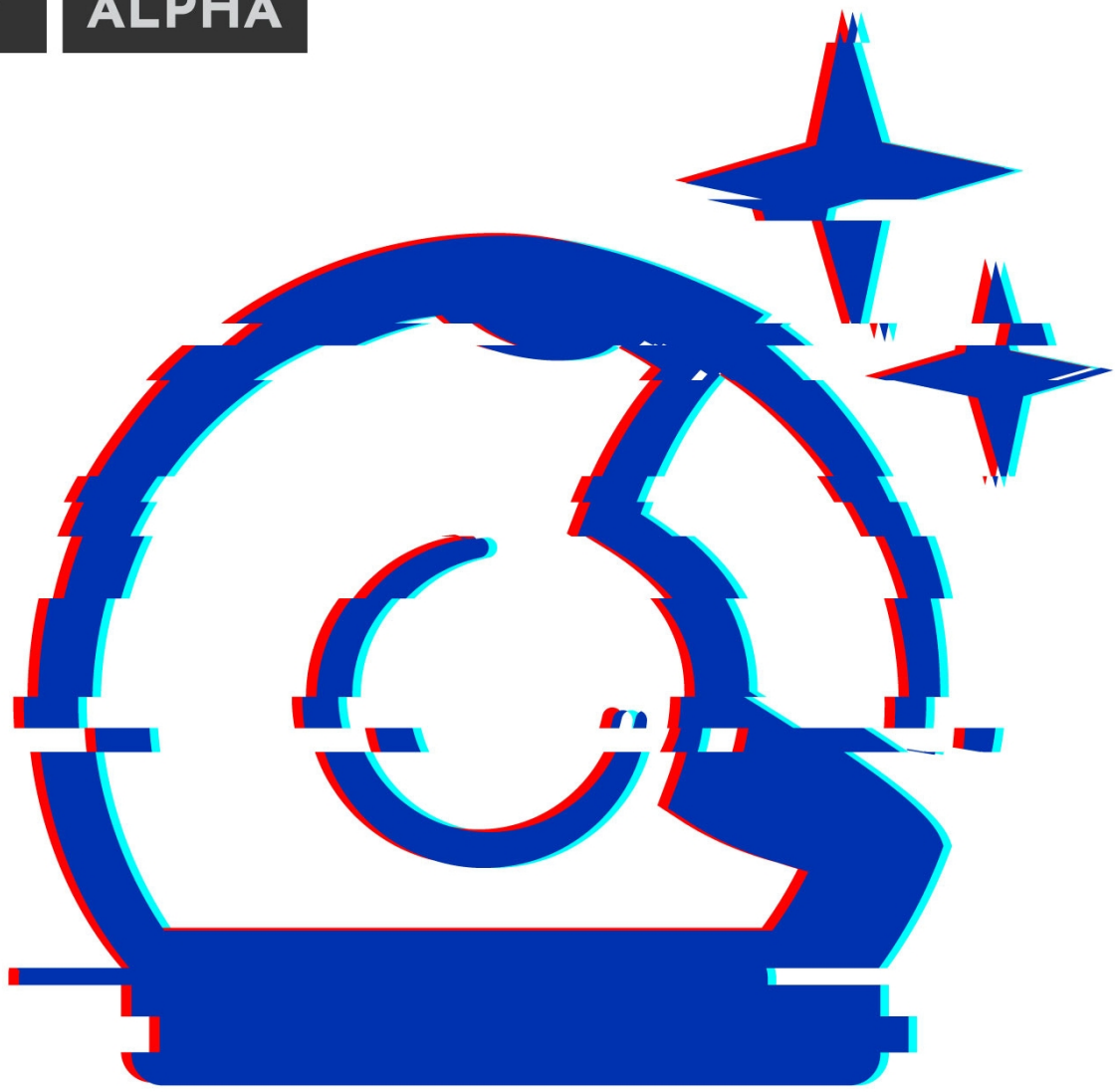
MONO CEROS ALPHA



**PROJECT**

# Conjure Finance

**CLIENT**

Conjure Finance

**DATE**

March 2021

**REVIEWERS**

Daniel Luca
@cleanunicorn

Andrei Simion
@andreiashu

# Table of Contents



# Details

- **Client** Conjure Finance
- **Date** March 2021
- **Lead reviewer** Daniel Luca (@cleanunicorn)
- **Reviewers** Daniel Luca (@cleanunicorn), Andrei Simion (@andreiashu)
- **Repository**: Conjure Finance
- **Commit hash** `230c58b64cee20a7a75361bcc543708cb7076267`
- **Technologies**
  - Solidity
  - Node.JS

# Issues Summary

| SEVERITY | OPEN | CLOSED |
|---|---|---|
| Informational | 5 | 0 |
| Minor | 5 | 0 |
| Medium | 7 | 0 |
| Major | 4 | 0 |

# Executive summary

This report represents the results of the engagement with **Conjure Finance** to review **Conjure Finance**.

The review was conducted over the course of **1 week** from **March 22 to March 26, 2021**. A total of **7.5 person-days** were spent reviewing the code.

## Week 1

During the first week, we started manually reviewing the contracts.

We noticed there is a quick sort implementation in the contract and it seems to be taken from a public gist.

code/contracts/ConjureFactory.sol#L323-L348

```
    /**
    * @dev implementation of a quicksort algorithm
    *
    * @param arr the array to be sorted
    * @param left the left outer bound element to start the sort
    * @param right the right outer bound element to stop the sort
    */
```

```
    function quickSort(uint[] memory arr, int left, int right) internal pure {
        int i = left;
        int j = right;
        if (i == j) return;
        uint pivot = arr[uint(left + (right - left) / 2)];
        while (i <= j) {
            while (arr[uint(i)] < pivot) i++;
            while (pivot < arr[uint(j)]) j--;
            if (i <= j) {
                (arr[uint(i)], arr[uint(j)]) = (arr[uint(j)], arr[uint(i)]);
                i++;
                j--;
            }
        }
        if (left < j)
            quickSort(arr, left, j);
        if (i < right)
            quickSort(arr, i, right);
    }
```

