SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Notional
Prepared by: Sherlock
Lead Security Expert: hyh
Dates Audited: April 10 - April 15, 2023
Prepared on: May 2, 2023

Introduction

Earn fixed income on your crypto or borrow at fixed rates for up to one year with Notional - DeFi's top fixed rate protocol.

Scope

Repository: notional-finance/contracts-v2-private

Branch: replacing-ctokens

Commit: 70c179a9df9c72137b8e836ce61e4e7e00541493

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
3	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

neumo
hyh
ck
hansfriese

Issue M-1: Minting doesn't control for resulting zero shares

Source: https://github.com/sherlock-audit/2023-04-notional-judging/issues/16

Found by

ck, hyh

Summary

It is possible to provide some amount of underlying to nwToken's mint and receive zero shares, effectively losing the investment.

Vulnerability Detail

The situation of non-zero underlying investment and zero nwTokens to be minted is a direct asset loss for a user and is not currently controlled.

Impact

Small amounts user provide can be lost fully. The damage looks to be mostly reputational, although some downstream systems can malfunction on receiving non-positive share amount.

Code Snippet

Both ETH and ERC20 versions of mint() rounds the number of shares down, but do not control for the resulting value to be positive:

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L68-L80

```
// CEtherInterface functions
function mint() external payable nonReentrant override {
    require(UNDERLYING_TOKEN == ETH_ADDRESS);
    require(finalExchangeRate != 0);

    if (msg.value == 0) return;

    uint256 assetTokenAmount = _convertToAsset(msg.value);

    // Handles event emission, balance update and total supply update
    super._mint(msg.sender, assetTokenAmount);
    _checkSupplyInvariant();
}
```

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L82-L101

```
// CErc20Interface functions
function mint(uint mintAmount) external nonReentrant override returns (uint) {
    require(UNDERLYING_TOKEN != ETH_ADDRESS);
    require(finalExchangeRate != 0);

    if (mintAmount == 0) return NO_ERROR;

    ERC20(UNDERLYING_TOKEN).safeTransferFrom(
        msg.sender,
        address(this),
        mintAmount
    );
    uint256 assetTokenAmount = _convertToAsset(mintAmount);

    // Handles event emission, balance update and total supply update
    super._mint(msg.sender, assetTokenAmount);

    _checkSupplyInvariant();
    return NO_ERROR;
}
```

Tool used

Manual Review

Recommendation

Consider adding such control to prevent pure asset loss for a user:

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L68-L80

```
    // CEtherInterface functions
    function mint() external payable nonReentrant override {
        require(UNDERLYING_TOKEN == ETH_ADDRESS);
        require(finalExchangeRate != 0);

        if (msg.value == 0) return;

        uint256 assetTokenAmount = _convertToAsset(msg.value);
+       require(assetTokenAmount != 0, "Zero shares");

        // Handles event emission, balance update and total supply update
        super._mint(msg.sender, assetTokenAmount);
```

```
        _checkSupplyInvariant();
    }
```

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L82-L101

```
    // CErc20Interface functions
    function mint(uint mintAmount) external nonReentrant override returns (uint)
↪   {
        require(UNDERLYING_TOKEN != ETH_ADDRESS);
        require(finalExchangeRate != 0);

        if (mintAmount == 0) return NO_ERROR;

        ERC20(UNDERLYING_TOKEN).safeTransferFrom(
            msg.sender,
            address(this),
            mintAmount
        );
        uint256 assetTokenAmount = _convertToAsset(mintAmount);
+       require(assetTokenAmount != 0, "Zero shares");

        // Handles event emission, balance update and total supply update
        super._mint(msg.sender, assetTokenAmount);

        _checkSupplyInvariant();
        return NO_ERROR;
    }
```

Discussion

hrishibhat

Fix: https://github.com/notional-finance/contracts-v2/commit/1845605ab0d9eec9b5dd374cf7c246957b534f85

dmitriia

Fix looks ok

Issue M-2: token() function should return the address of the nwToken not the cToken

Source: https://github.com/sherlock-audit/2023-04-notional-judging/issues/11

Found by

neumo

Summary

In contract `nwToken`, function `token()` should return the address of the asset token which, after migration, should be the `nwToken` address and not the `cToken` from which the funds were migrated.

