Notional x Index Coop Findings & Analysis Report

2022-07-18

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Notional x Index Coop smart contract system written in Solidity. The audit contest took place between June 7—June 14 2022.

Wardens

81 Wardens contributed reports to the Notional x Index Coop:

1. jonah1005
2. unforgiven
3. xiaoming90
4. csanuragjain
5. 0xDjango
6. berndartmueller
7. GreyArt (hickuphh3 and itsmeSTYJ)
8. Meera
9. minhquanym
10. kenzo
11. 0x52
12. IIIIIII
13. antonttc
14. GimelSec (rayn and sces60107)
15. sorrynotsorry
16. joestakey
17. 0xkatana
18. 0x29A (0x4non and rotcivegaf)

19. 0x1f8b
20. 0xf15ers (remora and twojoy)
21. Funen
22. hyh
23. PierrickGT
24. Chom
25. delfin454000
26. Waze
27. Picodes
28. Deivitto
29. fatherOfBlocks
30. 0xKitsune
31. TomJ
32. TerrierLover
33. simon135
34. _Adam
35. slywaters
36. oyc_109
37. ellahi
38. saian
39. sach1r0
40. catchup
41. c3phas
42. Sm4rty
43. Cityscape
44. 0xNazgul
45. hake
46. 0xmint
47. Tadashi

48. Lambda
49. hansfriese
50. Ruhum
51. zzzitron
52. Trumpero
53. sseefried
54. cloudjunky
55. cccz
56. Bronicle
57. Nethermind
58. dipp
59. cryptphi
60. 0xNineDec
61. z3s
62. JC
63. ayeslick
64. Tomio
65. rfa
66. 8olidity
67. 0xSolus
68. UnusualTurtle
69. asutorufos
70. samruna
71. kaden
72. ElKu
73. DavidGialdi
74. 0v3rf10w
75. ynnad
76. Fitraldys

77. djxploit

This contest was judged by gzeon.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 11 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 10 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 61 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 55 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Notional x Index Coop repository, and is composed of 5 smart contracts written in the Solidity programming language and includes 914 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (1)

[H-01] Rounding Issues In Certain Functions

Submitted by xiaoming90, also found by berndartmueller, GreyArt, jonah1005, kenzo, and minhquanym

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L52

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L134

Background

Per EIP 4626's Security Considerations (https://eips.ethereum.org/EIPS/eip-4626)

> Finally, ERC-4626 Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users:

- If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round *down*.
- If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round *up*.

Thus, the result of the `previewMint` and `previewWithdraw` should be rounded up.

Proof of Concept

The current implementation of `convertToShares` function will round down the number of shares returned due to how solidity handles Integer Division. ERC4626 expects the returned value of `convertToShares` to be rounded down. Thus, this function behaves as expected.

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L52

```
    function convertToShares(uint256 assets) public view override re
        uint256 supply = totalSupply();
        if (supply == 0) {
            // Scales assets by the value of a single unit of fCash
            uint256 unitfCashValue = _getPresentValue(uint256(Consta
            return (assets * uint256(Constants.INTERNAL_TOKEN_PRECIS
        }

        return (assets * totalSupply()) / totalAssets();
    }
```

ERC 4626 expects the result returned from `previewWithdraw` function to be rounded up. However, within the `previewWithdraw` function, it calls the `convertToShares` function. Recall earlier that the `convertToShares` function returned a rounded down value, thus `previewWithdraw` will return a rounded down value instead of round up value. Thus, this function does not behave as expected.

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L134

```
    function previewWithdraw(uint256 assets) public view override re
        if (hasMatured()) {
            shares = convertToShares(assets);
        } else {
            // If withdrawing non-matured assets, we sell them on th
```

```
            (uint16 currencyId, uint40 maturity) = getDecodedID();
            (shares, /* */, /* */) = NotionalV2.getfCashBorrowFromPr
                currencyId,
                assets,
                maturity,
                0,
                block.timestamp,
                true
            );
        }
    }
```

`previewWithdraw` and `previewMint` functions rely on `NotionalV2.getfCashBorrowFromPrincipal` and `NotionalV2.getDepositFromfCashLend` functions. Due to the nature of time-boxed contest, I was unable to verify if `NotionalV2.getfCashBorrowFromPrincipal` and `NotionalV2.getDepositFromfCashLend` functions return a rounded down or up value. If a rounded down value is returned from these functions, `previewWithdraw` and `previewMint` functions would not behave as expected.

Impact

Other protocols that integrate with Notional's fCash wrapper might wrongly assume that the functions handle rounding as per ERC4626 expectation. Thus, it might cause some intergration problem in the future that can lead to wide range of issues for both parties.

Recommended Mitigation Steps

Ensure that the rounding of vault's functions behave as expected. Following are the expected rounding direction for each vault function:

- previewMint(uint256 shares) - Round Up ⬆
- previewWithdraw(uint256 assets) - Round Up ⬆
- previewRedeem(uint256 shares) - Round Down ⬇
- previewDeposit(uint256 assets) - Round Down ⬇
- convertToAssets(uint256 shares) - Round Down ⬇

- convertToShares(uint256 assets) - Round Down ⬇

`previewMint` returns the amount of assets that would be deposited to mint specific amount of shares. Thus, the amount of assets must be rounded up, so that the vault won't be shortchanged.

`previewWithdraw` returns the amount of shares that would be burned to withdraw specific amount of asset. Thus, the amount of shares must to be rounded up, so that the vault won't be shortchanged.

Following is the OpenZeppelin's vault implementation for rounding reference:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20TokenizedVault.sol

Alternatively, if such alignment of rounding could not be achieved due to technical limitation, at the minimum, document this limitation in the comment so that the developer performing the integration is aware of this.

jeffywu (Notional) confirmed

gzeoneth (judge) increased severity to High and commented:

> Judging this and all duplicate regarding EIP4626 implementation as High Risk.

> EIP4626 is aimed to create a consistent and robust implementation patterns for Tokenized Vaults. A slight deviation from 4626 would broke composability and potentially lead to loss of fund (POC in https://github.com/code-423n4/2022-06-notional-coop-findings/issues/88 can be an example). It is counterproductive to implement EIP4626 but does not conform to it fully. Especially it does seem that most of the time `deposit` would be successful but not `withdraw`, making it even more dangerous when an immutable consumer application mistakenly used the wfcash contract.

Medium Risk Findings (10)

[M-01] fCash of the wrong maturity and asset can be sent to wrapper address before wrapper is deployed

Submitted by 0x52, also found by jonah1005 and unforgiven

Minting becomes impossible

Proof of Concept

onERC1155Received is only called when the size of the code deployed at the address contains code. Since create2 is used to deploy the contract, the address can be calculated before the contract is deployed. A malicious actor could send the address fCash of a different maturity or asset before the contract is deployed and since nothing has been deployed, onERC1155Received will not be called and the address will accept the fCash. After the contract is deployed and correct fCash is sent to the address, onERC1155Received will check the length of the assets held by the address and it will be more than 1 (fCash of correct asset and maturity and fCash with wrong maturity or asset sent before deployment). This will cause the contract to always revert essentially breaking the mint completely.