We proceeded to formally verify the implementation by using a simple contract:

```
contract QuickSort {
    /**
    * @dev implementation of a quicksort algorithm
    *
    * @param arr the array to be sorted
    * @param left the left outer bound element to start the sort
    * @param right the right outer bound element to stop the sort
    */
    function quickSort(uint[] memory arr, int left, int right) internal pure {
        int i = left;
        int j = right;
        if (i == j) return;
        uint pivot = arr[uint(left + (right - left) / 2)];
        while (i <= j) {
            while (arr[uint(i)] < pivot) i++;
            while (pivot < arr[uint(j)]) j--;
            if (i <= j) {
                (arr[uint(i)], arr[uint(j)]) = (arr[uint(j)], arr[uint(i)]);
                i++;
                j--;
            }
        }
        if (left < j)
            quickSort(arr, left, j);
        if (i < right)
            quickSort(arr, i, right);
    }

    function noSort(uint[] memory arr, int left, int right) internal pure {
        return;
```

```
    }

    function self(uint[] memory arr) public {
        // Sanity checks
        if (arr.length == 0) return;
        if (arr.length > 10) return;

        // Run quicksort
        quickSort(arr, 0, int(arr.length - 1));
        // noSort(arr, 0, int(arr.length - 1));

        // Check valid
        for (uint i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i+1]) {
                assert(false);
            }
        }
    }
}
```

And running Mythril on it:

```
$ myth version
Mythril version v0.22.17

$ myth a -m Exceptions ./formal/Quicksort.sol

The analysis was completed successfully. No issues were detected.
```

The function `noSort` was used to make sure an incorrect sorting algorithm is picked up by Mythril.

The code was included in the repository.

Continuing to check the code, we realized some of the basic functionality is not correct. Checking the tests, we realized there is an insufficient amount of tests and the tests are not strict enough. Lots of the tests check the methods can be called, but the result is never checked.

The lack of tests hints to an incorrect, inconsistent or incomplete functionality.

At the end of the week we delivered the report.

# Scope

The initial review focused on the Conjure Finance identified by the commit hash `230c58b64cee20a7a75361bcc543708cb7076267`.

We focused on manually reviewing the codebase, searching for security issues such as, but not limited to re-entrancy problems, transaction ordering, block timestamp dependency, exception handling, call stack depth limitation, integer overflow/underflow, self-destructible contracts, unsecured balance, use of origin, gas costly patterns, architectural problems, code readability.

We were not able to completely cover the scope because an unexpectedly high amount of issues was found, and we used lots of our time to explain the problems and draft the issues.

**Includes:**

- ConjureFactory.sol
- EtherCollateralFactory.sol

# Recommendations

We identified a few possible general improvements that are not security issues during the review, which will bring value to the developers and the community reviewing and using the product.

## Increase the number of tests

A good rule of thumb is to have 100% test coverage. This does not guarantee the lack of security problems, but it means that the desired functionality behaves as intended. The negative tests also bring a lot of value because not allowing some actions to happen is also part of the desired behavior.

Make tests not depend on Alchemy API or Etherscan. Tests should be able to run locally without the aid of an API. This also allows you to include a Continuous Integration step in the development process.

Tests are unreliable because they do not adequately test the functionality, but call the methods and expect not to fail.

## Set up Continuous Integration

Use one of the platforms that offer Continuous Integration services and implement a list of actions that compile, test, run coverage and create alerts when the pipeline fails.

Because the repository is hosted on GitHub, the most painless way to set up the Continuous Integration is through GitHub Actions.

Setting up the workflow can start based on this example template.

```yaml
name: Continuous Integration

on:
  push:
    branches: [master]
  pull_request:
    branches: [master]

jobs:
  build:
    name: Build and test
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [12.x]
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - run: npm ci
    - run: cp ./config.sample.js ./config.js
    - run: npm test

  coverage:
    name: Coverage
    needs: build
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [12.x]
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - run: npm ci
    - run: cp ./config.sample.js ./config.js
    - run: npm run coverage
    - uses: actions/upload-artifact@v2
      with:
        name: Coverage ${{ matrix.node-version }}
        path: |
          coverage/
```

This CI template activates on pushes and pull requests on the **master** branch.

```yaml
on:
  push:
```

```
    branches: [master]
  pull_request:
    branches: [master]
```

It uses an Ubuntu Docker image as a base for setting up the project.

```
    runs-on: ubuntu-latest
```

Multiple Node.js versions can be used to check integration. However, because this is not primarily a Node.js project, multiple versions don't provide added value.

```
    strategy:
      matrix:
        node-version: [12.x]
```

A script item should be added in the `scripts` section of package.json that runs all tests.

```
{
   "script": {
      "test": "buidler test"
   }
}
```

This can then be called by running `npm test` after setting up the dependencies with `npm ci`.

If any hidden variables need to be defined, you can set them up in a local version of `./config.sample.js` (locally named `./config.js`). If you decide to do that, you should also add `./config.js` in `.gitignore` to make sure no hidden variables are pushed to the public repository. The sample config file `./config.sample.js` should be sufficient to pass the test suite.

```
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - run: npm ci
    - run: cp ./config.sample.js ./config.js
    - run: npm test
```

You can also choose to run coverage and upload the generated artifacts.

```
    - run: npm run coverage
    - uses: actions/upload-artifact@v2
      with:
```

```
        name: Coverage ${{ matrix.node-version }}
        path: |
          coverage/
```

At the moment, checking the artifacts is not that easy, because one needs to download the zip archive, unpack it and check it. However, you can check the coverage in the **Actions** section once it's set up.

# Issues

## Sometimes funds might be unaccounted in `openLoan` function

Status Open Severity Major

### Description

The `openLoan` function splits the minting fees between the treasury and the deployer of the synth.

```
        // Fee distribution. Mint the fees into the FeePool and record fees paid
        if (mintingFee > 0) {
            // calculate back factory owner fee is 0.25 on top of creator fee
            arbasset.transfer(mintingFee / 4 * 3);

            address payable factoryowner = IConjureFactory(factoryaddress).getFactoryOwner();
            factoryowner.transfer(mintingFee / 4);
        }
```

Because Solidity does not support decimals there will be cases where the fee cannot be split exactly. Consider the following code as an example:

```
contract DivTest {

    function exec(uint256 mintingFee) public pure returns (uint256 divThenMult, uint256 multThen
        divThenMult = mintingFee / 4 * 3;
        multThenDiv = mintingFee * 3 / 4;
        quarter = mintingFee / 4;
        lost = mintingFee - (divThenMult + quarter);

        return (divThenMult, multThenDiv, quarter, lost);
    }
}
```

Calling the `exec` function with `1003` as the param results in `3` wei unaccounted for:


Having more ether in the contract than expected might create problems when trying to account for collateral ether from outside the contract, either in a UI or a different contract reading the balance.

code/contracts/EtherCollateralFactory.sol#L179

```
    function getContractInfo()
```

code/contracts/EtherCollateralFactory.sol#L200

```
        _ethBalance = address(this).balance;
```

### Recommendation

- Change the order of the operations: first multiply and then divide as done for the `multThenDiv` variable in the example above.
- Check if the sum of the splits is different than the `mintingFee` and account for any difference by adding it to `factoryowner` or `arbasset` split.