Vulnerability Detail

The functionalities Notional uses related to Compound V2 are located in contracts `CompoundHandler` and `cTokenAggregator`, and in the scope of this contest, they are merged into contract `nwToken`.

Function `token()` is a function of `AssetRateAdapter` contract, which is the base contract of `cTokenAggregator`, and is supposed to return the address of the new wrapper token after migration `nwToken`, but instead returns the address of the old `cToken`. This fact makes not possible to use the `nwToken` as asset rate oracle when enabling cash groups for a currency.

From contract `GovernanceAction`:

```
function enableCashGroup(
    uint16 currencyId,
    AssetRateAdapter assetRateOracle,
    CashGroupSettings calldata cashGroup,
    string calldata underlyingName,
    string calldata underlyingSymbol
) external override onlyOwner {
    _checkValidCurrency(currencyId);
    {
        // Cannot enable fCash trading on a token with a max collateral balance
        Token memory assetToken = TokenHandler.getAssetToken(currencyId);
        Token memory underlyingToken =
↪   TokenHandler.getUnderlyingToken(currencyId);
        require(
            assetToken.maxCollateralBalance == 0 &&
            underlyingToken.maxCollateralBalance == 0
        ); // dev: cannot enable trading, collateral cap
    }
```

```
    _updateCashGroup(currencyId, cashGroup);
    _updateAssetRate(currencyId, assetRateOracle);

    // Creates the nToken erc20 proxy that routes back to the main contract
    nTokenERC20Proxy proxy = new nTokenERC20Proxy(
        nTokenERC20(address(this)),
        currencyId,
        underlyingName,
        underlyingSymbol
    );
    nTokenHandler.setNTokenAddress(currencyId, address(proxy));
    emit DeployNToken(currencyId, address(proxy));
}
...
function _updateAssetRate(uint16 currencyId, AssetRateAdapter rateOracle)
↪   internal {
    // If rate oracle refers to address zero then do not apply any updates here,
↪   this means
    // that a token is non mintable.
    Token memory assetToken = TokenHandler.getAssetToken(currencyId);
    if (address(rateOracle) == address(0)) {
        // Sanity check that unset rate oracles are only for non mintable tokens
        require(assetToken.tokenType == TokenType.NonMintable, "G: invalid asset
↪   rate");
    } else {
        // Sanity check that the rate oracle refers to the proper asset token
        address token = AssetRateAdapter(rateOracle).token();
        require(assetToken.tokenAddress == token, "G: invalid rate oracle");

        uint8 underlyingDecimals;
        if (currencyId == Constants.ETH_CURRENCY_ID) {
            // If currencyId is one then this is referring to cETH and there is
↪   no underlying() to call
            underlyingDecimals = Constants.ETH_DECIMAL_PLACES;
        } else {
            address underlyingTokenAddress =
↪   AssetRateAdapter(rateOracle).underlying();
            Token memory underlyingToken =
↪   TokenHandler.getUnderlyingToken(currencyId);
            // Sanity check to ensure that the asset rate adapter refers to the
↪   correct underlying
            require(underlyingTokenAddress == underlyingToken.tokenAddress, "G:
↪   invalid adapter");
            underlyingDecimals = ERC20(underlyingTokenAddress).decimals();
        }

        // Perform this check to ensure that decimal calculations don't overflow
```

```
        require(underlyingDecimals <= Constants.MAX_DECIMAL_PLACES);
        mapping(uint256 => AssetRateStorage) storage store =
↪   LibStorage.getAssetRateStorage();
        store[currencyId] = AssetRateStorage({
            rateOracle: rateOracle,
            underlyingDecimalPlaces: underlyingDecimals
        });

        emit UpdateAssetRate(currencyId);
    }
}
```

We see that when Governance tries to enable a cash group, the user passes to the `enableCashGroup` an asset rate oracle that is used in the internal call to `_updateAssetRate`. The following lines would make the call revert if trying to pass the `nwToken` address as rate oracle:

```
...
address token = AssetRateAdapter(rateOracle).token();
require(assetToken.tokenAddress == token, "G: invalid rate oracle");
...
```

because token would be the old `cToken` but `assetToken.tokenAddress` would be the address of `nwToken`.