Recommended Mitigation Steps

When the contract is created create a function that reads how many fCash assets are at the address and send them away if they aren't of the correct asset and maturity

jeffywu (Notional) confirmed, but disagreed with severity and commented:

> I will need to write a PoC to confirm that this is the case but it seems plausible to me.

> Based on the Judging Criteria, this does not result in loss of funds. This will result in a loss of availability (available funds actually increase).

> My opinion is medium severity.

gzeoneth (judge) decreased severity to Medium and commented:

> Judging this as Med Risk due to no loss of funds and only possible before contract deployment.

[M-02] deposit() and mint() and _redeemInternal() in wfCashERC4626() will revert for all fcash that asset token is underlying token because they always call _mintInternal() with useUnderlying==True

Submitted by unforgiven

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L177-L184

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L168-L175

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L225-L241

Impact

For some `fcash` the asset token is underlying token (`asset.tokenType == TokenType.NonMintable`) and `NotionalV2` will not handle minting with `useUnderlying==True` for those `fcash` s (according to what I asked from sponsor). In summery most of the logics in `wfCashERC4626` will not work for those `fcash` tokens.

when for some `fcash` asset token is underlying token, all calls to `NotionalV2` should be with `useUnderlying==False` . but `deposit()` and `mint()` in `wfCashERC4626` contract call `_mintInternal()` with `useUnderlying==True` and it calls `NotionalV2.batchLend()` with `depositUnderlying==true` so the `NotionV2` call will fail for `fcash` tokens that asset token is underlying token and it would cause that `deposit()` and `mint()` logic `wfCashERC4626` will not work and

contract will be useless for those tokens. `_redeemInternal()` issue is similar and it calls `_burn()` with `redeemToUnderlying: true` which execution eventually calls `NotionalV2.batchBalanceAndTradeAction()` with `toUnderlying=True` which will revert so `_redeemInternal()` will fail and because `withdraw()` and `redeem` use it, so they will not work too for those `fcash` tokens that asset token is underlying token.

Proof of Concept

This is `deposit()` and `mint()` code in `wfCashERC4626`:

```
    /** @dev See {IERC4626-deposit} */
    function deposit(uint256 assets, address receiver) public ov
        uint256 shares = previewDeposit(assets);
        // Will revert if matured
        _mintInternal(assets, _safeUint88(shares), receiver, 0,
        emit Deposit(msg.sender, receiver, assets, shares);
        return shares;
    }

    /** @dev See {IERC4626-mint} */
    function mint(uint256 shares, address receiver) public overr
        uint256 assets = previewMint(shares);
        // Will revert if matured
        _mintInternal(assets, _safeUint88(shares), receiver, 0,
        emit Deposit(msg.sender, receiver, assets, shares);
        return assets;
    }
```

As you can see they both call `_mintInternal()` with last parameter as `true` which is `useUnderlying`'s value. This is `_mintInternal()` code:

```
    function _mintInternal(
        uint256 depositAmountExternal,
        uint88 fCashAmount,
        address receiver,
        uint32 minImpliedRate,
        bool useUnderlying
    ) internal nonReentrant {
        require(!hasMatured(), "fCash matured");
```

```
        (IERC20 token, bool isETH) = getToken(useUnderlying);
        uint256 balanceBefore = isETH ? address(this).balance :

        // If dealing in ETH, we use WETH in the wrapper instead
        // ETH natively but due to pull payment requirements for
        // ETH. batchLend only supports ERC20 tokens like cETH o
        // layer, it will support WETH so integrators can deal s
        // "batchLend" we will use "batchBalanceActionWithTrades
        // is more gas efficient (does not require and additiona
        // then everything will proceed via batchLend.
        if (isETH) {
            IERC20((address(WETH))).safeTransferFrom(msg.sender,
            WETH.withdraw(depositAmountExternal);

            BalanceActionWithTrades[] memory action = EncodeDeco
                getCurrencyId(),
                getMarketIndex(),
                depositAmountExternal,
                fCashAmount,
                minImpliedRate
            );
            // Notional will return any residual ETH as the nati
            // native ETH tokens will be wrapped back to WETH.
            NotionalV2.batchBalanceAndTradeAction{value: deposit
        } else {
            // Transfers tokens in for lending, Notional will tr
            token.safeTransferFrom(msg.sender, address(this), de

            // Executes a lending action on Notional
            BatchLend[] memory action = EncodeDecode.encodeLendT
                getCurrencyId(),
                getMarketIndex(),
                fCashAmount,
                minImpliedRate,
                useUnderlying
            );
            NotionalV2.batchLend(address(this), action);
        }

        // Mints ERC20 tokens for the receiver, the false flag c
        // operatorAck
        _mint(receiver, fCashAmount, "", "", false);

        _sendTokensToReceiver(token, msg.sender, isETH, balanceB
    }
```

As you can see it calls `NotionalV2` functions with `useUnderlying=True` but according to sponsor clarification `NotionalV2` would fail and revert for those calls because `useUnderlying=True` and `fcash`'s asset token is underlying token (`asset.tokenType == TokenType.NonMintable`). So in summery for `fcash` tokens which asset token is underlying token `NotionalV2` won't handle calls which include `useUnderlying==True` but in `wfCashERC4626` contract functions like `deposit()`, `mint()`, `withdraw()` and `redeem()` they all uses `useUnderlying==True` always so `wfCashERC4626` won't work for those specific type of tokens which asset token is underlying token(`asset.tokenType == TokenType.NonMintable`)

the detail explanations for functions `withdraw()` and `redeem()` are similar.

Tools Used

VIM

Recommended Mitigation Steps

Check that if for that `fcash` token asset token is underlying token or not and set `useUnderlying` based on that.

jeffywu (Notional) confirmed

gzeoneth (judge) decreased severity to Medium and commented:

> There doesn't seems to be loss of fund as `deposit` and `mint` would revert. Judging as Med Risk.