---

## `Conjure.getPrice` does not have enough observations at `init` time to compute an asset price

Status Open Severity Major

### Description

Synths that depend on the Uniswap V2 oracle will use the Time-Weighted Average Pricing. Below is a quote from Uniswap's building an oracle documentation (emphasis ours):

> Once you understand the kind of price average you require, it is a matter of storing the cumulative price variable from the pair as often as necessary, and **computing the average price using two or more observations** of the cumulative price variables.

During the initialization in `Conjure.init()` the contract checks the price of the oracle by calling the `getPrice` function:

code/contracts/ConjureFactory.sol#L214

```
        _deploymentPrice = getPrice();
```

In turn, `getPrice` will call `getInternalPrice()`:

code/contracts/ConjureFactory.sol#L420-L421

```
    function getPrice() public returns (uint) {
        uint256 returnPrice = getInternalPrice();
```

During the call to `init()` the `getInternalPrice` function will not have enough observations from Uniswap's Oracle to compute the price of an asset and this will result in a broken synth deployment:

code/contracts/ConjureFactory.sol#L485-L490

```
                // grab latest price after update decode between 0 and 10 days
                FixedPoint.uq112x112 memory price = _uniswapv2oracle.computeAverageTokenPrice(
                    _oracleData[i].oracleaddress,
                    0,
                    3600 * 24 * 10
                );
```

### Recommendation

Ensure that there are enough pre-recorded price observations before trying to determine the price for `_deploymentPrice` variable in the `init()` function.

---

## computeAverageTokenPrice is not a Uniswap function

Status Open Severity Major

### Description

In its current state of the repository, the `ConjureFactory` contains broken code and will result in unusable deployments of `Conjure` instances.

`ConjureFactory.getInternalPrice()` uses `computeAverageTokenPrice()` defined in the `UniswapV2OracleInterface`.

code/contracts/ConjureFactory.sol#L485-L490

```
                // grab latest price after update decode between 0 and 10 days
                FixedPoint.uq112x112 memory price = _uniswapv2oracle.computeAverageTokenPrice(
                    _oracleData[i].oracleaddress,
                    0,
                    3600 * 24 * 10
                );
```

code/contracts/ConjureFactory.sol#L132

```
        _uniswapv2oracle = UniswapV2OracleInterface(uniswapv2oracle);
```

code/contracts/interfaces/UniswapV2OracleInterface.sol#L7-L10

```
interface UniswapV2OracleInterface {
    function computeAverageTokenPrice(
        address token, uint256 minTimeElapsed, uint256 maxTimeElapsed
    ) external view returns (FixedPoint.uq112x112 memory);
```

Although the name of the `UniswapV2OracleInterface` might lead one to believe that this interface conforms to the Uniswap V2 TWAP price feed's public specs, this is not the case. This will lead to unworkable deployed instances of `Conjure` contracts (note: the argument `uniswapv2oracle_` passed to the `ConjureMint` can also be misleading).

code/contracts/ConjureFactory.sol#L795-L801

```
    function ConjureMint(
        string memory name_,
        string memory symbol_,
        address payable owner_,
        address uniswapv2oracle_,
        address collateralfactory_
    )
```

**Recommendation**

If the `UniswapV2OracleInterface` is similar to the one used in Indexed Finance at IndexedUniswapV2Oracle.sol, include that contract in the repository and provide documentation within `ConjureFactory` to make it clear that `uniswapv2oracle` constructor argument should conform to that type.

Additionally, the `test_conjure_factory.js` tests should be updated since this is a bag that would have been caught in the tests:

code/test/test_conjure_factory.js#L40-L46

```
    await conjureFactory.ConjureMint(
        "UNIT",
        "TEST",
        owner.address,
        zeroaddress,
        zeroaddress
    )
```

---

# Improve test code coverage

Status Open Severity Major

## Description

While full test code coverage doesn't guarantee a bug-free codebase it does however increase the confidence in the code. Additionally, it allows developers and auditors to focus on more exploratory potential attack vectors.

Currently, there are several key areas of code that are not covered in tests.

| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Lines |
|---|---|---|---|---|---|
| contracts/ | 63.74 | 46.88 | 62.5 | 64.07 | |
| CNJ.sol | 0 | 0 | 0 | 0 | ... 452,453,456 |
| ConjureFactory.sol | 76.88 | 59.09 | 78.79 | 77.39 | ... 733,735,736 |
| EtherCollateralFactory.sol | 98.82 | 79.03 | 100 | 98.83 | 629,631 |
| Owned.sol | 45.45 | 33.33 | 60 | 50 | ... 20,21,22,23 |
| SafeDecimalMath.sol | 38.1 | 33.33 | 41.67 | 38.1 | ... 183,184,187 |
| contracts/MockContracts/ | 100 | 100 | 100 | 100 | |
| ETHUSDOracle_MOCK.sol | 100 | 100 | 100 | 100 | |
| PriceTestOracle_MOCK.sol | 100 | 100 | 100 | 100 | |
| contracts/interfaces/ | 100 | 100 | 100 | 100 | |
| IConjure.sol | 100 | 100 | 100 | 100 | |
| IConjureFactory.sol | 100 | 100 | 100 | 100 | |
| IEtherCollateral.sol | 100 | 100 | 100 | 100 | |
| IEtherCollateralFactory.sol | 100 | 100 | 100 | 100 | |
| UniswapV2OracleInterface.sol | 100 | 100 | 100 | 100 | |
| contracts/lib/ | 0 | 0 | 0 | 0 | |
| FixedPoint.sol | 0 | 0 | 0 | 0 | ... 70,75,80,81 |
| All files | 62.89 | 46.64 | 59.48 | 63.27 | |

```
450       function getInternalPrice() internal returns (uint) {
451  49×      require(_oracleData.length > 0, "No oracle feeds supplied");
452           // storing all in an array for further processing
453  48×      uint[] memory prices = new uint[](_oracleData.length);
454
455  48×      for (uint i = 0; i < _oracleData.length; i++) {
456
457               // chainlink oracle
458  50×          if (_oracleData[i].oracleType == 0) {
459  20×              AggregatorV3Interface pricefeed = AggregatorV3Interface(_oracleData[i].oracleaddres
460  20×              uint price = uint(getLatestPrice(pricefeed));
461  20×              prices[i] = price;
462
463                   // norming price
464  20×              E if (_maximumDecimals != _oracleData[i].decimals) {
465  20×                  prices[i] = prices[i] * 10 ** (_maximumDecimals - _oracleData[i].decimals);
466                   }
467
468                   // if we have a basket asset we use weights provided
469  20×              I if (_assetType == 1) {
470                       prices[i] = prices[i] * _oracleData[i].weight;
471                   }
472               }
473
474               // uniswap TWAP
475  50×          I if (_oracleData[i].oracleType == 1) {
476                   // check if update price needed
477                   if (_uniswapv2oracle.canUpdatePrice(_oracleData[i].oracleaddress) == true) {
478                       // update price
479                       _uniswapv2oracle.updatePrice(_oracleData[i].oracleaddress);
480                   }
481
482                   // since this oracle is using token / eth prices we have to norm it to usd prices
483                   uint currentethtusdprice = uint(getLatestETHUSDPrice());
484
485                   // grab latest price after update decode between 0 and 10 days
486                   FixedPoint.uq112x112 memory price = _uniswapv2oracle.computeAverageTokenPrice(
487                       _oracleData[i].oracleaddress,
488                       0,
489                       3600 * 24 * 10
490                   );
491
492                   prices[i] = price.mul(currentethtusdprice).decode144();
493
```

```
386         /**
387         * @dev implementation of a square rooting algorithm
388         *
389         * @param y the value to be square rooted
390         * @return z the square rooted value
391         */
392         function sqrt(uint256 y) internal view returns (uint256 z) {
393             if (y > 3) {
394                 z = y;
395                 uint256 x = (y + 1) / 2;
396                 while (x < z) {
397                     z = x;
398                     x = (y.mul(UNIT).div(x) + x) / 2;
399                 }
400             } else if (y != 0) {
401                 z = 1;
402             }
403             // else z = 0
404         }
```

