralized loans after a flash crash.** As already mentioned above, under-collateralized loans can only be liquidated partially. Normally, such a partial liquidation brings the *remaining* loan back in — or at least closer to — the sufficiently collateralized range, i.e. the collateral / debt ratio is greater than it was before the liquidation. However, there are situations in which a partial liquidation *decreases* the collateral / debt ratio, for example if, even before the liquidation, the debt exceeds the collateral. While that's precisely the situation liquidations try to avoid in the first place, it might happen nevertheless, for example if the collateral price falls very quickly. Since a liquidation *at this point* decreases the collateral / debt ratio, the remaining loan can immediately be liquidated again, further decreasing said ratio, and so on. In the end, all or most of the collateral will be gone, but some of the loan will not have been repaid — and no one has an incentive to change that. Moreover, this essentially uncollateralized loan will accrue interest and grow.

  The dForce team had already been aware of this and informed us that they're willing to repay such loans in order to keep the system healthy. While this is clearly not an ideal solution, this issue seems to be inherent to the fundamental system design, and it seems unlikely that it can be avoided completely.

3 Recommendations

3.1 Ensure users have a clear understanding of what "Collateral" means

Description

Traditionally, collateral refers to: "an asset pledged as security for the repayment of a loan, to be forfeited in the event of a default."

Using dForce, users may deposit assets into the system and borrow assets from the system. When borrowing assets, the system calculates the amount that can be borrowed as a function of the value of the user's collateral. Users must explicitly mark deposited assets as collateral via the `Controller.enterMarkets` method, which adds the asset to the user's "collaterals" list:

code/contracts/Controller.sol:L1397-L1409

```
function _enterMarket(address _iToken, address _account)
    internal
    returns (bool)
{
    // Market not listed, skip it
    if (!iTokens.contains(_iToken)) {
        return false;
    }

    // add() will return false if iToken is in account's market list
    if (accountsData[_account].collaterals.add(_iToken)) {
        emit MarketEntered(_iToken, _account);
    }
```

Users may deposit and hold assets without marking them as collateral via this method. Using the traditional definition of "collateral," a user may expect that assets not marked as collateral cannot be seized in the event they default on a loan. However, this is not the case: ANY asset deposited by the user can be seized during liquidation, regardless of whether it was marked collateral or not.

Recommendation

Given this subversion of expectations, we recommend ensuring that dForce's users have a clear understanding of their risks and responsibilities when they deposit assets into the lending platform.

- Consider revisiting the term "collateral" to apply to all assets deposited into the system.
- Consider creating user-facing documentation that clearly outlines the meaning of the term.

3.2 Short-circuit `Base._updateInterest` by returning early if `accrualBlockNumber == block.number`

Resolution

This recommendation was implemented in commit `37205c6`.

Description

`Base._updateInterest` is executed before most operations in the dForce system. The method accumulates interest from borrows since the last time the method was called, and adds a portion to the contract's reserves. It then updates these values in contract state, ensuring the action being taken is using the most up-to-date values:

code/contracts/TokenBase/Base.sol:L140-L144

```
// Writes the previously calculated values into storage.
accrualBlockNumber = _vars.currentBlockNumber;
borrowIndex = _vars.newBorrowIndex;
totalBorrows = _vars.newTotalBorrows;
totalReserves = _vars.newTotalReserves;
```

`_updateInterest` is relatively long, and will likely be called several times per block. In this case, the `blockDelta` used to calculate accumulated interest will result in a calculated value of "0" interest accumulated:

code/contracts/TokenBase/Base.sol:L109-L126

```
// Records the current block number.
_vars.currentBlockNumber = block.number;

// Calculates the number of blocks elapsed since the last accrual.
_vars.blockDelta = _vars.currentBlockNumber.sub(accrualBlockNumber);

/**
 * Calculates the interest accumulated into borrows and reserves and the new index:
 *  simpleInterestFactor = borrowRate * blockDelta
 *  interestAccumulated = simpleInterestFactor * totalBorrows
 *  newTotalBorrows = interestAccumulated + totalBorrows
 *  newTotalReserves = interestAccumulated * reserveFactor + totalReserves
 *  newBorrowIndex = simpleInterestFactor * borrowIndex + borrowIndex
 */
_vars.simpleInterestFactor = _vars.borrowRate.mul(_vars.blockDelta);
_vars.interestAccumulated = _vars.simpleInterestFactor.rmul(
    _vars.totalBorrows
);
```

When no interest has been accumulated, the method's state changes have no net effect.