[M-03] The logic of _isUnderlying() in NotionalTradeModule is wrong which will cause mintFCashPosition() and redeemFCashPosition() revert on `fcash` tokens which asset token is underlying token (asset.tokenType == TokenType.NonMintable)

Submitted by unforgiven

For some `fcash` the asset token is underlying token (`asset.tokenType == TokenType.NonMintable`) and `NotionalV2` will not handle minting or burning when it is called with `useUnderlying==True` for those `fcash` s (according to what I asked from sponsor). In summery most of the logics in `NotionalTradeModule` will not work for those `fcash` tokens because `_isUnderlying()` returns `true` result for those tokens which would make `NotionalTradeModule` 's logic for `mintFCashPosition()` and `redeemFCashPosition()` will eventually call `redeemToUnderlying()` and `mintViaUnderlying()` in `wfCashLogic` and those function in `wfCashLogic` will call `NotionalV2` with `useUnderlying==True` and `NotionalV2` will fail and revert for `fcash` tokens which asset token is underlying token, so the whole transaction will fail and `_mintFCashPosition()` and `_redeemFCashPosition()` logic in `NotionalTradeModule` will not work for those `fcash` tokens and manager can't add them to `set` protocol.

Proof of Concept

when for some `fcash` asset token is underlying token, all calls to `NotionalV2` should be with `useUnderlying==False` . but `_isUnderlying()` in `NotionalTradeModule` contract first check that `isUnderlying = _paymentToken == underlyingToken` so for `fcash` tokens where asset token is underlying token it is going to return `isUnderlying==True` . let's assume that for some specific `fcash` asset token is underlying token (`asset.tokenType == TokenType.NonMintable`) and follow the code execution. This is `_isUnderlying()` code in `NotionalTradeModule` :

```
        function _isUnderlying(
            IWrappedfCashComplete _fCashPosition,
            IERC20 _paymentToken
        )
        internal
        view
        returns(bool isUnderlying)
        {
            (IERC20 underlyingToken, IERC20 assetToken) = _getUnderl
            isUnderlying = _paymentToken == underlyingToken;
            if(!isUnderlying) {
                require(_paymentToken == assetToken, "Token is neith
            }
```

```
    }
```

As you can see it calls `_getUnderlyingAndAssetTokens()` and then check `_paymentToken == underlyingToken` to see that if payment token is equal to `underlyingToken`. `_getUnderlyingAndAssetTokens()` uses `getUnderlyingToken()` and `getAssetToken()` in `wfCashBase`. This is `getUnderlyingToken()` code in `wfCashBase`:

```
    /// @notice Returns the token and precision of the token tha
    /// to. For example, fUSDC will return the USDC token addres
    /// address will represent ETH.
    function getUnderlyingToken() public view override returns (
        (Token memory asset, Token memory underlying) = Notional

        if (asset.tokenType == TokenType.NonMintable) {
            // In this case the asset token is the underlying
            return (IERC20(asset.tokenAddress), asset.decimals);
        } else {
            return (IERC20(underlying.tokenAddress), underlying.
        }
    }
```

As you can see for our specific `fcash` token this function will return asset token as underlying token. so for this specific `fcash` token, the asset token and underlying token will be same in `_isUnderlying()` of `NationalTradeModule` but because code first check `isUnderlying = _paymentToken == underlyingToken` so the function will return `isUnderlying=True` as a result for our specific `fcash` token (which asset token is underlying token) This is `_mintFCashPosition()` and `_redeemFCashPosition()` code in `NotionalTradeModule`:

```
    /**
     * @dev Redeem a given fCash position from the specified ser
     * @dev Alo adjust the components / position of the set toke
     */
    function _mintFCashPosition(
        ISetToken _setToken,
        IWrappedfCashComplete _fCashPosition,
        IERC20 _sendToken,
```

```
    uint256 _fCashAmount,
    uint256 _maxSendAmount
)
internal
returns(uint256 sentAmount)
{
    if(_fCashAmount == 0) return 0;

    bool fromUnderlying = _isUnderlying(_fCashPosition, _ser


    _approve(_setToken, _fCashPosition, _sendToken, _maxSenc

    uint256 preTradeSendTokenBalance = _sendToken.balanceOf(
    uint256 preTradeReceiveTokenBalance = _fCashPosition.bal

    _mint(_setToken, _fCashPosition, _maxSendAmount, _fCashA


    (sentAmount,) = _updateSetTokenPositions(
        _setToken,
        address(_sendToken),
        preTradeSendTokenBalance,
        address(_fCashPosition),
        preTradeReceiveTokenBalance
    );

    require(sentAmount <= _maxSendAmount, "Overspent");
    emit FCashMinted(_setToken, _fCashPosition, _sendToken,
}

/**
 * @dev Redeem a given fCash position for the specified rece
 * @dev Alo adjust the components / position of the set toke
 */
function _redeemFCashPosition(
    ISetToken _setToken,
    IWrappedfCashComplete _fCashPosition,
    IERC20 _receiveToken,
    uint256 _fCashAmount,
    uint256 _minReceiveAmount
)
internal
returns(uint256 receivedAmount)
{
    if(_fCashAmount == 0) return 0;
```

```
        bool toUnderlying = _isUnderlying(_fCashPosition, _recei
        uint256 preTradeReceiveTokenBalance = _receiveToken.bala
        uint256 preTradeSendTokenBalance = _fCashPosition.balanc

        _redeem(_setToken, _fCashPosition, _fCashAmount, toUnder


        (, receivedAmount) = _updateSetTokenPositions(
            _setToken,
            address(_fCashPosition),
            preTradeSendTokenBalance,
            address(_receiveToken),
            preTradeReceiveTokenBalance
        );


        require(receivedAmount >= _minReceiveAmount, "Not enough
        emit FCashRedeemed(_setToken, _fCashPosition, _receiveTo

    }
```

As you can see they both uses `_isUnderlying()` to find out that if `_sendToken` is asset token or underlying token. for our specific `fcash` token, the result of `_isUnderlying()` will be `True` and `_mintFCashPosition()` and `_redeemFCashPosition()` will call `_mint()` and `_redeem()` with `toUnderlying` set as `True`. This is `_mint()` and `_redeem()` code:

```
    /**
     * @dev Invokes the wrappedFCash token's mint function from
     */
    function _mint(
        ISetToken _setToken,
        IWrappedfCashComplete _fCashPosition,
        uint256 _maxAssetAmount,
        uint256 _fCashAmount,
        bool _fromUnderlying
    )
    internal
    {
        uint32 minImpliedRate = 0;
```

```
        bytes4 functionSelector =
            _fromUnderlying ? _fCashPosition.mintViaUnderlying.s
        bytes memory mintCallData = abi.encodeWithSelector(
            functionSelector,
            _maxAssetAmount,
            uint88(_fCashAmount),
            address(_setToken),
            minImpliedRate,
            _fromUnderlying
        );
        _setToken.invoke(address(_fCashPosition), 0, mintCallDat
    }

    /**
     * @dev Redeems the given amount of fCash token on behalf of
     */
    function _redeem(
        ISetToken _setToken,
        IWrappedfCashComplete _fCashPosition,
        uint256 _fCashAmount,
        bool _toUnderlying
    )
    internal
    {
        uint32 maxImpliedRate = type(uint32).max;

        bytes4 functionSelector =
            _toUnderlying ? _fCashPosition.redeemToUnderlying.se
        bytes memory redeemCallData = abi.encodeWithSelector(
            functionSelector,
            _fCashAmount,
            address(_setToken),
            maxImpliedRate
        );
        _setToken.invoke(address(_fCashPosition), 0, redeemCallD
    }
```

