Formal Verification of dForce Lending protocol

Summary

This document describes the specification and verification of **dForce Lending protocol** using Certora Prover. The work was undertaken from **January 26th - February 8th 2021**. The latest commit that was reviewed and run through the Certora Prover was `1def2464ed2d00111744cd987d7fc854d06136b8`.

The scope of our verification was the iToken and Controller contracts with a focus on aspects of solvency and integrity.

The Certora Prover proved the implementation of the iToken and Controller are correct with respect to the formal rules written by the Certora team. During the verification process, the Certora Prover discovered minor bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high-level specifications of iToken and Controller. All the rules are publically available in a public GitHub: https://github.com/dforce-network/LendingContracts/tree/certora/spec

List of Main Issues Discovered

Severity: High

Issue:	Flashloan can cause users to be unfairly liquidated.
Rules Broken:	Flashloan's execution should not impact the externally observed exchange rate.
Description:	When a user takes a flash loan of the entire supply, the current cash of the system drops to zero while the borrows do not increase. This scenario can cause the exchange rate to drop to zero, rendering some users insolvent. An attacker can liquidate these insolvent users to seize their other collaterals and profit from the insolvency penalty.
Mitigation:	Account flash-loaned cash as borrowed to make the exchange rate stable within the flashloan.

Severity: Medium

Issue:	Owner of an iToken can be the iToken itself.
Rules Broken:	Exchange rate stability
Description:	If the iToken contract is its own owner, then withdrawReserves() is not really withdrawing the reserves. Reserves are reduced without reducing cash too.
Mitigation:	Disallow self-ownership of iTokens.

Severity: Medium

Issue:	Owner might add an unrelated iToken to the controller.
Rules Broken:	The list of iTokens may only contain iTokens that have the current controller set as their controller.
Description:	_addMarket() does not check that the added iToken points to the current controller, so the controller's owner could mistakenly add a foreign iToken. Such a mistake may allow users to synthetically increase their equity but the system is not able to seize those foreign iTokens. Such iTokens cannot be currently removed from the controller.
Mitigation:	Before adding an iToken to a controller A, check that the iToken's controller is also A.

Severity: Medium

Issue:	Insolvency.
Rules Broken:	Increase in an account's borrow is not greater than increase of total borrows by more than a "small number" resulting from rounding errors.
Description:	If at some point the totalBorrows becomes 0 but there are leftover account borrows, each accrual of interest will update the indices (increasing users' debt) while not increasing the total borrows (since interest accumulated is a fraction of the total borrows).
Mitigation:	It is possible to track in code the actual sum of borrows and sweep up the difference to the total borrows when it's becoming large enough.

Severity: Medium

Issue:	Total borrows may not reflect the actual sum of borrowed amounts.
Rules Broken:	Exchange rate stability
Description:	When a loan is repaid, the actual repay amount is adjusted to be the minimum between the repay amount and the account's borrows. The code allows reducing the account borrows by this actual amount and checks for underflow, while total borrows update is allowing an underflow and updates to 0 in that case.
Mitigation:	Simulations show that indeed the total borrows may deviate from the sum of borrows. Specification rules were used to show a very slow increase in the gap between total borrows variable and actual sum of borrows. The rate by which the gap can increase rises slowly as the gap increases (about 1e-17).

Severity: Low

Issue:	Mint allows altruism.
Rules Broken:	Exchange rate stability
Description:	A user may specify too small an underlying amount such that the amount of minted iTokens is 0. This could affect usability as users may be surprised that the mint did not result in them earning iTokens. This can also lead to increases in exchange rate (if such mints are aggregated).
Mitigation:	It is acceptable to donate cash to the protocol.

Severity: Low

Issue:	iTokens cannot be removed from the controller.
Rules Broken:	iTokens can be removed
Description:	Once an iToken was added to the controller, it cannot be removed from the iTokens list.
Mitigation:	Allow iTokens to be removed from the controller, as a privileged owner action and only if the iToken removed does not point to this controller.

Severity: Low

Issue:	Flashloan changes exchange rate.
Rules Broken:	Exchange rate stability
Description:	After a flashloan the protocol should have cash increased by at least protocol fee. If the cash in the system increased by exactly the protocol fee, the exchange rate is not affected. If the cash in the system was increased by more than the fee, the exchange rate's numerator can increase by more than the quantity of $flashloanAmount * flashloanFeeRatio * (1 - protocolFeeRatio)$.
Mitigation:	No mitigation needed - It is acceptable to donate cash to the protocol

Severity: Low

Issue:	Transfer reverting if RewardDistributor is not set.
Rules Broken:	Sufficient conditions for the success of pre-transfer hook
Description:	Some properties of the controller are defined in the initialization phase and after the deployment. Those must be set before the market activity begins.
Mitigation:	Mitigated by a well organized deployment procedure.