Recommendation

In order to save gas on repeated calls in the same block, `_updateInterest` should return early if `accrualBlockNumber == block.number`.
Note that the function currently emits an `UpdateInterest` event even if this is not the first call in this block. It might be worth mentioning that returning early if `accrualBlockNumber == block.number` will change that behavior — which is probably a good thing since nothing has been updated anyway.

3.3 Plan and test the Owner in real-world scenarios, including an eventual transition to a smart contract

Description

issue 4.5 describes a requirement in `RewardDistributor` that conflicts with dForce's plans to transition the Owner role to a smart contract. This finding suggests that this eventual transition has not been sufficiently planned, and is wholly untested.

Given that this is an important milestone for the protocol, it is important to plan for its execution well in advance. Whether the Owner role will be held by an EOA, a multisig, or a DAO, the capabilities of the Owner are very important to the system as a whole.

Examples

The following examples outline some basic design considerations dForce should plan for now, ahead of this eventual transition:

- **The Owner needs to be able to execute multiple actions atomically.** This is primarily important because there will be situations where multiple function calls are required to safely carry out a change.
  - For example, in the event the `MSDController` needs to be updated, calls to `iMSD._setMSDController` and `MSDS._setMSDController` need to happen atomically. Without atomic execution of both methods, a window of time exists where transactions may interact with one or both of these contracts in a half-configured state.
  - As another example, if an `InterestRateModel` is found to be faulty, a complete upgrade requires calling each iToken individually (`iToken._setInterestRateModel`).
- **Owner actions should be timelocked, in order to remove unpredictability for users.** As explained in 2.2 Risks, instant configuration changes mean users cannot be sure what the behavior of a function call will be, as this behavior can change at any time. Transitioning the Owner role to a smart contract enables the possibility of implementing time-locked actions, where a mandatory delay ensures users are able to react to pending Owner actions in time.

Recommendation

- Extend current testing of the Owner role with tests where the Owner is replaced by a smart contract, preferably one capable of batching actions.
- Come up with real-world scenarios where Owner actions are needed, then test these scenarios. An easy way to come up with these scenarios is to consider cases where a bug is discovered in one or more of the system's components. What actions should the Owner take in these scenarios? Can those actions be taken safely using both an EOA and a smart contract?

3.4 Avoid code duplication

Description

There are several instances of duplicated code throughout the codebase. This should generally be avoided as it reduces maintainability and readability of the source code, increases source code length, and might increase bytecode length.

Examples

1. There are four interest rate models; three of them employ the asset's utilization rate and define the exact same function for its computation. In issue 4.3, we suggest modifications to this function; currently, they would have to be applied to all three instances of this code.
2. All four models contain the following function, which —although trivial and unlikely to change — would better be placed in a base contract all interest rate models inherit from:

code/contracts/InterestRateModel/InterestRateModel.sol:L65-L70

```
/**
 * @notice Ensure this is an interest rate model contract.
 */
function isInterestRateModel() external pure returns (bool) {
    return true;
}
```

Recommendation

Well-known techniques like inheritance and use of libraries help avoid code duplication.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 `iETH.exchangeRateStored` may not be accurate when invoked from external contracts Major

Resolution

This issue was addressed in commit `9876e3a` by using a modifier to track the current `msg.value` of payable functions.

Description

`iETH.exchangeRateStored` returns the exchange rate of the contract as a function of the current cash of the contract. In the case of `iETH`, current cash is calculated as the contract's ETH balance minus `msg.value`:

code/contracts/iETH.sol:L54-L59

```
/**
 * @dev Gets balance of this contract in terms of the underlying
 */
function _getCurrentCash() internal view override returns (uint256) {
    return address(this).balance.sub(msg.value);
}
```

`msg.value` is subtracted because the majority of `iETH` methods are payable, and `msg.value` is implicitly added to a contract's balance before execution begins. If `msg.value` were not subtracted, the value sent with a call could be used to inflate the contract's exchange rate artificially.