As you can see they are using `_toUnderlying` value to decide calling between (`mintViaUnderlying()` or `mintViaAsset()`) and (`redeemToUnderlying()` or `redeemToAsset()`), for our specific `fcash` `_toUnderlying` will be `True` so those functions will call `mintViaUnderlying()` and `redeemToUnderlying()` in `wfCashLogic`. `mintViaUnderlying()` and `redeemToUnderlying()` in `wfCashLogic` execution flow eventually would call `NotionalV2` functions with

`useUnderlying=True` for this specific `fcash` token, but `NotionalV2` will revert for that call because for that `fcash` token asset token is underlying token and `NotionalV2` can't handle calls with `useUnderlying==True` for that `fcash` Token. This will cause all the transaction to fail and manager can't call `redeemFCashPosition()` or `mintFCashPosition()` functions for those `fcash` tokens that asset token is underlying token. In summery `NotionalTradeModule` logic will not work for all `fcash` tokens becasue the logic of `_isUnderlying()` is wrong for `fcash` tokens that asset token is underlying token.

Tools Used

VIM

Recommended Mitigation Steps

Change the logic of `_isUnderlying()` in `NotionalTradeModule` so it returns correct results for all `fcash` tokens. One simple solution can be that it first check `payment token` value with `asset token` value.

ckoopmann (Index Coop) confirmed, but disagreed with severity and commented:

> Will need input from @jeffywu here, is the description of the Notional side of things correct ?

> I'll also try to reproduce this issue in a test maybe to make sure if it is valid.

> Not sure if this is "High Risk" as no funds seem to be at risk. However from the description it might render the `NotionalTradeModule` incompatible with certain fCash tokens, which we certainly want to avoid. So will have to review this in more detail.

jeffywu (Notional) commented:

> Agree with @ckoopmann that severity should be reduced here, what this would cause is a revert not any loss of funds.

> The simple fix would just be to always use `mintViaAsset` and `mintViaUnderlying` for the case when fCash has a non-mintable type (currently

> there are no fCash assets of this type and none planned). However, we can also add the ability to query this on the wfCash side to make things more compatible.

ckoopmann (Index Coop) commented:

> @jeffywu What do you think of the suggested mitigation strategy ?

> As far as I understand we would just have to change: `isUnderlying = _paymentToken == underlyingToken;` to `isUnderlying = _paymentToken != assetToken;` to avoid the described issue, no ?

jeffywu (Notional) commented:

> To make this more fool proof what I can do is just add a check on the wrapped fCash side to see if the token is non-mintable and then override the useUnderlying flag internally there. I think that will be a better solution since getUnderlyingToken already has logic to return the asset token when it is marked as non-mintable. That would mean that you would not need to make any changes, I think this will make it easier for integrating developers in the future.

> Specifically add checks here: https://github.com/notional-finance/wrapped-fcash/blob/master/contracts/wfCashLogic.sol#L57-L58 and here: https://github.com/notional-finance/wrapped-fcash/blob/master/contracts/wfCashLogic.sol#L210-L211

> and overwrite the incoming useUnderlying if we are in this situation.

ckoopmann (Index Coop) commented:

> @jeffywu Sounds great to me. If I understand correctly that means I would not have to change anything on the trade module side. Let me know if that is incorrect.

jeffywu (Notional) resolved and commented:

> @ckoopmann, no changes necessary on your side. You can see the changes here.

> https://github.com/notional-finance/wrapped-fcash/pull/11/commits/0ab1ae1080c8eb14fd24d180a01f8ec2c8919022#diff-

> [7c9f6e4700cce75c3c2abb4902f45f7398dcac73135a605b59825b26de7d6af0R245](#)

[gzeoneth (judge) decreased severity to Medium and commented](#):

> There doesn't seems to be loss of fund as it would revert. Judging as Med Risk.

[M-04] `IsWrappedFcash` check is a gas bomb

Submitted by jonah1005

In the `_isWrappedFCash` check, the `notionalTradeModule` check whether the component is a wrappedCash with the following logic.

```
        try IWrappedfCash(_fCashPosition).getDecodedID() returns
            try wrappedfCashFactory.computeAddress(_currencyId,
                return _fCashPosition == _computedAddress;
            } catch {
                return false;
            }
        } catch {
            return false;
        }
```

The above logic is dangerous when `_fCashPosition` do not revert on `getDecodedID` but instead give a wrong format of return value. The contract would try to decode the return value into `returns(uint16 _currencyId, uint40 _maturity)` and revert. The revert would consume what ever gas it's provided.

[CETH](#) is an example. There's a fallback function in `ceth`

```
    function () external payable {
        requireNoError(mintInternal(msg.value), "mint failed");
    }
```

As a result, calling `getDecodedID` would not revert. Instead, calling `getDecodedID` of `CETH` would consume all remaining gas. This creates so many issues. First, users

would waste too much gas on a regular operation. Second, the transaction might fail if `ceth` is not the last position. Third, the wallet contract can not interact with set token with ceth as it consumes all gas.

Proof of Concept

The following contract may fail to redeem setTokens as it consumes too much gas (with 20M gas limit).

Test.sol

```
function test(uint256 _amount) external {
    cToken.approve(address(issueModule), uint256(-1));
    wfCash.approve(address(issueModule), uint256(-1));
    issueModule.issue(setToken, _amount, address(this));
    issueModule.redeem(setToken, _amount, address(this));
}
```

Also, we can check how much gas it consumes with the following function.

```
function TestWrappedFCash(address _fCashPosition) public vie
    if(!_fCashPosition.isContract()) {
        return false;
    }
    try IWrappedfCash(_fCashPosition).getDecodedID() returns
        try wrappedfCashFactory.computeAddress(_currencyId,
            return _fCashPosition == _computedAddress;
        } catch {
            return false;
        }
    } catch {
        return false;
    }
}
```

Test this function with `cdai` and `ceth`, we can observe that there's huge difference of gas consumption here.

```
Gas used:            30376 of 130376
```

```
Gas used:              19479394 of 19788041
```

Tools Used

Hardhat

Recommended Mitigation Steps

I recommend building a map in the notionalTradeModule and inserting the wrappeCash in the `mintFCashPosition` function.

```
function addWrappedCash(uint16 _currencyId, uint40 _maturity) pu
    address computedAddress = wrappedfCashFactory.computeAddress
    wrappedFCash[computedAddress] = true;
}
```

Or we could replace the try-catch pattern with a low-level function call and check the return value's length before decoding it.