Improve test quality and make sure the values change in the direction you want, and also by the expected value.

code/test/test_collateral_liquidation.js#L142-L146

```
    // do assertions
    expect(loan_info_before.collateralAmount).to.not.equal(loan_info_after.collateralAmount);
    expect(loan_info_before.loanAmount).to.not.equal(loan_info_after.loanAmount);
    expect(wallet_before).to.not.equal(wallet_after);
    expect(balance).to.not.equal(balance_after);
```

Most of the methods are not very well tested, and most tests ensure the methods can be called, but the effects are not properly checked.

Should check how much ether was collected.

code/test/test_conjure_basics.js#L79-L81

```
it("Should be able to call collect fees", async function () {
  await conjure.connect(addr1).collectFees();
});
```

Should check the revert message, otherwise, the method might revert because of a different reason.

code/test/test_conjure_basics.js#L83-L85

```
it("Should revert if non owner calls init", async function () {
  await expect(conjure.init()).to.be.reverted;
});
```

code/test/test_conjure_basics.js#L87-L101

```
it("Should not init with odd array values", async function () {
  await expect(conjure.connect(addr1).init(
      0,
      0,
      ["1", "120000000000000000000"],
      false,
```

```
        ["0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419"],
        [],
        ["signature1"],
        [0x00],
        [0],
        [100],
        [8]
    )).to.be.reverted;
  });
```

A lot of the public methods have no tests.

The burn method has only 1 test and it only checks for reverts.

code/test/test_conjure_basics.js#L185-L190

```
  it("Should not be able to call burn from non collateral contract", async function () {
    await expect(conjure.connect(addr1).burn(
        addr1.address,
        1
    )).to.be.reverted;
  });
```

The comment does not match the implementation, `getPrice` is called 5 times, not 4.

code/test/test_conjure_pricing.js#L221-L226

```
    // now query price 4 times to see the mock effect
    await conjure.getPrice();
    await conjure.getPrice();
    await conjure.getPrice();
    await conjure.getPrice();
    await conjure.getPrice();
```

The variables `addr1`, `addr2`, `addr3` and `addr4` are never used.

code/test/test_conjure_pricing.js#L13

```
  let owner, addr1, addr2, addr3, addr4;
```

**Recommendation**

Ensure that all code is tested and the critical pieces of the system are thoroughly covered by tests.

Improve test quality to make sure the system behaves as expected.

---

## Optimize method `openLoanIDsByAccount`

Status Open Severity Medium

### Description

The method `openLoanIDsByAccount` needs to obtain the open loans from a bigger list of open and closed loans.

It first needs to identify the open loans and save them to a list.

code/contracts/EtherCollateralFactory.sol#L291-L301

```
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[_account];

        uint256[] memory _openLoanIDs = new uint256[](synthLoans.length);
        uint256 _counter = 0;

        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].timeClosed == 0) {
                _openLoanIDs[_counter] = synthLoans[i].loanID;
                _counter++;
            }
        }
```

Then it needs to create a new array with the correct size. This is a limitation of Solidity because it does not support `.push` operations on lists in memory, it only works on state lists.

code/contracts/EtherCollateralFactory.sol#L302-L303

```
        // Create the fixed size array to return
        uint256[] memory _result = new uint256[](_counter);
```

After the list is created, it needs to add the items in there.

code/contracts/EtherCollateralFactory.sol#L305-L308

```
        // Copy loanIDs from dynamic array to fixed array
        for (uint256 j = 0; j < _counter; j++) {
            _result[j] = _openLoanIDs[j];
        }
```

And finally, it can return the correctly sized array.

code/contracts/EtherCollateralFactory.sol#L309-L310

```
        // Return an array with list of open Loan IDs
        return _result;
```

There is a more "hackish" way of optimizing the filtering without knowing the size of the list beforehand. This can be achieved by using assembly and changing the array size in

place. However, you need to be careful when using assembly because it will change memory data and you need to be aware of the side effects.

We created an example that illustrates this hack.

```
contract OptimizedFilter {
    function filter(uint[] memory numbers) public pure returns (uint[] memory) {
        // Do one pass of the array and obtain all needed ids
        uint[] memory filtered = new uint[](numbers.length);
        uint j;
        for (uint i = 0; i < numbers.length; i++) {
            if (numbers[i] > 10) {
                filtered[j++] = numbers[i];
            }
        }

        // Change the list size of the array in place
        assembly {
            mstore(filtered, j)
        }

        // Return the resized array
        return filtered;
    }
}
```

### Recommendation

Consider optimizing the filtering and by using the assembly if you are confident in the side effects the assembly block adds.

### References

- Allocating Memory Arrays

---

## Rewrite `_getLoanFromStorage` to improve gas costs

Status Open Severity Medium

### Description

The method is extremely inefficient.

code/contracts/EtherCollateralFactory.sol#L695-L709

```
    /**
     * @dev gets a loan struct from the storage
     *
     * @param account the account which opened the loan
     * @param loanID the ID of the loan to close
```

```
     * @return synthLoan the loan struct given the input parameters
    */
    function _getLoanFromStorage(address account, uint256 loanID) private view returns (SynthLoa
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == loanID) {
                synthLoan = synthLoans[i];
            }
        }
    }
```

There are multiple problems with it.