As part of execution, `iETH` makes calls to the `Controller`, which performs important checks using (among other things) the stored exchange rate. When `exchangeRateStored` is invoked from the `Controller`, the call context has a `msg.value` of 0. However, the `msg.value` sent by the initial `iETH` execution is still included in the contract's balance. This means that the `Controller` receives an exchange rate inflated by the initial call's `msg.value`.

Examples

This problem occurs in multiple locations in the `Controller`:

- `beforeMint` uses the exchange rate to ensure the supply capacity of the market is not reached. In this case, inflation would prevent the entire supply capacity of the market from being utilized:

code/contracts/Controller.sol:L670-L678

```
// Check the iToken's supply capacity, -1 means no limit
uint256 _totalSupplyUnderlying =
    IERC20Upgradeable(_iToken).totalSupply().rmul(
        IiToken(_iToken).exchangeRateStored()
    );
require(
    _totalSupplyUnderlying.add(_mintAmount) <= _market.supplyCapacity,
    "Token supply capacity reached"
);
```

- `beforeLiquidateBorrow` uses the exchange rate via `calcAccountEquity` to calculate the value of the borrower's collateral. In this case, inflation would increase the account's equity, which could prevent the liquidator from liquidating:

code/contracts/Controller.sol:L917-L919

```
(, uint256 _shortfall, , ) = calcAccountEquity(_borrower);

require(_shortfall > 0, "Account does not have shortfall");
```

Recommendation

- Rather than having the `Controller` query the `iETH.exchangeRateStored`, the exchange rate could be passed-in to `Controller` methods as a parameter.
- Ensure no other components in the system rely on `iETH.exchangeRateStored` after being called from `iETH`.

4.2 Unbounded loop in `Controller.calcAccountEquity` allows DoS on liquidation Major

Description

`Controller.calcAccountEquity` calculates the relative value of a user's supplied collateral and their active borrow positions. Users may mark an arbitrary number of assets as collateral, and may borrow from an arbitrary number of assets. In order to calculate the value of both of these positions, this method performs two loops.

First, to calculate the sum of the value of a user's collateral:

code/contracts/Controller.sol:L1227-L1233

```
// Calculate value of all collaterals
// collateralValuePerToken = underlyingPrice * exchangeRate * collateralFactor
// collateralValue = balance * collateralValuePerToken
// sumCollateral += collateralValue
uint256 _len = _accountData.collaterals.length();
for (uint256 i = 0; i < _len; i++) {
    IiToken _token = IiToken(_accountData.collaterals.at(i));
```

Second, to calculate the sum of the value of a user's borrow positions:

code/contracts/Controller.sol:L1263-L1268

```
// Calculate all borrowed value
// borrowValue = underlyingPrice * underlyingBorrowed / borrowFactor
// sumBorrowed += borrowValue
_len = _accountData.borrowed.length();
for (uint256 i = 0; i < _len; i++) {
    IiToken _token = IiToken(_accountData.borrowed.at(i));
```

From dForce, we learned that 200 or more assets would be supported by the Controller. This means that a user with active collateral and borrow positions on all 200 supported assets could force any `calcAccountEquity` action to perform some 400 iterations of these loops, each with several expensive external calls.

Examples

By modifying dForce's unit test suite, we showed that an attacker could force the cost of `calcAccountEquity` above the block gas limit. This would prevent all of the following actions, as each relies on `calcAccountEquity`:

- `iToken.transfer` and `iToken.transferFrom`
- `iToken.redeem` and `iToken.redeemUnderlying`
- `iToken.borrow`
- `iToken.liquidateBorrow` and `iToken.seize`

The following actions would still be possible:

- `iToken.mint`
- `iToken.repayBorrow` and `iToken.repayBorrowBehalf`

As a result, an attacker may abuse the unbounded looping in `calcAccountEquity` to prevent the liquidation of underwater positions. We provided dForce with a PoC here: gist.