Severity: Low

Issue:	Insolvency.
Rules Broken:	Exchange rate is always positive
Description:	If the total reserves are big enough, after liquidating or redeeming, the sum of cash and borrows can decrease to be exactly worth the total reserves. It means the nominator in the calculation of the exchange rate is now zero. The system is now insolvent, and no new tokens can be minted or redeemed. Any debt left in the insolvent system will never be paid. This scenario happens when a mass of users redeem a particular token at once (the market is falling).
Mitigation:	Add a condition that if the exchange rate is calculated to be zero, return some default positive minimal value instead.

Disclaimer

Notations

1. ✔ indicates the rule is formally verified on the latest commit. We write ✔* when the rule was verified on a simplified version of the code (or under some assumptions).
2. ❌ indicates the rule is violated in the version under test.
3. ✍ indicates the rule is not yet formally specified.
4. 🔁 indicates the rule is postponed **(<due to other issue, low priority?>)** .
5. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
   The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim$op C_2, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of iToken

iTokens are synthetic tokens derived from some underlying asset (e.g. DAI). The iTokens are used to interact with the dForce lending protocol. Each iToken is connected to a particular *controller* that holds a set of supported iTokens and maintains users' participation in collateralization and borrowing. The iToken holds the individual indices, total borrows, and reserves as well as users' individual balances.

Functions

controller(): address
Returns the controller associated with the iToken.

underlying(): address
Returns the underlying token associated with the iToken.

balanceOf(address account): uint
Returns the number of iTokens belonging to *account*.

allowance(address account, address allowed): uint
Returns the number of iTokens that *account* allows *allowed* to transfer on *account*'s behalf.

totalSupply(): uint
Returns the number of iTokens in circulation.

cash(): uint
Returns the number of underlying tokens that are owned by the iToken.

totalBorrows(): uint
Returns the number of underlying tokens that are borrowed.

totalReserves(): uint
Returns the number of underlying tokens that are in the reserve.

exchangeRateStored(): uint
Returns the current exchange rate. When the supply of iTokens is 0, the exchange rate is 1. Otherwise, it is computed by the formula:

```
(cash() + totalBorrows() - totalReserves()) / totalSupply()
```

reserveRatio(): uint
Returns the current reserve ratio, the portion of interest that goes into the reserves.

maxReserveRatio(): uint
Returns the maximal allowed reserve ratio, set to 1 (or 1e18, scaled).

borrowsSize(address account): uint
Returns the number of iTokens that *account* is borrowing.

borrowBalanceStored(address account): uint, uint
Returns the principal borrow and interest index of the user *account* in this iToken.

mint(address caller, uint amount): bool
caller sends out underlying and in return gets an amount newly minted iTokens determined by the exchange rate.

redeem(address caller, uint amount): bool
caller sends out iTokens to be burned and in return gets a matching underlying amount determined by the exchange rate.

borrow(address caller, uint amount): bool
caller borrows an amount *amount* of iTokens. This increases the debt of the user towards the system.

repayBorrow(address caller, uint amount): uint256
caller repays an amount *amount* of iTokens to cover a part their debt. The actual repay amount is returned. (The actual repay amount may be smaller than the given amount.)

seize(address caller, address liquidator, address account, uint amount): bool
caller invokes the seize operation on behalf of a *liquidator* on the iTokens of *account*, and reduces it by *amount* transferring it to *liquidator*. Returns true if the operation was successful.

liquidateBorrow(address liquidator, address account, uint amount, iToken collateralTo): bool
Liquidates *account*'s borrow in the current iToken if it is in shortfall state. The liquidator repays an *amount* of the borrow and receives the account's *collateralToken* iTokens, in an amount equivalent to the repay amount plus a liquidation incentive.

updateInterest()
A periodic per-block operation that accrues interest, updating total borrows, total reserves, and the borrow index.

flashloan(address recipient, uint loanAmount, bytes data)
An operation that sends a *loanAmount* of cash to the *recipient* and executes a special interface the *recipient* implements. After the *recipient*'s execution ends, the cash of the system must be restored with an extra fee, a part of which is transferred to the reserve.

validState(): bool
True if all the following three conditions hold:

1. The reserve ratio is not greater than the maximal reserve ratio.
2. The borrow rate is not greater than the maximal borrow rate.
3. If the total supply is greater than zero, then `cash() + totalBorrows() - totalReserves() > 0`