Something like this might be a fix.

```
    (bool success, bytes memory returndata) = target.delegateca
    if (!success || returndata.length != DECODED_ID_RETURN_LENGT
        return false;
    }
// abi.decode ....
```

ckoopmann (Index Coop) confirmed and commented:

> Correct, this is an issue that I also recently ran into (after the contest had already started) when doing additional tests. My solution was to just add a fixed gas limit to the `getDecodedID` call which seemed to solve it.

> In an earlier version of the contract I had a manual mapping as suggested here, however this is not ideal since the setToken could obtain fCash positions using other SetModules (such as the general TradeModule) which would then not be registered in this mapping.

> Limiting the gas usage of these calls seems like an easier and more robust mitigation strategy. (might want to make these gas limits updateable though)

gzeoneth (judge) decreased severity to Medium and commented:

> Valid but don't think this is High Risk, `eth_estimateGas` should fail preventing most user from interacting with a ridiculous gas limit.

[M-05] transferfCash does not work as expected

Submitted by csanuragjain

If maturity is reached and user has asked for redeem with opts.transferfCash as true, then if (hasMatured()) turns true at wfCashLogic.sol#L216 causing fcash to be cashed out in underlying token and then sent to receiver. So receiver obtains underlying when fcash was expected. The sender wont get an error thinking fcash transfer was success

Proof of Concept

1. User A calls redeem with opts.transferfCash as true and receiver as User B
2. Since maturity is reached so instead of transferring the fCash, contract would simply cash out fCash and sent the underlying token to the receiver which was not expected

Recommended Mitigation Steps

If opts.transferfCash is true and maturity is reached then throw an error mentioning that fCash can no longer be transferred

jeffywu (Notional) confirmed:

> Sounds reasonable.

[M-06] Users Might Not Be Able To Purchase Or Redeem SetToken

Submitted by xiaoming90, also found by berndartmueller

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L309

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L385

Proof of Concept

Whenever a setToken is issued or redeemed, the `moduleIssueHook` and `moduleRedeemHook` will be triggered. These two hooks will in turn call the `_redeemMaturedPositions` function to ensure that no matured fCash positions remain in the Set by redeeming any matured fCash position.

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L309

```
    /**
     * @dev Hook called once before setToken issuance
     * @dev Ensures that no matured fCash positions are in the set w
     */
    function moduleIssueHook(ISetToken _setToken, uint256 /* _setTok
        _redeemMaturedPositions(_setToken);
    }

    /**
     * @dev Hook called once before setToken redemption
     * @dev Ensures that no matured fCash positions are in the set w
     */
    function moduleRedeemHook(ISetToken _setToken, uint256 /* _setTo
        _redeemMaturedPositions(_setToken);
    }
```

The `_redeemMaturedPositions` will loop through all its fCash positions and attempts to redeem any fCash position that has already matured. However, if one of

the fCash redemptions fails, it will cause the entire function to revert. If this happens, no one could purchase or redeem the setToken because `moduleIssueHook` and `modileRedeemHook` hooks will revert every single time. Thus, the setToken issuance and redemption will stop working entirely and this setToken can be considered "bricked".

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L385

```
/**
 * @dev Redeem all matured fCash positions for the given SetToke
 */
function _redeemMaturedPositions(ISetToken _setToken)
internal
{
    ISetToken.Position[] memory positions = _setToken.getPositic
    uint positionsLength = positions.length;

    bool toUnderlying = redeemToUnderlying[_setToken];

    for(uint256 i = 0; i < positionsLength; i++) {
        // Check that the given position is an equity position
        if(positions[i].unit > 0) {
            address component = positions[i].component;
            if(_isWrappedFCash(component)) {
                IWrappedfCashComplete fCashPosition = IWrappedfC
                if(fCashPosition.hasMatured()) {
                    (IERC20 receiveToken,) = fCashPosition.getTc
                    if(address(receiveToken) == ETH_ADDRESS) {
                        receiveToken = weth;
                    }
                    uint256 fCashBalance = fCashPosition.balance
                    _redeemFCashPosition(_setToken, fCashPositic
                }
            }
        }
    }
}
```

Impact

User will not be able to purchase or redeem the setToken. User's fund will be stuck in the SetToken Contract. Unable to remove matured fCash positions from SetToken and update positions of its asset token.

Recommended Mitigation Steps

This is a problem commonly encountered whenever a method of a smart contract calls another contract — you cannot rely on the other contract to work 100% of the time, and it is dangerous to assume that the external call will always be successful.

It is recommended to:

- Consider alternate method of updating the asset position so that the SetToken's core functions (e.g. issuance and redemption) will not be locked if one of the matured fCash redemptions fails.
- Evaluate if `_redeemMaturedPositions` really need to be called during SetToken's issuance and redemption. If not, consider removing them from the hooks, so that any issue or revert within `_redeemMaturedPositions` won't cause the SetToken's issuance and redemption functions to stop working entirely.
- Consider implementing additional function to give manager/user an option to specify a list of matured fCash positions to redeem instead of forcing them to redeem all matured fCash positions at one go.

ckoopmann (Index Coop) acknowledged, but disagreed with severity and commented:

> The `_isWrappedFCash(component)` should make sure that these calls are only executed on valid wrappedfCash instances deployed from the configured factory.

> The issue does not outline a practical scenario / test case where this issue would actually arise. If it did the manager could probably still remove / redeem the component either via this module, or one of the other Trade modules.

> However I'll have to look into this if I can find a scenario where this would actually arise. I will also consider making this redeem step optional, but I will have to think more about this as this might introduce other more serious issues.

> @jeffywu Do you see any scenario where the redemption of a matured fCash position might fail in this context?

> EDIT: Just noticed that https://github.com/code-423n4/2022-06-notional-coop-findings/issues/226 mentions (among others) the scenario of USDC tokens being blocked which I guess is unlikely but outside of our control. This makes me think we might want to make the redemption of matured tokens during issuance / redemption optional.

ckoopmann (Index Coop) confirmed and commented:

> Changed label from acknowledged to confirmed, since on second thought I think we likely want to adopt some kind of mitigation strategy for this. However still tentative / unclear which strategy we want to adopt.

[M-07] Residual Allowance Might Allow Tokens In SetToken To Be Stolen

Submitted by xiaoming90

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L418

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L493

Proof of Concept

Whenever `_mintFCashPosition` function is called to mint new fCash position, the contract will call the `_approve` function to set the allowance to `_maxSendAmount` so that the fCash Wrapper contact can pull the payment tokens from the SetToken contract during minting.