Recommendation

There are many possible ways to address this issue. Some ideas have been outlined below, and it may be that a combination of these ideas is the best approach:

In general, **cap the number of markets and borrowed assets a user may have**: The primary cause of the DoS is that the number of collateral and borrow positions held by a user is only restricted by the number of supported assets. The PoC provided above showed that somewhere around 150 collateral positions and 150 borrow positions, the gas costs of `calcAccountEquity` use most of the gas in a block. Given that gas prices often spike along with turbulent market conditions and that liquidations are far more likely in turbulent market conditions, a cap on active markets / borrows should be much lower than 150 each so as to keep the cost of liquidations as low as possible.

dForce should perform their own gas cost estimates to determine a cap, and choose a safe, low value. Estimates should be performed on the high-level `liquidateBorrow` method, so as to simulate an actual liquidation event. Additionally, estimates should factor in a changing block gas limit, and the possibility of opcode gas costs changing in future forks. It may be wise to make this cap configurable, so that the limits may be adjusted for future conditions.

4.3 Fix utilization rate computation and respect reserves when lending Medium

Resolution

The dForce team has informed us that the only two interest rate models that are still in use are `StablecoinInterestRateModel` and `StandardInterestRateModel`. For these, recommendation 2 has been addressed in commits 2a0e974 and c11fa9b.

Description

The utilization rate `UR` of an asset forms the basis for interest calculations and is defined as `borrows / ( borrows + cash - reserves)`.

code/contracts/InterestRateModel/InterestRateModel.sol:L72-L88

```
/**
 * @notice Calculate the utilization rate: `_borrows / (_cash + _borrows - _reserves)`
 * @param _cash Asset balance
 * @param _borrows Asset borrows
 * @param _reserves Asset reserves
 * @return Asset utilization [0, 1e18]
 */
function utilizationRate(
    uint256 _cash,
    uint256 _borrows,
    uint256 _reserves
) internal pure returns (uint256) {
    // Utilization rate is 0 when there are no borrows
    if (_borrows == 0) return 0;

    return _borrows.mul(BASE).div(_cash.add(_borrows).sub(_reserves));
}
```

The implicit assumption here is that `reserves <= cash`; in this case — and if we define `UR` as `0` for `borrows == 0` — we have `0 <= UR <=1`. We can view `cash - reserves` as "available cash". However, the system does not guarantee that `reserves` never exceeds `cash`. If `reserves > cash` (and `borrows + cash - reserves > 0`), the formula for `UR` above gives a utilization rate above `1`. This doesn't make much sense conceptually and has undesirable technical consequences; an especially severe one is analyzed in issue 4.4.

Recommendation

If `reserves > cash` — or, in other words, available cash is negative — this means part of the reserves have been borrowed, which ideally shouldn't happen in the first place. However, the `reserves` grow automatically over time, so it might be difficult to avoid this entirely. We recommend (1) avoiding this situation whenever it is possible and (2) fixing the `UR` computation such that it deals more gracefully with this scenario. More specifically:

1. Loan amounts should not be checked to be smaller than or equal to `cash` but `cash - reserves` (which might be negative). Note that the current check against `cash` happens more or less implicitly because the transfer just fails for insufficient `cash`.
2. Make the utilization rate computation return `1` if `reserves > cash` (unless `borrows == 0`, in which case return `0` as is already the case).

Remark

Internally, the utilization rate and other fractional values are scaled by `1e18`. The discussion above has a more conceptual than technical perspective, so we used unscaled numbers. When making changes to the code, care must be taken to apply the scaling.

4.4 If `Base._updateInterest` fails, the entire system will halt Medium

Resolution

dForce removed `settleInterest` from `TokenAdmin._setInterestRateModel` and `MSDS._setInterestRateModel` in commit `27f9a28`.

Description

Before executing most methods, the `iETH` and `iToken` contracts update interest accumulated on borrows via the method `Base._updateInterest`. This method uses the contract's interest rate model to calculate the borrow interest rate. If the calculated value is above `maxBorrowRate` (`0.001e18`), the method will revert:

code/contracts/TokenBase/Base.sol:L92-L107

```
function _updateInterest() internal virtual override {
    InterestLocalVars memory _vars;
    _vars.currentCash = _getCurrentCash();
    _vars.totalBorrows = totalBorrows;
    _vars.totalReserves = totalReserves;

    // Gets the current borrow interest rate.
    _vars.borrowRate = interestRateModel.getBorrowRate(
        _vars.currentCash,
        _vars.totalBorrows,
        _vars.totalReserves
    );
    require(
        _vars.borrowRate <= maxBorrowRate,
        "_updateInterest: Borrow rate is too high!"
    );
```

If this method reverts, the entire contract may halt and be unrecoverable. The only ways to change the values used to calculate this interest rate lie in methods that must first call `Base._updateInterest`. In this case, those methods would fail.

