DeltaPrime Smart Contract Review

Reviewer: Piotr Szlachciak (https://twitter.com/PiotrSzlachciak)

This review was performed over two days (Dec 04 - 05 2021) and consisted of reading and reasoning about the code. No automated vulnerability checkers or issue checklists were used. The reviewer was made aware that the code was also being audited by independent auditing firms.

Originally this report included issues for the redstone-finance/redstone-evm-connector code that is DeltaPrime's dependency. Those issues were moved to a separate report.

Number of issues found:

- **Medium** severity: 2
- **Low** severity: 12
- Style suggestions & optimizations: 21

MEDIUM SEVERITY ISSUES

`ERC20.transfer` CALLS SHOULD BE REPLACED WITH `TransferHelper`

Severity: medium

There are many ERC20 tokens that don't follow the standard and don't return a value, or follow it too closely and do not revert on failed transfers. TransferHelper handles all those cases gracefully.

ASSETS CAN ONLY BE ALLOWED OR NOT

Severity: medium

Adding an asset as a supported asset is sticky - removing it means triggering liquidations for many active loans. I propose to add a special invest blacklist that would prevent a token from being bought but still allow it to be sold. That way tokens can be gracefully phased out:

1. Mark token as sell-only and inform the community
2. People exit their positions
3. De-list the token
4. Liquidate remaining loans (only people that don't care anyway)

LOW SEVERITY ISSUES

INVALID INVARIANT FOR `calculateDepositRate`

Severity: low

This vulnerability was first reported as more serious but it turned out to be an error in the projects documentation, so the discrepancy below doesn't actually violate the stated goals of the project.

https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L52

The `calculateDepositRate` function is based on the following invariant:

```
value_of_loans * borrowing_rate = value_of_deposits * deposit_rate
```

However the project readme states that "The total amount of interest earned by depositors always equals the total amount of interest owned by borrowers.". This could be expressed mathematically as:

```
L = value_of_loans * ((1 + borrowing_rate) ^ seconds_passed - 1)
R = value_of_deposits * ((1 + deposit_rate) ^ seconds_passed - 1)
L = R
```

Those two invariants are in contradiction with each other. Let's demonstrate with the following example:

```
totalLoans = 1000
totalDeposits = 2000

// using VariableUtilisationRatesCalculator
poolUtilisation = 0.5
borrowingRate = 0.11
depositRate = 0.055

// check code invariant (OK)
totalLoans * borrowingRate = 110
totalDeposits * depositRate = 110

// apply interest over a year using CompoundingIndex
loanMultiplier = 1.1162780702447197
depositMultiplier = 1.0565406183101167

// check readme invariant (NOT OK)
loanInterest = 116.27807024471963
depositInterest = 113.08123662023354
```

As a potential fix I would have a single interest rate for borrowing, distribute earnings proportionally to deposits and derive a lending APY on the frontend. Not a trivial change.

For completeness here is the code to calculate the compounding interest in this example:

```
// compounding-index.test.ts
describe("Maintains balance", () => {
  let di: CompoundingIndex;
  let bi: CompoundingIndex;

  before("deploy the Compounding index", async () => {
    di = await init("0.055", owner);
    bi = await init("0.11", owner);
  });

  it("after a year", async () => {
    await time.increase(time.duration.years(1));
    let dv = fromWei(await di.getIndex());
    let bv = fromWei(await bi.getIndex());

    let borrowed = 1000;
    let deposited = 2000;

    let owed = borrowed * bv;
    let gained = deposited * dv;

    console.log(bv, dv);
    console.log(owed - borrowed, gained - deposited);
  });
});
```

UNCHECKED `Pool` INITIALIZER PARAMETER

Severity: low

https://github.com/ava-loan/deltaprime/blob/master/contracts/Pool.sol#L42-L46

The contracts `ratesCalculator_`, `depositIndex_` and `borrowIndex_` are all indirectly checked in `_updateRates`.