After migration, the rate adapter for each of the four currencies migrated is set to the `nwToken`: https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/patchfix/MigrateCTokens.sol#L107-L109

As we can see in the tests, all four nwTokens are created passing the address of the old `cToken` as `COMPOUND_TOKEN` in the constructor.

Impact

Medium

Code Snippet

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L147-L149

Tool used

Manual review.

Recommendation

Change the following function in contract `nwToken`:

```
function token() external view returns (address) {
    return COMPOUND_TOKEN;
}
```

With this:

```
function token() external view returns (address) {
    return address(this);
}
```

Discussion

jeffywu

Valid, good catch.

hrishibhat

Fix: https://github.com/notional-finance/contracts-v2/commit/1845605ab0d9eec9b5dd374cf7c246957b534f85

dmitriia

Fix looks ok

Issue M-3: redeemUnderlying doesn't round up, can provide free underlying for the shares given

Source: https://github.com/sherlock-audit/2023-04-notional-judging/issues/10

Found by

hansfriese, hyh

Summary

All nwToken asset to underlying and back translations rounds down. This allows for withdrawing more than supplying in a `supply floor underlying amount, withdraw ceiling amount` manner.

Vulnerability Detail

Let's suppose it's nwETH, so underlying is ETH with 18 dp, token is cETH with 8 dp. `finalExchangeRate` needs to have `x = 28` dp in order to _convertToAsset's `underlyingAmount * EXCHANGE_RATE_PRECISION / finalExchangeRate` having cETH's `(18 + 18 - x) = 8` dp result, and _convertToUnderlying's `assetAmount * finalExchangeRate / EXCHANGE_RATE_PRECISION` having ETH's `(8 + x - 18) = 18` dp result.

Let's say `finalExchangeRate = 200822050757246498024651213` (it's cETH rate on mainnet as of time of this writing: https://etherscan.io/token/0x4ddc2d193948926d02f9b1fe9e1daa0718270ed5#readContract).

Bob calls mint() with `msg.value = 1 eth`, obtains `underlyingAmount * EXCHANGE_RATE_PRECISION / finalExchangeRate = 1e18 * 1e18 / 200822050757246498024651213 = 4979532856` nwETH, then calls `redeemUnderlying(1000000000150000000)`, which burns the same `underlyingAmount * EXCHANGE_RATE_PRECISION / finalExchangeRate = 1000000000150000000 * 1e18 / 200822050757246498024651213 = 4979532856` nwETH, providing Bob with free `150000000 wei`.

Impact

Bob can obtain all excess funds from the contract this way as long as _checkSupplyInvariant() allows.

Since that's excess funds only, setting the severity to be medium.

Code Snippet

redeemUnderlying() calls _convertToAsset() to get nwToken amount from the underlying amount requested:

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L116-L127

```
function redeemUnderlying(uint redeemAmount) external nonReentrant override
↪  returns (uint) {
    if (redeemAmount == 0) return NO_ERROR;
    require(finalExchangeRate != 0);

    // Handles event emission, balance update and total supply update
    super._burn(msg.sender, _convertToAsset(redeemAmount));

    _transferUnderlyingToSender(redeemAmount);

    _checkSupplyInvariant();
    return NO_ERROR;
}
```

_convertToAsset() always rounds down:

https://github.com/sherlock-audit/2023-04-notional/blob/main/contracts-v2-private/contracts/external/adapters/nwToken.sol#L180-L182

```
function _convertToAsset(uint256 underlyingAmount) private view returns
↪  (uint256) {
    return underlyingAmount * EXCHANGE_RATE_PRECISION / finalExchangeRate;
}
```

I.e. the shares required for the given amount of underlying are rounded down and so some amount of underlying within the same shared count can be obtained for free.

Tool used

Manual Review

Recommendation

_checkSupplyInvariant() looks to be handling the overall solvency of the contract.

To be on the safe side consider rounding up the number of shares required for a given amount of underlying in redeemUnderlying().

Discussion

hrishibhat

Fix: https://github.com/notional-finance/contracts-v2/commit/1845605ab0d9eec9b5dd374cf7c246957b534f85

dmitriia

Fix looks ok