Properties

Exchange rate

1. **Exchange rate stability ❌**
   The exchange rate is a function of the cash, borrows, reserves and iToken supply. In order to approximate the correct movement of the exchange rate, this rule individually compares the difference between the numerator part and the denominator part, assuming no interest accrual in `op()`:

```
{ numerator = cash() + totalBorrows() - totalReserves() ∧
  denominator = totalSupply() ∧
  accrualBlockNumber() == currentBlockNumber }
      op() at currentBlockNumber;
{ deltaNum = numerator - (cash() + totalBorrows() - totalReserves()) ∧
  deltaDenom = denominator - totalSupply() ∧
  deltaNum < 0 ⇔ deltaDenom < 0 ∧
  deltaNum > 0 ⇔ deltaDenom > 0
}
```

   The following table describes correct movements ('+' for increase and '-' for decrease, '0' for no change) in each quantity:

	cash	borrows	reserves	supply
mint ❌	+	0	0	+
redeem ✔	-	0	0	-
borrow ✔	-	+	0	0
repayBorrow ❌ (violated because total borrows could be smaller than account borrows)	+	-	0	0
updateInterest (see "Interest accrual impact" rule) ✔	0	+	+	0
withdrawReserves ✔ (owner must not be the iToken)	-	0	-	0
flashloan ❌	+	0	+	0

2. **Flashloan's execution should not impact the externally observed exchange rate** ❌
   When a flashloan is executed, the stored exchange rate observed by the executor and other contracts should be equal to the exchange rate right before calling the executor.

3. **When the total supply is positive, cash + totalBorrows > totalReserves** ❌

4. **Exchange rate cannot go down to zero.** This is a result of #8. ❌

```
{ exchangeRateStored() > 0 } op(); { exchangeRateStored() > 0 }
```

Borrow and Total Borrows

5. **Increase in an account's borrow is not greater than increase of total borrows by more than a "small number"** ❌
   The gap between the totalBorrows quantity and the sum over all users' borrows should be similar to one another. In particular, this rule checks that the increase in sum over all users' borrows is not larger than totalBorrows by more than some small ε (e.g., 1000e-18). Since each function modifies at most one user's borrows, we use an arbitrary user's *user* borrow balance.

```
{ userBorrows = borrowBalanceStored(user) ∧
  total = totalBorrows() ∧
  total > 0 // extra requirement
}
      op()
{ borrowBalanceStored(user) ≥ userBorrows ⇒
      (borrowBalanceStored(user) - userBorrows ≤
            totalBorrows() - total + ε)}
```

6. **Only a single account's borrow snapshot can be updated in any single operation ✔**

```
{ (principal1, index1) = accountBorrows(user1) ∧
  (principal2, index2) = accountBorrows(user2) ∧
  user1 ≠ user2 }
      op();
{ (principal1, index1) = accountBorrows(user1) ∨
  (principal2, index2) = accountBorrows(user2) }
```

7. **Total borrows is equal to the sum of individual account borrows under the no-interest assumption (**up to a small number due to rounding errors) ❌

8. **If a user repaid a borrow, then the system must have gained some value (could be zero though) ✔**

```
{ validState() ∧ c = cash() }
      repayBorrow(borrower, amount)
{ cash() ≥ c }
```

9. **When a user with total borrows of x repays a borrow for amount y, then the systems gains at least min (x, y) in cash ✔**

```
{ validState() ∧
  c = cash() ∧
  b = borrowBalanceStored(borrower) }
      repayBorrow(borrower, amount)
{ cash() - c ≥ min(b, amount) }
```

10. **If a user borrows X then immediately repays a borrow of X at the same block, the balances and supply of the system are unaffected ✔**

```
{ recipient_iTokens = balanceOf(borrower) ∧
  supply = totalSupply() }
      borrow_success = borrow(borrower, amount);
      repayBorrow(borrower, amount)
{ borrow_success ⇒
      recipient_iTokens = balanceOf(borrower) ∧
      supply = totalSupply() }
```

Mint and Redeem

11. **mint() has no effect with amount 0 - the iToken supply, iToken balance of minter, underlying balance of minter and underlying balance of iToken are unaffected ✔**

```
{ minterBalance = balanceOf(minter) ∧
  minterUnderlyingBalance = underlying().balanceOf(minter) ∧
  c = cash() ∧
  supply = totalSupply() }
     mint(minter, 0);
{ minterBalance = balanceOf(minter) ∧
  minterUnderlyingBalance = underlying().balanceOf(minter) ∧
  c = cash() ∧
  supply = totalSupply() }
```

12. **redeem functions have no effect with amount 0 - the iToken supply, iToken balance of minter, underlying balance of minter and underlying balance of iToken are unaffected ✔**

```
{ redeemerBalance = balanceOf(redeemer) ∧
  redeemerUnderlyingBalance = underlying().balanceOf(redeemer) ∧
  c = cash() ∧
  supply = totalSupply() }
     redeem(redeemer, 0);
{ redeemerBalance = balanceOf(redeemer) ∧
  redeemerUnderlyingBalance = underlying().balanceOf(redeemer) ∧
  c = cash() ∧
  supply = totalSupply() }
```