However, because `borrowersRegistry_` isn't called it should be checked for being either `address(0)` (which is permitted because of canBorrow) or being a contract (EXTCODESIZE).

UNCHECKED `setRatesCalculator` AND `setBorrowersRegistry` PARAMETERS

Severity: low

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L64 https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L77

The addresses passed can be EOAs. They should be checked for being contracts (EXTCODESIZE).

`approve` IS DANGEROUS AND SHOULD BE ACCOMPANIED WITH `increaseAllowance` AND `decreaseAllowance`

Severity: low

https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#IERC20-approve-address-uint256-

There are well known issues with approve and having those methods can mitigate the damage.

NON-SMART CONTRACT POOL USERS WILL UNLIKELY EVER REPAY THE LOAN IN FULL

Severity: low

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L200

Note that this might be a non-issue if the BorrowersRegistry only ever allows smart contracts to borrow.

Because the borrowed amount is dynamic and interest accrues every second it can be close to impossible for regular users to submit a transaction that repays the loan in full. For that they would need to know the timestamp of the block that will include their transaction.

As a mitigation I suggest allowing repay amounts greater than the borrowed value. Excess funds should be sent back to the sender.

`getAllLoans` WILL STOP WORKING AFTER ENOUGH LOANS HAVE BEEN CREATED

Severity: low

https://github.com/ava-loan/deltaprime/blob/main/contracts/SmartLoansFactory.sol#L95 https://github.com/ava-loan/deltaprime/blob/main/contracts/SmartLoansFactory.sol#L34

From the perspective of the UI it should be better to rely on events, especially since they can be filtered by the loan creator. Removing this method will also save 25000 gas on every loan creation, because there is no need to keep the array in storage.

`PangolinExchange` SHOULD EXTEND `ReentrancyGuard`

Severity: low

https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol

It currently extends ReentrancyGuardUpgradeable but it is not deployed behind a proxy.

`.transfer` CALLS FOR AVAX SHOULD BE REPLACED WITH `TransferHelper`

Severity: low

Solidity .transfer calls limit the gas being sent with the transaction which can cause problems.

`__ReentrancyGuard_init` IS NOT CALLED

Severity: low

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L35 https://github.com/ava-loan/deltaprime/blob/main/contracts/SmartLoan.sol#L35

Pool and SmartLoan both extend ReentrancyGuardUpgradeable but don't call it's init method. This results in the first call to nonReentrant functions being more expensive.

`SmartLoan`'S CODE IS TOO COMPLEX

Severity: low

I suggest putting public apis at the top of the file, not calling one public api from another and maintaining a list of internal operations at the bottom. This will help solve issues like multiple calls to `isSolvent()`.

This is assigned low severity instead of style because complex code hides bugs easier.

`nonReentrant` MODIFIER APPLIED INCONSISTENTLY

Severity: low

Why is `PangolingExchange.buyAsset` `nonReentrant` but `sellAsset` isn't?

INTEGRATION TEST FAILS OCCASIONALLY

Severity: low

Running integration tests causes two tests to fail. Unfortunately they now work again. I tried reproducing deterministically by passing `--fork-block-number 7788500` to `yarn forked-test-node` but this just causes a lot of errors, I presume because the rpc does not point to an archive node.

Commands used:

```
yarn forked-test-node
// in another terminal
yarn migrate:local
yarn test-integration
```

Failing tests:

```
  1) Smart loan
       A loan with debt and repayment
         should prevent borrowing too much:
     AssertionError: Expected transaction to be reverted with LoanInsolvent(), but other exception
was thrown: ProviderError: Error: Transaction reverted without a reason string


  2) Smart loan
       A loan with sellout and proxy upgradeability
         should fail a sellout attempt:
     AssertionError: Expected transaction to be reverted with LoanSolvent(), but other exception was
thrown: ProviderError: Error: VM Exception while processing transaction: reverted with an
unrecognized custom error
```

STYLE SUGGESTIONS & OPTIMIZATIONS

INDEX CONTRACTS DON'T NEED TO BE DEPLOYED FOR `Pool` IMPLEMENTATION CONTRACT

Severity: style

https://github.com/ava-loan/deltaprime/blob/master/contracts/Pool.sol#L31-L32

The initialize function deploys the indices for the proxy, however the implementation needlessly deploys the contracts during deployment.