The method needs to return the loan data based on a provided `loanID`. In the current state of the storage layout, the loan cannot be obtained directly by using the `loanID`, and the method needs to iterate over all of the loans the account has.

All of the items are loaded up in the memory. This is extremely inefficient.

code/contracts/EtherCollateralFactory.sol#L703

```
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[account];
```

Setting `synthLoans` as `storage` instead of `memory` will not load all of the data in memory, but load the storage pointer to the list.

When the required `loanID` is found, `synthLoan` saves the whole structure.

code/contracts/EtherCollateralFactory.sol#L705-L707

```
            if (synthLoans[i].loanID == loanID) {
                synthLoan = synthLoans[i];
            }
```

After the method finds the required loan, it keeps going, effectively wasting gas because it should not find another loan with the same `loanID`. Breaking the loop will decrease gas costs.

**Recommendation**

Either fix all problems or change the storage layout.

**References**

Changing how the data is stored in the contract can reduce the complexity of getting a loan to O(1). You can inspire yourself from this implementation.

- OpenZeppelin's EnumerableSet

---

# Some functions are expected to be view functions, but they unexpectedly change state

Status Open Severity Medium

### Description

The method `getPrice` is expected to be a getter because of its name.

code/contracts/ConjureFactory.sol#L415-L420

```
    /**
     * @dev gets the latest price of the synth in USD by calculation and write the chackpoints f
     *
     * @return the current synths price
    */
    function getPrice() public returns (uint) {
```

Hence, one is expected to get the price and return it, but it also changes state.

code/contracts/ConjureFactory.sol#L434-L435

```
        _latestobservedprice = returnPrice;
        _latestobservedtime = block.timestamp;
```

These 2 contract properties already have generated getters by Solidity, because they are defined as public.

code/contracts/ConjureFactory.sol#L88-L92

```
    // the latest observed price
    uint256 public _latestobservedprice;

    // the latest observed price timestamp
    uint256 public _latestobservedtime;
```

However, the contract state `_latestobservedprice` also has an explicit getter defined in the contract which makes the automatic generated one obsolete.

code/contracts/ConjureFactory.sol#L406-L413

```
    /**
     * @dev gets the latest recorded price of the synth in USD
     *
     * @return the last recorded synths price
    */
    function getLatestPrice() public view returns (uint) {
        return _latestobservedprice;
    }
```

Surprisingly the contract doesn't have an explicit getter for `_latestobservedtime`.

Because the `getPrice` method changes state, it also forces other methods that call it to not be defined as `view`, and signal to the reader that they might change state, even though they don't need to.

This is specific to `loanAmountFromCollateral` and `collateralAmountForLoan`.

code/contracts/EtherCollateralFactory.sol#L213-L220

```
    /**
     * @dev Gets the amount of synths which can be issued given a certain loan amount
     *
     * @param collateralAmount the given ETH amount
     * @return the amount of synths which can be minted with the given collateral amount
    */
    function loanAmountFromCollateral(uint256 collateralAmount) public returns (uint256) {
        uint currentprice = IConjure(arbasset).getPrice();
```

code/contracts/EtherCollateralFactory.sol#L229-L236

```
    /**
     * @dev Gets the collateral amount needed (in ETH) to mint a given amount of synths
     *
     * @param loanAmount the given loan amount
     * @return the amount of collateral (in ETH) needed to open a loan for the synth amount
    */
    function collateralAmountForLoan(uint256 loanAmount) public returns (uint256) {
        uint currentprice = IConjure(arbasset).getPrice();
```

These methods should be defined as view because their purpose is to return a value for the user / UI, not update the price for the whole system.

A view function exists which returns the price and doesn't update the state, but that is not called in `loanAmountFromCollateral` and `collateralAmountForLoan`.

**Recommendation**

Split the method `getPrice` into 2 methods, one that updates the price and one that returns the saved price. Name them accordingly, use each one as needed and make sure to create methods with very limited responsibilities.

**References**

One should follow the Single-responsibility principle, which is very beneficial when applied to methods.

- Single-responsibility principle
- Single Responsibility Principle for Methods

---

# Use same owned pattern throughout the code base

Status Open Severity Medium

### Description

Conjure has an implemented Owner pattern which consists of: setting the owner, allowing only the owner to call some methods, allow the owner to move ownership to another address.

Also, the comment does not correctly reflect the implementation, the `_owner` isn't necessarily the deployer of the contract.

code/contracts/ConjureFactory.sol#L45-L46

```
    // the owner and creator of the contract
    address payable public _owner;
```

code/contracts/ConjureFactory.sol#L117-L126

```
    constructor (
        string memory name_,
        string memory symbol_,
        address payable owner_,
        address factoryaddress_,
        address uniswapv2oracle,
        address collateralfactory_
    )
    {
        _owner = owner_;
```

code/contracts/ConjureFactory.sol#L158-L173

```
    function init(
        uint256 mintingFee_,
        uint8 assetType_,
        // pack variables together because of otherwise stack too depp error
        uint256[2] memory divisorRatio_,
        bool inverse_,
        address[] memory oracleAddresses_,
        uint8[] memory oracleTypes_,
        string[] memory signatures_,
        bytes[] memory calldata_,
        uint256[] memory values_,
        uint8[] memory weights_,
        uint256[] memory decimals_
    ) public
    {
        require(msg.sender == _owner, "Only owner");
```

code/contracts/ConjureFactory.sol#L268-L277

```
    /**
     * @dev lets the owner change the contract owner
     *
     * @param _newOwner the new owner address of the contract
    */
    function changeOwner(address payable _newOwner) external {
        require(msg.sender == _owner);
        _owner = _newOwner;
        emit NewOwner(_newOwner);
    }
```

code/contracts/ConjureFactory.sol#L279-L285

```
    /**
     * @dev lets the owner collect the fees accrued
    */
    function collectFees() external {
        require(msg.sender == _owner, "Only owner");
        _owner.transfer(address(this).balance);
    }
```

However, EtherCollateral inherits an `Owned` contract which brings the same needed functionality.

code/contracts/EtherCollateralFactory.sol#L11

```
import "./Owned.sol";
```

code/contracts/EtherCollateralFactory.sol#L19

```
contract EtherCollateral is ReentrancyGuard, Owned {
```

### Recommendation

Use the same Owned pattern everywhere to increase readability, reduce code duplication and increase code usage.