13. **mint is an additive operation ❌**

```
{ mint(minter, x);
  mint(minter, y) } ~balanceOf(minter) { mint(minter, x+y) }
```

Interest

14. **Interest accrual impact ✔**
    When the interest is updated, cash and supply stay constant, while borrows and reserves may increase.

```
{ cash = cash() ∧
  supply = totalSupply() ∧
  borrows = totalBorrows() ∧
  reserves = totalReserves() }
      updateInterest()
{ cash = cash() ∧
  supply = totalSupply() ∧
  borrows ≤ totalBorrows() ∧
  reserves ≤ totalReserves() }
```

15. **The interest rate does not update if we are in the same block number as the accrual block.** Specifically we care that the total borrows and total reserves are not affected. This is critical for reentrancy cases in flash loan. ✔

```
{ borrows = totalBorrows() ∧ reserve = totalReserves() }
      op() at block accrualBlockNumber()
{ borrows = totalBorrows() ∧ reserve = totalReserves() }
```

Miscellaneous

16. **Only the iToken contract can call seize ✔**
    Seize can only be called by an iToken that has the same controller as the current iToken.

```
{ callerIsItoken = controller().isInItokens(caller) ∧
  balance = balanceOf(account) }
  success = seize(caller, liquidator, account, amount);
{ success ⇒ callerIsItoken ∧
  balanceOf(account) ≠ balance ⇒ callerIsItoken }
```

17. iTokens cannot be removed from a non-borrowing account ✔

If an account *a* has no borrows, then *a*'s iToken balances can only be decreased if:

(1) *a* is seized (called by another iToken); or
(2) *a* has approved another account; or
(3) *a* is executing the operation as the caller

```
{ borrowsSize(a) = 0 ∧
  preBalance = balanceOf(a) ∧
  preAllowance = allowance(a, sender) ∧
  callerIsItoken = controller().isInItokens(caller) }
      op() by caller;
{ balanceOf(a) ≥ preBalance ∨
  (op = seize ∧ callerIsItoken) ∨
  preAllowance ≥ preBalance - balanceOf(a) ∨
  caller = a }
```

18. Liquidation of a user's funds is impossible if they're not borrowing ✔

```
{ preBalance = balanceOf(a) ∧
  borrowsSize(a) = 0 }
   success = liquidateBorrow(liquidator, a, amount, collateralToken);
{ !success ∧ balanceOf(a) = preBalance }
```

19. The reserve ratio is never greater than the maximal reserve ratio ✔

```
{ reserveRatio() < maxReserveRatio() }
      op();
{ reserveRatio() < maxReserveRatio() }
```

Verification of Controller

Functions

isInITokens(address a) : bool
Returns true if *a* is in the set of supported iTokens.

iTokensLength() : uint256
Returns the length of the registered iTokens list.

borrowFactor(address iToken) : uint256 [mantissa]
Returns the borrow factor of the given *iToken*. Returns 0 if *iToken* is not registered.

Properties

20. **Sufficient conditions for success of pre transfer hook ✔**
    The before transfer hook will permit a transfer if:
    (1) Transfers are not paused
    (2) The market is listed (i.e. has a positive borrow factor)
    (3) The sending account is not participating in the market OR the sending account has not marked the market as collateral
    (4) The sending account has no borrows in the market
    (5) The RewardsDistributor was initialized
    (6) no ETH are transferred to the call

21. **An iToken is registered in the iTokens set if and only if it has a positive borrow factor ✔**

```
{ borrowFactor(iToken) > 0 ⇔ isInItokens(iToken) }
      op();
{ borrowFactor(iToken) > 0 ⇔ isInItokens(iToken) }
```

(If the number of iTokens is MAX_UINT, then addMarket() may overflow the length of the array holding the iTokens and then the iTokens list may seemingly be empty. This causes a violation of the invariant, but since it's caused by an external library edge case, this technicality is omitted from the above definition.)

22. **The list of iTokens can be changed by at most a single element in every single operation ✔**

The MAX_UINT to 0 overflow is omitted on purpose - this rule shows it is not a realistic scenario.

```
{ len = iTokensLength() ∧
  len < MAX_UINT }
      op();
{ | iTokensLength() - len | ≤ 1 }
```

23. **iTokens are persistent ✔**

Once an iToken is added, it stays in the list. This rule's proof can be interpreted negatively as it is impossible to remove an iToken, one can only pause some of its functionalities.

```
{ isInItokens(iToken) } op(); { isInItokens(iToken) }
```

24. **The list of iTokens may only contain iTokens that have the current controller set as their controller ❌**

```
{ isInItokens(iToken) => iToken.controller() = thisController }
      thisController.op();
{ isInItokens(iToken) => iToken.controller() = thisController }
```