I suggest just having the variables declared:

```
CompoundingIndex depositIndex;
CompoundingIndex borrowIndex;
```

`getIndexedValue` NEEDLESSLY READS STORAGE TWICE

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/CompoundingIndex.sol#L84

It first reads `userUpdateTime[user]` to later read it again in `getLastUserUpdateTime`.

I suggest the following refactor:

```
uint256 userTime = userUpdateTime[user];
uint256 prevUserIndex = userTime == 0 ? BASE_RATE : prevIndex[userTime];
```

`approve` IS POTENTIALLY TOO COMPLEX

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L109-L118

The approve function guards against setting the approval to a value that exceeds the balance. However the approval can still exceed user's balance if they first approve and then transfer. This check causes the approve function to perform multiple storage writes, balooning costs.

If this check is indeed important a potential way to reduce costs is to do a check against balanceOf instead of performing the update.

If the idea is to disallow infinite approvals maybe the check should be against max uint instead.

`calculateDepositRate` AND `calculateBorrowingRate` CAN HAVE PURE VISIBILITY

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L56 https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L80

Currently both functions are marked as `external view`. This is unfortunate, because `calculateDepositRate` uses `calculateBorrowingRate` and it has to use `this` to perform the external call. I suggest updating the visibilities as follows:

- `calculateDepositRate` - `external pure`
- `calculateBorrowingRate` - `public pure`, removing the need for `this` in `calculateDepositRate`

`Pool._updateRates` IS VERY INEFFICIENT

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L297-L300

Firstly, the function calculates `totalBorrowed()` and `totalSupply()` twice. Setting the rate on an index does not change the index so the calculations return the same values when performed the second time. This means that the first optimisation could be this:

```
uint256 borrowed = totalBorrowed();
uint256 deposited = totalSupply();
depositIndex.setRate(_ratesCalculator.calculateDepositRate(borrowed, deposited));
borrowIndex.setRate(_ratesCalculator.calculateBorrowingRate(borrowed, deposited));
```

Secondly we can see that the operations `calculateDepositRate` and `calculateBorrowingRate` are pure mathematical calculations. Moreover `calculateDepositRate` depends on `calculateBorrowingRate`. This means that the `borrowingRate` is computed twice. We can fix this by adding a new method `calculateRates` to `IRatesCalculator`. Then we refactor:

```
(uint256 depositRate, uint256 borrowingRate) =
    _ratesCalculator.getRates(totalBorrowed(), totalSupply());
depositIndex.setRate(depositRate);
borrowIndex.setRate(borrowingRate);
```

UNUSED IMPORTS

Severity: style

https://github.com/ava-loan/deltaprime/blob/master/contracts/CompoundingIndex.sol#L6

The CompoundingIndex contract imports console but does not use it.

UNNECESSARY RAY CONVERSION IN `getPoolUtilisation`

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L43-L45

The calculation should always yield the same result if written as `_totalLoans.wadDiv(_totalDeposits)` with the exception of the very last bit, giving a difference of at most 10 ** (-18). The conversion to ray would make some sense if there were more intermediate steps involved.

UNNECESSARY RAY CONVERSION IN `calculateDepositRate`

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L62-L65

Multiplying wads converted to rays does not increase precision. During division the additional precision is immediately lost because of the conversion back to wad with the exception of the very last bit, giving a difference of at most 10 ** (-18).

UNNECESSARY RAY CONVERSIONS IN `calculateBorrowingRate`

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L88-L90 https://github.com/ava-loan/deltaprime/blob/main/contracts/VariableUtilisationRatesCalculator.sol#L94-L96

Multiplying wads converted to rays does not increase precision.