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L418

```
    function _mintFCashPosition(
        ISetToken _setToken,
        IWrappedfCashComplete _fCashPosition,
        IERC20 _sendToken,
        uint256 _fCashAmount,
        uint256 _maxSendAmount
    )
    internal
    returns(uint256 sentAmount)
    {
        if(_fCashAmount == 0) return 0;
        bool fromUnderlying = _isUnderlying(_fCashPosition, _sendTok

        _approve(_setToken, _fCashPosition, _sendToken, _maxSendAmou

        uint256 preTradeSendTokenBalance = _sendToken.balanceOf(addr
        uint256 preTradeReceiveTokenBalance = _fCashPosition.balance

        _mint(_setToken, _fCashPosition, _maxSendAmount, _fCashAmour

            ..SNIP..
    }
```

Note that `_maxSendAmount` is the maximum amount of payment tokens that is allowed to be consumed during minting. This is not the actual amount of payment tokens consumed during the minting process. Thus, after the minting, there will definitely be some residual allowance since it is unlikely that the fCash wrapper contract will consume the exact maximum amount during minting.

The following piece of code shows that having some residual allowance is expected. The `_approve` function will not set the allowance unless there is insufficient allowance.

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-

notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L493

```
    /**
     * @dev Approve the given wrappedFCash instance to spend the set
     */
    function _approve(
        ISetToken _setToken,
        IWrappedfCashComplete _fCashPosition,
        IERC20 _sendToken,
        uint256 _maxAssetAmount
    )
    internal
    {
        if(IERC20(_sendToken).allowance(address(_setToken), address(
            bytes memory approveCallData = abi.encodeWithSelector(_s
            _setToken.invoke(address(_sendToken), 0, approveCallData
        }
    }
```

Impact

Having residual allowance increases the risk of the asset tokens being stolen from the SetToken contract. SetToken contract is where all the tokens/assets are held. If the Notional's fCash wrapper contract is compromised, it will allow the compromised fCash wrapper contract to withdraw funds from the SetToken contract due to the residual allowance.

Note that Notional's fCash wrapper contract is not totally immutable, as it is a upgradeable contract. This is an additional risk factor to be considered. If the Notional's deployer account is compromised, the attacker could upgrade the Notional's fCash wrapper contract to a malicious one to withdraw funds from the Index Coop's SetToken contract due to the residual allowance.

Index Coop and Notional are two separate protocols and teams. Thus, it is a good security practice not to place any trust on external party wherever possible to ensure that if one party is compromised, it won't affect the other party. Thus, there should not be any residual allowance that allows Notional's contract to withdraw funds from Index Coop's contract in any circumstance.

In the worst case scenario, a “lazy” manager might simply set the `_maxAssetAmount` to `type(uint256).max`. Thus, this will result in large amount of residual allowance left, and expose the SetToken contract to significant risk.

Recommended Mitigation Steps

Approve the allowance on-demand whenever `_mintFCashPosition` is called, and reset the allowance back to zero after each minting process to eliminate any residual allowance.

```
    function _mintFCashPosition(
        ISetToken _setToken,
        IWrappedfCashComplete _fCashPosition,
        IERC20 _sendToken,
        uint256 _fCashAmount,
        uint256 _maxSendAmount
    )
    internal
    returns(uint256 sentAmount)
    {
        if(_fCashAmount == 0) return 0;
        bool fromUnderlying = _isUnderlying(_fCashPosition, _sendTok

        _approve(_setToken, _fCashPosition, _sendToken, _maxSendAmou

        uint256 preTradeSendTokenBalance = _sendToken.balanceOf(addr
        uint256 preTradeReceiveTokenBalance = _fCashPosition.balance

        _mint(_setToken, _fCashPosition, _maxSendAmount, _fCashAmour

            ..SNIP..

    +       // Reset the allowance back to zero after minting
    +       _approve(_setToken, _fCashPosition, _sendToken, 0);
    }
```

Update the `_approve` accordingly to remove the if-statement related to residual allowance.

```
    function _approve(
        ISetToken _setToken,
```

```
        IWrappedfCashComplete _fCashPosition,
        IERC20 _sendToken,
        uint256 _maxAssetAmount
    )
    internal
    {
-       if(IERC20(_sendToken).allowance(address(_setToken), address
            bytes memory approveCallData = abi.encodeWithSelector(_s
            _setToken.invoke(address(_sendToken), 0, approveCallData
-       }
    }
```

ckoopmann (Index Coop) confirmed and commented:

> This looks like a prudent suggestion. Will review and potentially adopt the suggested mitigation.

[M-08] DOS set token through erc777 hook

Submitted by jonah1005

https://github.com/code-423n4/2022-06-notional-coop/blob/main/index-coop-notional-trade-module/contracts/protocol/modules/v1/DebtIssuanceModule.sol#L131-L141

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC777/ERC777.sol#L376-L380

Impact

The `wfCash` is an `erc777` token. ERC777.sol#L376-L380 Users can get the control flow before sending token and after receiving tokens. This creates attack vectors that require extra caution in designing modules. Any combination of modules may lead to a possible exploit. To elaborate on the dangerousness of the re-entrancy attack, a possible scenario is presented.

Before the exploit, we first elaborate on three attack vectors:

1. DebtIssuanceModule.sol#L131-L141 The issuance module would pull tokens from the sender before minting setToken.

Assume there are three compoenents in this set. 1. CDai. 2. wfCash In the `_callTokensToSend`, the setToken has received `cdai` and the `totalSupply` is still the same.

2. `nonReentrant` does not protect cross-contract re-entrancy. This means, that during the `issue` of issuance module, users can trigger other modules' functions.
3. Restricted functions with `onlyManagerAndValidSet` modifier may be triggered by the exploiter as well. Manager of a setToken is usually a manager contract. Assume it's a multisig-wallet, the exploiter can front-run the execute transaction and replay the payload during his exploit. Note, a private transaction from flash-bot can still be front-run. Please refer to the uncle bandit risk

Given the above attack vectors, the exploiter have enough weapons to exploit the `setToken` at a propriate time. Note that different combination of modules may have different exploit paths. As long as the above attack vectors remain, the setToken is vulnerable.

Assume a setToken with `CompoundLeverageModule`, `NotionalTradeModule` and `BasicIssuanceModule` with the following positions: 1. CDAI: 100 2. wfCash-DAI 100 and totalSupply = 100. The community decides to remove the `compoundLeverageModule` from the set token. Since `notionalTradeModule` can handle cDAI, the community vote to just call `removeModule` to remove `compoundLeverageModule`. The exploiter has the time to build an exploit and wait the right timing to come.

0. The exploiter listen the manager multisig wallet.
1. Exploiter issue 10 setToken.
2. During the `_callTokensToSend` of `wfcash`, the totalSupply = 100, CDAI = 110, wfCash-DAI = 110.
3. Call `sync` of `CompoundLeverageModule`. `_getCollateralPosition` get `_cToken.balanceOf(address(_setToken)) = 110` and `totalSupply = 100` and update the `DefaultUnit` of `CETH` 1.1X.