One other potential avenue for recovery exists: the Owner role may update the interest rate calculation contract via `TokenAdmin._setInterestRateModel`:

code/contracts/TokenBase/TokenAdmin.sol:L46-L63

```
/**
 * @dev Sets a new interest rate model.
 * @param _newInterestRateModel The new interest rate model.
 */
function _setInterestRateModel(
    IInterestRateModelInterface _newInterestRateModel
) external virtual onlyOwner settleInterest {
    // Gets current interest rate model.
    IInterestRateModelInterface _oldInterestRateModel = interestRateModel;

    // Ensures the input address is the interest model contract.
    require(
        _newInterestRateModel.isInterestRateModel(),
        "_setInterestRateModel: This is not the rate model contract!"
    );

    // Set to the new interest rate model.
    interestRateModel = _newInterestRateModel;
```

However, this method also calls `Base._updateInterest` before completing the upgrade, so it would fail as well.

Examples

We used interest rate parameters taken from dForce's unit tests to determine whether any of the interest rate models could return a borrow rate that would cause this failure. The default `InterestRateModel` is deployed using these values:

```
baseInterestPerBlock: 0
interestPerBlock: 5.074e10
highInterestPerBlock: 4.756e11
high: 0.75e18
```

Plugging these values in to their borrow rate calculations, we determined that the utilization rate of the contract would need to be `2103e18` in order to reach the max borrow rate and trigger a failure. Plugging this in to the formula for utilization rate, we derived the following ratio:

`reserves >= (2102/2103)*borrows + cash`

With the given interest rate parameters, if token reserves, total borrows, and underlying cash meet the above ratio, the interest rate model would return a borrow rate above the maximum, leading to the failure conditions described above.

Recommendation

Note that the examples above depend on the specific interest rate parameters configured by dForce. In general, with reasonable interest rate parameters and a reasonable reserve ratio, it seems unlikely that the maximum borrow rate will be reached. Consider implementing the following changes as a precaution:

- As utilization rate should be between `0` and `1` (scaled by `1e18`), prevent utilization rate calculations from returning anything above `1e18`. See issue 4.3 for a more thorough discussion of this topic.
- Remove the `settleInterest` modifier from `TokenAdmin._setInterestRateModel`: In a worst case scenario, this will allow the Owner role to update the interest rate model without triggering the failure in `Base._updateInterest`.

4.5 `RewardDistributor` requirement prevents transition of Owner role to smart contract Medium

Resolution

This issue was addressed in commit `4f1e31b` by invoking `_updateDistributionSpeed` directly.

Description

From dForce, we learned that the eventual plan for the system Owner role is to use a smart contract (a multisig or DAO). However, a requirement in `RewardDistributor` would prevent the `onlyOwner` method `_setDistributionFactors` from working in this case.

`_setDistributionFactors` calls `updateDistributionSpeed`, which requires that the caller is an EOA:

code/contracts/RewardDistributor.sol:L179-L189

```
/**
 * @notice Update each iToken's distribution speed according to current global speed
 * @dev Only EOA can call this function
 */
function updateDistributionSpeed() public override {
    require(msg.sender == tx.origin, "only EOA can update speeds");
    require(!paused, "Can not update speeds when paused");

    // Do the actual update
    _updateDistributionSpeed();
}
```

In the event the Owner role is a smart contract, this statement would necessitate a complicated upgrade to restore full functionality.

Recommendation

Rather than invoking `updateDistributionSpeed`, have `_setDistributionFactors` directly call the internal helper `_updateDistributionSpeed`, which does not require the caller is an EOA.

4.6 `MSDController._withdrawReserves` does not update interest before withdrawal Medium

Resolution

This issue was addressed in commit `2b5946e` by changing `calcEquity` to update the interest of each MSDMinter assigned to an MSD asset.

Note that this method iterates over each MSDMinter, which may cause out-of-gas issues if the number of MSDMinters grows. dForce has informed us that the MSDMinter role will only be held by two contracts per asset (`iMSD` and `MSDS`).