`getAssetPriceInAVAXWei` PERFORMS A LOT OF THE SAME CALCULATIONS TWICE

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/SmartLoan.sol#L278

`getPriceFromMsg` decodes calldata, verifies signatures and iterates over all data points to find the specific asset. This is done twice, even though with slight modifications could only be done once.

`ether` KEYWORD SHOULD BE REPLACED WITH `e18`

Severity: style

https://github.com/ava-loan/deltaprime/blob/master/contracts/Pool.sol#L21 https://github.com/ava-loan/deltaprime/blob/master/contracts/Pool.sol#L334 and others

The value does not represent the amount of money but rather a fraction.

I suggest that instead of using `0.95 ether` as a constant you'd use `0.95e18`. Similarly `1 ether` could be written as `1e18`.

EXPLICIT CALCULATION OF SECONDS IN A YEAR

Severity: style

https://github.com/ava-loan/deltaprime/blob/master/contracts/CompoundingIndex.sol#L17 https://docs.soliditylang.org/en/latest/units-and-global-variables.html?highlight=days#time-units

I suggest writing it as:

```
uint256 private constant SECONDS_IN_YEAR = 365 days;
```

THE NAME `setAssets` SUGGESTS IT WILL OVERWRITE EXISTING ONES

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol#L110

Instead it adds or updates assets. Maybe this should be renamed to `updateAssets` and `updateAssets` should be renamed to `_updateAssets`.

MODIFIERS COMMENT DOESN'T LIST ANY MODIFIERS

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol#L32

The comment should be removed

USELESS FUNCTION `refundTokenBalance`

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol#L81 https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol#L35-L39

Line 81 can be replaced with line 38 and the function can be removed.

NO NEED TO CALL `exchange.getAssetAddress`

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/SmartLoan.sol#L88

Just use `address(token)`.

`pangolinRouter.WAVAX` CAN BE A HARDCODED CONSTANT

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol#L193 https://github.com/ava-loan/deltaprime/blob/main/contracts/PangolinExchange.sol#L205

MARKING INTERNAL STATE VARIABLES AS PRIVATE

Severity: style

https://github.com/ava-loan/deltaprime/blob/master/contracts/Pool.sol#L31-L32

While the default, internal visibility does not expose `depositIndex` and `borrowIndex` all other state variables are marked as `private`. For consistency I recommend marking them as private too.

MISSING `.gitignore`

Severity: style

Note that this likely only affects the repository used for the review.

Causes folders like `cache` or `node_modules` to clutter the git changelog.

I suggest adding a `.gitignore` file.

BUILD ARTIFACTS

Severity: style

Note that this likely only affects the repository used for the review.

When building the repository with `yarn build` some artifacts committed to the repository are modified. This includes typechain bindings.

I suggest adding `public` and `typechain` folders to .gitignore, however it seems that the `typechain` folder cannot be fully regenerated, because running `yarn build` will not produce typechain bindings for migration contracts.

USE PRETTIER SOLIDITY

Severity: style

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L184

I noticed that the contracts have inconsistent code formatting. I suggest using: https://github.com/prettier-solidity/prettier-plugin-solidity

RETRACTED ISSUES

`borrow` CAN EXCEED `MAX_POOL_UTILISATION_FOR_BORROWING`

Severity: medium

This vulnerability doesn't actually exist in the code and was erroneously reported, because of a mistake by the reviewer. The reported text is left below for completeness.

https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L334 https://github.com/ava-loan/deltaprime/blob/main/contracts/Pool.sol#L183

The code first checks if `totalBorrowed / totalSupply < MAX_POOL_UTILISATION_FOR_BORROWING`. If this is the case the total borrowed amount can be increased by any value.

I suggest extending the canBorrow modifier with a `value` parameter and checking `(totalBorrowed + value) / totalSupply < MAX_POOL_UTILISATION_FOR_BORROWING`.