4. Replay multisig wallet's payload and remove `compoundLeverageModule`.
5. The `setToken` can no longer issue / redeem as it would raise `undercollateralized` error. Further, `setValuer` would give a pumped valuation that may cause harm to other protocols.

Proof of Concept

POC The exploit is quite lengthy. Please check the `Attack.sol` for the main exploit logic.

```
function register() public {
    _ERC1820_REGISTRY.setInterfaceImplementer(address(this),
    _ERC1820_REGISTRY.setInterfaceImplementer(address(this),
}

function attack(uint256 _amount) external {
    cToken.approve(address(issueModule), uint256(-1));
    wfCash.approve(address(issueModule), uint256(-1));
    issueModule.issue(setToken, _amount, address(this));
}

function tokensToSend(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes calldata userData,
    bytes calldata operatorData
) external {
    compoundModule.sync(setToken, false);
    manager.removeModule(address(setToken));
}
```

Recommended Mitigation Steps

The design choice of wfcash being an `ERC777` seems unnecessary to me. Over the past two years, ERC777 leads to so many exploits. IMBTC-UNISWAP CREAM-AMP I recommend the team use ERC20 instead.

If the SetToken team considers supporting ERC777 necessary, I recommend implementing protocol-wide cross-contract reentrancy prevention. Please refer to

Rari-Capital. Comptroller.sol#L1978-L2002

Note that, `Rari` was exploited given this reentrancy prevention. Simply making `nonReentrant` cross-contact prevention may not be enough. I recommend to setToken protocol going through every module and re-consider whether it's re-entrancy safe.

ckoopmann (Index Coop) commented:

> @jeffywu : What do you think ? Can we drop the ERC777 interface from wfCash (it's not used by the NotionalTradeModule afaik). If not, I'll have to review this issue in more details and see if we need a mitigation on our side.

> Note that the issue mostly references the DebtIssuanceModule which we probably wont / cant change unless there is a major vulnerability.

jeffywu (Notional) confirmed and commented:

> @ckoopmann I think we can just drop ERC777

[M-09] Silent overflow of `_fCashAmount`

Submitted by GreyArt, also found by Meera

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L526

Description

If a `_fCashAmount` value that is greater than uint88 is passed into the `_mint` function, downcasting it to uint88 will silently overflow.

Recommended Mitigation Steps

```
// Use a safe downcast function e.g. wfCashLogic::_safeUint88
```

```
function _safeUint88(uint256 x) internal pure returns (uint88) {
    require(x <= uint256(type(uint88).max));
    return uint88(x);
}
```

ckoopmann (Index Coop) confirmed and commented:

> This seems reasonable and we'll probably adopt the suggested mitigation approach.

[M-10] User can alter amount returned by redeem function due to control transfer

Submitted by 0xDjango

Control is transferred to the receiver when receiving the ERC777. They are able to transfer the ERC777 to another account, at which time the before and after balance calculation will be incorrect.

```
        uint256 balanceBefore = IERC20(asset()).balanceOf(recei

        if (msg.sender != owner) {
            _spendAllowance(owner, msg.sender, shares);
        }
        _redeemInternal(shares, receiver, owner);
//////////////
Control is transferred to user. They can alter their balance her
////////////

        uint256 balanceAfter = IERC20(asset()).balanceOf(receive
        uint256 assets = balanceAfter - balanceBefore;

//////////
Assets can be as low as 0 if they have transferred the same amou
//////////

        emit Withdraw(msg.sender, receiver, owner, assets, share
        return assets;
```

jeffywu (Notional) disputed and commented:

> Control of the contract is not transferred to anyone during a `balanceOf` call which is read only. No state can be modified.

fatherGoose1 (warden) commented:

> The control is transferred in `_redeemInternal()` which calls `_burn()` on the ERC777 which transfers control.

> `_burn()` function logic here: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/109778c17c7020618ea4e035efb9f0f9b82d43ca/contracts/token/ERC777/ERC777.sol#L390-L400

jeffywu (Notional) confirmed and commented:

> Understood, will change to confirmed.

Low Risk and Non-Critical Issues

For this contest, 61 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **berndartmueller** received the top score from the judge.

The following wardens also submitted reports: lllllll, GimelSec, sorrynotsorry, xiaoming90, GreyArt, Meera, jonah1005, joestakey, unforgiven, Funen, 0xf15ers, Ruhum, Picodes, hyh, fatherOfBlocks, Deivitto, zzzitron, Trumpero, TomJ, TerrierLover, sseefried, simon135, PierrickGT, oyc_109, minhquanym, ellahi, csanuragjain, cloudjunky, Chom, cccz, Bronicle, antonttc, 0xDjango, 0x29A, 0x1f8b, Sm4rty, saian, sach1r0, Nethermind, kenzo, hake, dipp, delfin454000, cryptphi, Cityscape, catchup, c3phas, 0xNineDec, 0xNazgul, 0xmint, _Adam, z3s, Waze, Tadashi, slywaters, Lambda, JC, hansfriese, ayeslick, *and* 0xkatana.

Codebase Impressions & Summary

Overall, the contracts are very well implemented and the code quality is very high. Protocol developers provided adequate documentation and information on the

protocol.

Running the test suite is extensive and covers most of the code, however, a few things stood out:

- `test_wrapped_fcash.test_transfer_fcash_contract` is skipped
- many tests are failing on the forked mainnet due to using addresses with insufficient token balances (e.g. `whales.DAI_EOA` has insufficient DAI tokens)
- the `notionalTradeModule.spec` test suite has many open `TODOs`

Table of Contents

[L-01] Zero-address checks are missing

Zero-address checks are a best practice for input validation of critical address parameters. While the codebase applies this to most cases, there are many places where this is missing in constructors and setters.

Impact: Accidental use of zero-addresses may result in exceptions, burn fees/tokens, or force redeployment of contracts.

Findings

NotionalTradeModule.sol

L140 - `wrappedfCashFactory = _wrappedfCashFactory;`
L141 - `weth = _weth;`

wfCashBase.sol

L30 - `NotionalV2 = _notional;`
L31 - `WETH = _weth;`

WrappedfCashFactory.sol

L18 - `BEACON = _beacon;`

Recommended Mitigation Steps

Add zero-address checks, e.g.:

```
    require(_weth != address(0), "Zero-address");
```

[L-02] Use of floating pragma

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

https://swcregistry.io/docs/SWC-103

Findings

wfCashERC4626.sol `pragma solidity ^0.8.0;`

Recommended Mitigation Steps

Lock the pragma version to the same version as used in the other contracts and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile it locally.

[L-03] Events not emitted for important state changes

When changing state variables events are not emitted. Emitting events allows monitoring activities with off-chain monitoring tools.

Findings

NotionalTradeModule.sol#L301 `function setRedeemToUnderlying(ISetToken _setToken, bool _toUnderlying)`

Recommended Mitigation Steps

Emit events for state variable changes.