Description

`MSDController._withdrawReserves` allows the Owner to mint the difference between an MSD asset’s accumulated debt and earnings:

code/contracts/msd/MSDController.sol:L182-L195

```
function _withdrawReserves(address _token, uint256 _amount)
    external
    onlyOwner
    onlyMSD(_token)
{
    (uint256 _equity, ) = calcEquity(_token);

    require(_equity >= _amount, "Token do not have enough reserve");

    // Increase the token debt
    msdTokenData[_token].debt = msdTokenData[_token].debt.add(_amount);

    // Directly mint the token to owner
    MSD(_token).mint(owner, _amount);
```

Debt and earnings are updated each time the asset’s `iMSD` and `MSDS` contracts are used for the first time in a given block. Because `_withdrawReserves` does not force an update to these values, it is possible for the withdrawal amount to be calculated using stale values.

Recommendation

Ensure `_withdrawReserves` invokes `iMSD.updateInterest()` and `MSDS.updateInterest()`.

4.7 `permit` functions use deployment-time instead of execution-time chain ID Minor

Resolution

This has been addressed in commits a7b8fb0 and d659f2b. The approach taken by the dForce team is to include the chain ID separately in the digest to be signed and keep the deployment/initialization-time chain ID in the `DOMAIN_SEPARATOR`. This avoids recomputing the `DOMAIN_SEPARATOR` in the event of a chain split and it continues to work on the new chain; the downside is that now there are two chain IDs in the data to be signed — and after a chain split, they are even different on the new chain — which might be confusing for the signer.

Description

The contracts `Base`, `MSD`, and `MSDS` each have an EIP-2612-style `permit` function that supports approvals with EIP-712 signatures. We focus this discussion on the `Base` contract, but the same applies to `MSD` and `MSDS`.
When the contract is initialized, the chain ID is queried (with the `CHAINID` opcode) and becomes part of the `DOMAIN_SEPARATOR` — a hash of several values which (presumably) don’t change over the lifetime of the contract and that can therefore be computed only once, when the contract is deployed.

code/contracts/TokenBase/Base.sol:L23-L56

```
function _initialize(
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    IControllerInterface _controller,
    IInterestRateModelInterface _interestRateModel
) internal virtual {
    controller = _controller;
    interestRateModel = _interestRateModel;
    accrualBlockNumber = block.number;
    borrowIndex = BASE;
    flashloanFeeRatio = 0.0008e18;
    protocolFeeRatio = 0.25e18;
    __Ownable_init();
    __ERC20_init(_name, _symbol, _decimals);
    __ReentrancyGuard_init();

    uint256 chainId;

    assembly {
        chainId := chainid()
    }
    DOMAIN_SEPARATOR = keccak256(
        abi.encode(
            keccak256(
                "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
            ),
            keccak256(bytes(_name)),
            keccak256(bytes("1")),
            chainId,
            address(this)
        )
    );
}
```

The `DOMAIN_SEPARATOR` is supposed to prevent replay attacks by providing context for the signature; it is hashed into the digest to be signed.

code/contracts/TokenBase/Base.sol:L589-L610

```
bytes32 _digest =
    keccak256(
        abi.encodePacked(
            "\x19\x01",
            DOMAIN_SEPARATOR,
            keccak256(
                abi.encode(
                    PERMIT_TYPEHASH,
                    _owner,
                    _spender,
                    _value,
                    _currentNonce,
                    _deadline
                )
            )
        )
    );
address _recoveredAddress = ecrecover(_digest, _v, _r, _s);
require(
    _recoveredAddress != address(0) && _recoveredAddress == _owner,
    "permit: INVALID_SIGNATURE!"
);
```

The chain ID is not necessarily constant, though. In the event of a chain split, only one of the resulting chains gets to keep the original chain ID and the other will have to use a new one. With the current pattern, a signature will be valid on both chains; if the `DOMAIN_SEPARATOR` is recomputed for every verification, a signature will only be valid on the chain that keeps the original ID — which is probably the intended behavior.

Remark

The reason why the not necessarily constant chain ID is part of the supposedly constant `DOMAIN_SEPARATOR` is that EIP-712 predates the introduction of the `CHAINID` opcode. Originally, it was not possible to query the chain ID via opcode, so it had to be supplied to the constructor of a contract by the deployment script.