---

## Multiple optimizations can be done in `getLatestETHUSDPrice`

Status Open Severity Medium

### Description

The method can be improved in a few ways

code/contracts/ConjureFactory.sol#L303-L321

```
    /**
     * @dev gets the latest ETH USD Price from the given oracle
     *
     * @return the current eth usd price
    */
    function getLatestETHUSDPrice() public view returns (int) {
        AggregatorV3Interface priceFeed = ethusdchainlinkoracle;
        (
        ,
        int price,
        ,
        ,
        ) = priceFeed.latestRoundData();

        uint returnDecimals = priceFeed.decimals();
        uint tempprice = uint(price) * 10 ** (_maximumDecimals - returnDecimals);

        return int(tempprice);
    }
```

This operation doesn't do anything, and the `ethusdchainlinkoracle` value can be used directly.

code/contracts/ConjureFactory.sol#L309

```
        AggregatorV3Interface priceFeed = ethusdchainlinkoracle;
```

The variable can be defined as immutable or constant because it will never change. Having it out of the contract state will decrease gas cost significantly.

code/contracts/ConjureFactory.sol#L109-L115

```
    // the eth usd price feed chainlink oracle address
    //chainlink eth/usd mainnet: 0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419
    //chainlink eth/usd rinkeby: 0x8A753747A1Fa494EC906cE90E9f37563A8AF630e
    AggregatorV3Interface public ethusdchainlinkoracle = AggregatorV3Interface(
        0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419

    );
```

The decimal number is not expected to change and could be hardcoded in the contract or saved at deploy time.

code/contracts/ConjureFactory.sol#L317

```
        uint returnDecimals = priceFeed.decimals();
```

It's unclear why the price is typecast to `uint` and back to `int`.

code/contracts/ConjureFactory.sol#L318-L320

```
        uint tempprice = uint(price) * 10 ** (_maximumDecimals - returnDecimals);

        return int(tempprice);
```

We see the `int` is typecast back into `uint` before being used in `getInternalPrice`.

code/contracts/ConjureFactory.sol#L483

```
                uint currentethtusdprice = uint(getLatestETHUSDPrice());
```

### Recommendation

Clean up the method and optimize calculation.

---

## Use the "Checks Effects Interactions" pattern

Status Open Severity Medium

### Description

There are a few cases where the "Checks Effects Interactions" is not followed and can create problems.

The require should be moved before initializing the structure.

code/contracts/ConjureFactory.sol#L194-L205

```
        for (uint i = 0; i < oracleAddresses_.length; i++) {
            _oracleStruct memory temp_struct;
            temp_struct.oracleaddress = oracleAddresses_[i];
            temp_struct.oracleType = oracleTypes_[i];
            temp_struct.signature = signatures_[i];
            temp_struct.calldatas = calldata_[i];
            temp_struct.weight = weights_[i];
            temp_struct.values = values_[i];
            temp_struct.decimals = decimals_[i];
            _oracleData.push(temp_struct);

            require(decimals_[i] <= 18, "Decimals too high");
```

The ether transfer should be moved at the end of the method.

code/contracts/EtherCollateralFactory.sol#L624-L645

```
        // Send liquidated ETH collateral to msg.sender
        msg.sender.transfer(totalCollateralLiquidated);

        // check if we have a full closure here
```

```
        if (amountToLiquidate >= amountOwed) {
            _closeLoan(synthLoan.account, synthLoan.loanID, true);
            // emit loan liquidation event
            emit LoanLiquidated(
                _loanCreatorsAddress,
                _loanID,
                msg.sender
            );
        } else {
            // emit loan liquidation event
            emit LoanPartiallyLiquidated(
                _loanCreatorsAddress,
                _loanID,
                msg.sender,
                amountToLiquidate,
                totalCollateralLiquidated
            );
        }
```

### Recommendation

Update code in order to more closely follow the "Checks Effects Interactions" pattern.

### References

- Solidity's Documentation
- Checks Effects Interactions

---

# Improve the square root computation

Status Open Severity Medium

### Description

Consider these results obtained by calling the implemented `sqrt` method.

| | Input number | ConjureFactory.sqrt | Approx.sqrt |
| --- | --- | --- | --- |
| 2**2 | 4 | 2 | 2 |
| 2**6 | 64 | 32 | 8 |
| | 99 | 50 | 9 |
| | 24660535258592184423 9 | 15703673219534397973 | 15703673219 |
| 2**64 | 18446744073709551616 | 4294967296000000000 | 4294967296 |
| | 9999999999999 | 5000000000000 | 3162277 |
| | 10 | 5 | 3 |

The first column represents a short notation of the tested number to guide the reader to the result. `Input number` represents the number used in each implementation as input.

`ConjureFactory.sqrt` represents the result of the currently implemented squared root. `Approx.sqrt` represents the result of the implementation below.

```
contract Approx {
    /**
    * @dev implementation of a square rooting algorithm
    *
    * @param y the value to be square rooted
    * @return z the square rooted value
    */
    function sqrt(uint256 y) public view returns (uint256 z) {
        if (y > 3) {
            z = y;
            uint256 x = (y + 1) / 2;
            while (x < z) {
                z = x;
                x = (y / x + x) / 2;
            }
        } else if (y != 0) {
            z = 1;
        }
        // else z = 0
    }
}
```

This implementation has a few changes compared to the `ConjureFactory.sqrt`:

- It does not use `SafeMath` because, within certain constraints, the results are always correct.
- It does not multiply by the `UNIT` (this creates incorrect results)

code/contracts/ConjureFactory.sol#L398

```
                x = (y.mul(UNIT).div(x) + x) / 2;
```

### Recommendation

Check other sqrt implementations, write tests to validate your own implementation and rewrite the method.

### References

Consider using ABDK's sqrt implementation, but be mindful of the specific implementation they use because they support a 64.64-bit fixed point representation

ABDKMath64x64.sol#L350-L359

```
  /**
   * Calculate sqrt (x) rounding down.  Revert if x < 0.
   *
```

```
     * @param x signed 64.64-bit fixed point number
     * @return signed 64.64-bit fixed point number
     */
    function sqrt (int128 x) internal pure returns (int128) {
      require (x >= 0);
      return int128 (sqrtu (uint256 (x) << 64));
    }
```

Or using Uniswap's implementation.

contracts/libraries/Math.sol#L10-L11

```
    // babylonian method (https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylo
    function sqrt(uint y) internal pure returns (uint z) {
```

---

## Make `calculateAmountToLiquidate` pure for a significant gas discount

Status Open Severity Minor

### Description

The method `calculateAmountToLiquidate` takes the `debtBalance` and the `collateral` and returns the amount to liquidate.

code/contracts/EtherCollateralFactory.sol#L258-L274

```
    /**
    * @dev Gets the amount to liquidate which can potentially fix the c ratio given this formula
     * r = target issuance ratio
     * D = debt balance
     * V = Collateral
     * P = liquidation penalty
     * Calculates amount of synths = (D - V * r) / (1 - (1 + P) * r)
     *
     * If the C-Ratio is greater than Liquidation Ratio + Penalty in % then the C-Ratio can be f
     * otherwise a greater number is returned and the debttoCover from the calling function is u
     *
     * @param debtBalance the amount of the loan or debt to calculate in USD
     * @param collateral the amount of the collateral in USD
     *
     * @return the amount to liquidate to fix the C-Ratio if possible
     */
    function calculateAmountToLiquidate(uint debtBalance, uint collateral) public view returns (
```

The calculation is based on 2 internal state variables `liquidationRatio` and `liquidationPenalty`. It doesn't need external information and it doesn't change state.

This is why the method is defined as `view`.

The state variable `liquidationRatio` is set in the constructor and it's never changed.

code/contracts/EtherCollateralFactory.sol#L121

```
        liquidationRatio = _ratio / 100;
```

Similarly, the state variable `liquidationPenalty` is set at deploy time and also never changed.

code/contracts/EtherCollateralFactory.sol#L47

```
    uint256 public liquidationPenalty = SafeDecimalMath.unit() / 10;
```

Because they are both set at deploy time and never changed, they can be defined as `immutable`. This would allow the method `calculateAmountToLiquidate` to be defined as `pure` and would save a lot of gas when executed.

code/contracts/EtherCollateralFactory.sol#L43-L47

```
    // Liquidation ratio when loans can be liquidated
    uint256 public liquidationRatio;

    // Liquidation penalty when loans are liquidated. default 10%
    uint256 public liquidationPenalty = SafeDecimalMath.unit() / 10;
```

### Recommendation

Define the variables `liquidationRatio` and `liquidationPenalty` as immutable and set the method `calculateAmountToLiquidate` as pure for a significant gas discount.

### References

- Constant and Immutable State Variables

---

## Unclear documentation and require message in setAccountLoanLimit function


### Description

The documentation for `setAccountLoanLimit` seems outdated:

code/contracts/EtherCollateralFactory.sol#L143-L149

```
    /**
     * @dev Sets the account loan limit to the desired value
     * array indicating which tokens had their prices updated.
     *
     * @param _loanLimit the new account loan limit
    */
    function setAccountLoanLimit(uint256 _loanLimit) external onlyOwner {
```

The `require` message is confusing to the reader:

code/contracts/EtherCollateralFactory.sol#L150

```
        require(_loanLimit < ACCOUNT_LOAN_LIMIT_CAP, "Owner cannot set higher than ACCOUNT_LOAN_
```

It states that the owner cannot set a higher value, but the check also doesn't allow an equal value.

**Recommendation**

Update the documentation to reflect in which case the value provided is valid.

Update the code or the error message to be consistent.

---

# Unecessary temp_struct in Conjure.init() function


**Description**

The `temp_struct` variable is unecessary in the `Conjure.init()` function:

code/contracts/ConjureFactory.sol#L194-L203

```
        for (uint i = 0; i < oracleAddresses_.length; i++) {
            _oracleStruct memory temp_struct;
            temp_struct.oracleaddress = oracleAddresses_[i];
            temp_struct.oracleType = oracleTypes_[i];
            temp_struct.signature = signatures_[i];
            temp_struct.calldatas = calldata_[i];
            temp_struct.weight = weights_[i];
            temp_struct.values = values_[i];
            temp_struct.decimals = decimals_[i];
            _oracleData.push(temp_struct);
```

This leads to higher gas usage than necessary especially since it happens within a loop.

**Recommendation**

Rewrite the code as:

```
_oracleData.push(_oracleStruct({
        oracleaddress: oracleAddresses_[i],
        ....
}))
```

This will increase readability and ensure all the fields are updated.

---

# Improve constructor efficiency in Conjure

Status Open Severity Minor

### Description

The Conjure constructor is implemented as

code/contracts/ConjureFactory.sol#L117-L139

```
    constructor (
        string memory name_,
        string memory symbol_,
        address payable owner_,
        address factoryaddress_,
        address uniswapv2oracle,
        address collateralfactory_
    )
    {
        _owner = owner_;
        _factoryaddress = factoryaddress_;
        _totalSupply = 0;
        _name = name_;
        _symbol = symbol_;

        _uniswapv2oracle = UniswapV2OracleInterface(uniswapv2oracle);
        _collateralFactory = collateralfactory_;

        _balances[_owner] = _totalSupply;
        _inited = false;

        emit Transfer(address(0), _owner, _totalSupply);
    }
```

There are a few inefficiencies that stand out.

These operations do not change anything in the constructor state.

code/contracts/ConjureFactory.sol#L128

```
        _totalSupply = 0;
```

code/contracts/ConjureFactory.sol#L135

```
        _balances[_owner] = _totalSupply;
```

code/contracts/ConjureFactory.sol#L136

```
        _inited = false;
```

The values start as zero or false, depending on the variable type. This is why they could be removed without changing anything in how the contract behaves.

We're not sure if the `_owner` should own any tokens, because we see this assignment and the value assigned is zero.

code/contracts/ConjureFactory.sol#L135

```
        _balances[_owner] = _totalSupply;
```

Also, a `Transfer` event is emitted, even though the value is zero.

code/contracts/ConjureFactory.sol#L138

```
        emit Transfer(address(0), _owner, _totalSupply);
```

**Recommendation**

Make sure the operations make sense and they don't need to be updated.

If you decide they are correct as they are, you should remove the dead code.

Add tests that validate the correct, expected behavior.

---

# Slightly improve gas costs when saving oracle weights


**Description**

Some variables are defined as `uint8` as opposed to a `uint256`. Having `uint8` can sometimes save gas if multiple variables are saved closely together. Solidity is able to pack multiple items that need less than 32 bytes in one storage slot.

In this case, this cannot be applied because the variables around the `uint8 weight;` do not use less than 32 bytes.

code/contracts/ConjureFactory.sol#L67-L77

```
    // struct for oracles
    struct _oracleStruct {
        address oracleaddress;
        // 0... chainlink, 1... uniswap twap, 2... custom
        uint oracleType;
        string signature;
        bytes calldatas;
        uint8 weight;
        uint256 decimals;
        uint256 values;
    }
```

When calling the `Conjure.init` method there is actually more gas used to process the `weights_[i]` item.