Recommendation

An obvious fix is to compute the `DOMAIN_SEPARATOR` dynamically in `permit`. However, since a chain split is a relatively unlikely event, it makes sense to compute the `DOMAIN_SEPARATOR` at deployment/initialization time and then check in `permit` whether the current chain ID equals the one that went into the `DOMAIN_SEPARATOR`. If that is true, we proceed as before. If the chain ID has changed, we could (1) just revert, or (2) recompute the `DOMAIN_SEPARATOR` with the new chain ID. Solution (1) is probably the easiest and most straightforward to implement, but it should be noted that it makes the permit functionality of this contract completely unusable on the new chain.

4.8 `iETH.receive()` does not support contracts executing during their constructor Minor

Description

`iETH.receive()` requires that the caller is a contract:

code/contracts/iETH.sol:L187-L195

```
/**
 * @notice receive ETH, used for flashloan repay.
 */
receive() external payable {
    require(
        msg.sender.isContract(),
        "receive: Only can call from a contract!"
    );
}
```

This method uses the `extcodesize` of an account to check that the account belongs to a contract. However, contracts currently executing their constructor will have an `extcodesize` of 0, and will not be able to use this method.

This is unlikely to cause significant issues, but dForce may want to consider supporting this edge case.

Recommendation

Use `msg.sender != tx.origin` as a more reliable method to detect use by a contract.

Appendix 1 - Files in Scope

This review concerned the following files:

File	SHA-1 hash
Controller.sol	c783e56d9e25ffedccae55ef8204b2645739ad5f
iETH.sol	2c9951228681f4fea21b3b655f21a68ad6325388
InterestRateModel/FixedInterestRateModel.sol	f8a269a6a2c8ba7045725589c3155b7c21cbd320
InterestRateModel/InterestRateModel.sol	fc26dca35f66adaf5045f0bc88346895cf86619d
InterestRateModel/StablecoinInterestRateModel.sol	3638eb2877938b8110f401d8bdae0ede45316e11
InterestRateModel/StandardInterestRateModel.sol	444104e36cc18075276c07e817caff3e934b51de
interface/IControllerInterface.sol	21bb0c988afb73cdccc0a2ee50a790c66de0b12a
interface/IFlashloanExecutor.sol	5c746cada49fdd9b083bdef516b69d23d1d44476
interface/IInterestRateModelInterface.sol	c04af354e71017ed964866db8a480d15ed1b71a8
interface/IiToken.sol	19907f97bd417036dc7d1125d5948afda8221432
interface/IPriceOracle.sol	eb6879315cb735402b95b529161609368ed654c6
interface/IRewardDistributor.sol	a9752e52973d3d7158e700886b588ee6229833d6
iToken.sol	4d3ac05d13a75d406b929c97f0c9e249cd840faf
library/ERC20.sol	324800bf529a093aff54b6bbbd921c3df8c50edc
library/Initializable.sol	8f29e0749469160237386e85121fd0306cb83464
library/Ownable.sol	68c38e20cfe7ab0de0a119590f3224e591658224
library/ProxyAdmin.sol	21c6851d1d682425144ddb46be5725aa3b2cde08
library/ReentrancyGuard.sol	e70aefbceeab591323c47e601f6b90958fc55906
library/SafeRatioMath.sol	3c22a2b782b225fa168e0653c50eb660f46f994a
msd/iMSD.sol	669e0edcf804373510c3a88b01a481121864bdd6
msd/MSDController.sol	55b7b51b91a1696533bbea2c59630628c77d486a
msd/MSD.sol	dc138f4e86bc21693bdaa8f37464fbf3ee7f36ad
msd/MSDS.sol	304da929d8ddf77429b48031e575838437ff9e18
RewardDistributor.sol	33801ccd521d6386d3cd32485f7f065fa18f5357
TokenBase/Base.sol	52b1e0a5f0379f1acf50ebc8a19a02e4fb0af6bd
TokenBase/TokenAdmin.sol	b7dae95a7c244d1b9bc9d2598676eb2d70f65fc7
TokenBase/TokenERC20.sol	4f4cb160efcd7b3cef1452e1b870b57f6a4cb8c4
TokenBase/TokenEvent.sol	f8c67f20e43d2f2f4dc0f9e1ea00b4815286ce78
TokenBase/TokenStorage.sol	b486e462a9fd93945e73f38226573bbae4238138

Appendix 2 - Disclosure