To reflect this increased gas cost, we created this example.

```
contract A8 {
    struct OracleData {
        bytes list;
        uint8 number;
    }

    OracleData public od;

    function save(bytes memory list, uint8 number) public {
        od.list = list;
        od.number = number;
    }
}

contract A256 {
    struct OracleData {
        bytes list;
        uint number;
    }

    OracleData public od;

    function save(bytes memory list, uint number) public {
        od.list = list;
        od.number = number;
    }
}
```

Gas usage is as follows during each `.save` call.

- `A8.save(0x1122334455, 200)` consumes `42671 gas`
- `A256.save(0x1122334455, 200)` consumes `41782 gas`

One can see there's less gas consumed when `number` is defined as `uint` instead of `uint8`.

Having the `weight` defined as `uint256` will not decrease the contract's security because the following check is made.

code/contracts/ConjureFactory.sol#L200

```
            temp_struct.weight = weights_[i];
```

code/contracts/ConjureFactory.sol#L206

```
            weightcheck += weights_[i];
```

code/contracts/ConjureFactory.sol#L209-L212

```
        // for basket assets weights must add up to 100
        if (_assetType == 1) {
            require(weightcheck == 100, "Weights not 100");
        }
```

### Recommendation

If there's no other reason, feel free to define the `weight` as `uint256` for decreased gas costs.

### References

- Layout of State Variables in Storage

---

## Inconsistent use of arbasset

Status Open Severity Informational

### Description

The `syntharb` is a helper function that typecasts `arbasset` to `IConjure` interface:

code/contracts/EtherCollateralFactory.sol#L813-L815

```
    function syntharb() internal view returns (IConjure) {
        return IConjure(arbasset);
    }
```

In some places of the code `arbasset` is cast to `IConjure` manually, without making use of the `syntharb` helper function:

code/contracts/EtherCollateralFactory.sol#L220

```
        uint currentprice = IConjure(arbasset).getPrice();
```

While in other areas the `syntharb` function is used:

code/contracts/EtherCollateralFactory.sol#L546-L547

```
        // burn funds from msg.sender for repaid amount
        syntharb().burn(msg.sender, _repayAmount);
```

### Recommendation

Use only one consistent method to cast `arbasset` to `IConjure`.

---

## Code style is inconsistent

Status Open Severity Informational

### Description

There are several places where the Solidity Style Guide could be used to improve the readability of the code.

code/contracts/EtherCollateralFactory.sol#L239-L242

```
        return
        loanAmount
        .multiplyDecimal(collateralizationRatio.divideDecimalRound(currentethusdprice).multiplyD
        .divideDecimalRound(ONE_HUNDRED);
```

The Style Guide also addresses the order layout of a contract. In the following example, the events are defined at the end of the contract but it would be more helpful to follow the Style Guide and place them at the top after the state variables declarations:

code/contracts/EtherCollateralFactory.sol#L817-L823

```
    // ========== EVENTS ==========

    event IssueFeeRateUpdated(uint256 issueFeeRate);
    event AccountLoanLimitUpdated(uint256 loanLimit);
    event LoanLiquidationOpenUpdated(bool loanLiquidationOpen);
    event LoanCreated(address indexed account, uint256 loanID, uint256 amount);
    event LoanClosed(address indexed account, uint256 loanID);
```

Another instance of inconsistent style is the method name `openLoanIDsByAccount`. Usually a verb should be the first word of the method (get, set, open, close, liquidate, ...). In this case the method does not open loans, but returns opened loans.

code/contracts/EtherCollateralFactory.sol#L284-L290

```
    /**
     * @dev Gets all open loans by a given account address
     *
     * @param _account the opener of the loans
     * @return all open loans by ID in form of an array
    */
    function openLoanIDsByAccount(address _account) external view returns (uint256[] memory) {
```

### Recommendation

Please consider following the Solidity Style Guide and NatSpec doc format.

Follow a consistent naming scheme for state variables and methods.

---

## Reuse `setIssueFeeRate` in constructor

Status Open Severity Informational

### Description

The `EtherCollateral` constructor can reuse the code from `setIssueFeeRate` to process the `_mintingfeerate` argument.

This is the code in the constructor:

code/contracts/EtherCollateralFactory.sol#L112-L113

```
        // max 2.5% fee for minting
        require(_mintingfeerate <= 250, "Minting fee too high");
```

This is the same code duplicated in `setIssueFeeRate` function:

code/contracts/EtherCollateralFactory.sol#L131-L133

```
    function setIssueFeeRate(uint256 _issueFeeRate) external onlyOwner {
        // max 2.5% fee for minting
        require(_issueFeeRate <= 250, "Minting fee too high");
```

### Recommendation

Call the `setIssueFeeRate` function in the constructor and remove the duplicated code that processes the `_mintingfeerate` argument.

---

## Avoid using underflows for defining max uint values

Status Open Severity Informational

### Description

There are several places where uint underflow is used to specify a max value for uint256 or uint96 (grep `uint\d+\(-1\)`):

code/contracts/CNJ.sol#L141-L147

```
    function approve(address spender, uint256 rawAmount) external returns (bool) {
        uint96 amount;
        if (rawAmount == uint256(-1)) {
            amount = uint96(-1);
        } else {
            amount = safe96(rawAmount, "Cnj::approve: amount exceeds 96 bits");
        }
```

code/contracts/CNJ.sol#L165-L167

```
        if (rawAmount == uint256(-1)) {
            amount = uint96(-1);
        } else {
```

code/contracts/ConjureFactory.sol#L699

```
        if (spender != src && spenderAllowance != uint256(-1)) {
```

This method of specifying a max uint is a bit hacky and not necessarily easy to read.

Note that, starting with Solidity v0.8.0 such an expression will revert.

### Recommendation

Compute max for uint256 as `type(uint256).max` and uint96 as `type(uint96).max`. This has been implemented since Solidity v0.6.8:

> Language Features: Implemented type(T).min and type(T).max for every integer type T that returns the smallest and largest value representable by the type.

You can store the value in a constant to reduce the gas costs. It has the advantage of being easier to read, more standardized, and compatible with future versions of solidity.

---

## Pin the solidity version number

Status Open Severity Informational

### Description

The solidity version number is not pinned (note the `^` character):

code/contracts/ConjureFactory.sol#L1-L3

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;
pragma experimental ABIEncoderV2;
```

This can result in several issues because the compilation output is unpredictable: the solidity version will change with every path release.

**Recommendation**

Use the same fixed version across the project. For example, to use the latest stable v0.7 Solidity version:

```
pragma solidity 0.7.6;
```

Note there's no `^` prefix character to the version number. This is on purpose: upgrading the Solidity version number in a project should be a conscious decision not left to the packaging system (npm or yarn).

---

# License