[L-04] Matured fCash positions not automatically redeemed in `NotionalTradeModule.initialize`

The function comments imply that matured fCash positions are redeemed within `NotionalTradeModule.initialize`. However, this redemption is not implemented in this function.

Findings

NotionalTradeModule.sol#L216

```
* Redeem all fCash positions that have reached maturity for thei
```

Recommended Mitigation Steps

Consider calling the function `_redeemMaturedPositions` to redeem matured fCash positions or adapt the function comments.

[L-05] Misleading `NotionalTradeModule._mintFCashPosition` function comments

The function comments imply that the given fCash position is redeemed. However, this function implements **minting** fCash tokens.

Findings

NotionalTradeModule.sol#L415

```
* @dev Redeem a given fCash position from the specified send tok
```

Recommended Mitigation Steps

Fix the comments to mention minting instead of redeeming.

[L-06] Misleading comment in `wfCashLogic._burn` function

The comment next to the `_withdrawCashToAccount` function call implies that the `from` address is the withdrawal receiver. However, `opts.receiver` is the receiver.

Findings

wfCashLogic.sol#L230

```
// Transfer withdrawn tokens to the `from` address // @audit-ir
_withdrawCashToAccount(
    currencyId,
    opts.receiver,
```

```
            _safeUint88(assetInternalCashClaim),
            opts.redeemToUnderlying
        );
```

Recommended Mitigation Steps

Fix the comment.

[L-07] Matured fCash can still be wrapped via `ERC1155 transfer`

Contrary to the key invariants stated in the **README** (*"After maturity, wrapped fCash can no longer be minted."*), matured fCash can be sent to this `wfCash` contract to receive wrapped fCash tokens in return.

Findings

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L107-L152

Recommended Mitigation Steps

Consider adding a check for fCash maturity:

```
    require(!hasMatured(), "fCash matured");
```

[L-08] Contracts are using outdated OpenZeppelin version `^3.4.2-solc-0.7`

Within `notional-wrapped-fcash package.json`, an outdated OZ version is used (which has known vulnerabilities, see https://snyk.io/vuln/npm%3A%40openzeppelin%2Fcontracts).

However, as `Brownie` is used to install dependencies and compile the contracts, using this outdated version declared in the `package.json` does not impose any risks so far.

Anyway, to prevent any issues in the future (e.g. using solely `hardhat` to compile and deploy the contracts), upgrade the used OZ packages within the `package.json` to the latest versions.

See similar findings:

- https://github.com/code-423n4/2022-02-hubble-findings/issues/81
- https://github.com/code-423n4/2022-02-hubble-findings/issues/81

Findings

https://github.com/code-423n4/2022-06-notional-coop/blob/main/notional-wrapped-fcash/package.json#L14

Recommended Mitigation Steps

Consider using the latest OZ packages within `package.json`.

[L-09] `wfCashERC4626` contract does not conform to `EIP4626`

The `wfCashERC4626` contract implements the `EIP4626` standard (EIP-4626: Tokenized Vault Standard).

However, according to `EIP4626`, the below-mentioned functions do not fully adhere to the specs. They possibly revert due to `require` checks or revert due to external calls reverting.

Findings

L47 - `function totalAssets() public view override returns (uint256)`

Possibly reverts due to `_getMaturedValue` and `_getPresentValue` reverting.

L52 - `function convertToShares(uint256 assets) public view override returns (uint256 shares)`

Possibly reverts due to `_getPresentValue` and `totalAssets` reverting.

L64 - `function convertToAssets(uint256 shares) public view override returns (uint256 assets)`

Possibly reverts due to `_getPresentValue` and `totalAssets` reverting.

L85 - `function maxWithdraw(address owner) public view override returns (uint256)`

Possibly reverts due to `convertToShares` within `previewWithdraw` reverting.

Recommended Mitigation Steps

Given the circumstances, in most of the mentioned cases, it's not possible to implement it without ever reverting. Nevertheless, I want to point out that this contract does not fully conform with the `EIP4626` standard.

[N-01] Use the `isETH` return value from `wfCashBase.getToken` instead of checking equality with `ETH_ADDRESS`

The `wfCashBase.getToken` returns `bool isETH` which can be used to figure out if the returned token is the native ETH token.

Findings

NotionalTradeModule.sol#L400-L403

```
    (IERC20 receiveToken,) = fCashPosition.getToken(toUnderlying);
    if(address(receiveToken) == ETH_ADDRESS) { // @audit-info use re
        receiveToken = weth;
    }
```

Recommended Mitigation Steps

Consider using the returned `isETH` variable:

```
    (IERC20 receiveToken, bool isETH) = fCashPosition.getToken(toUnc
```

```
    if(isETH) {
        receiveToken = weth;
    }
```

ckoopmann (Index Coop) commented:

> Very nice QA report imo. All the issues regarding the NotionalTradeModule are valid and actionable. Also good format and concise description of the issues and proposed mitigation / fix.

Gas Optimizations

For this contest, 55 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: antonttc, Meera, 0xKitsune, joestakey, 0xkatana, 0x29A, 0x1f8b, Waze, PierrickGT, hyh, GreyArt, delfin454000, Chom, berndartmueller, TomJ, Tomio, TerrierLover, slywaters, simon135, rfa, 8olidity, 0xf15ers, _Adam, 0xSolus, UnusualTurtle, saian, sach1r0, Picodes, oyc_109, ellahi, Deivitto, catchup, c3phas, asutorufos, Sm4rty, samruna, kaden, Funen, fatherOfBlocks, ElKu, DavidGialdi, csanuragjain, Cityscape, 0xNazgul, 0v3rf10w, ynnad, Tadashi, minhquanym, Lambda, hansfriese, hake, Fitraldys, djxploit, *and* 0xmint.

	Issue	Instances
1	Avoid contract existence checks by using solidity version 0.8.10 or later	6
2	Multiple accesses of a mapping/array should use a local variable cache	5
3	`internal` functions only called once can be inlined to save gas	7
4	`<array>.length` should not be looked up in every loop of a `for` -loop	2
5	`require()` / `revert()` strings longer than 32 bytes cost extra gas	4
6	`private` functions not called by the contract should be removed to save deployment gas	1
7	Optimize names to save gas	1
8	Using `bool` s for storage incurs overhead	3

	Issue	Instances
9	Use a more recent version of solidity	1
10	Use a more recent version of solidity	1
11	It costs more gas to initialize non- `constant` /non- `immutable` variables to zero than to let the default of zero be applied	7
12	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	5
13	Splitting `require()` statements that use `&&` saves gas	2
14	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	53
15	`abi.encode()` is less efficient than `abi.encodePacked()`	3
16	Using `private` rather than `public` for constants, saves gas	1
17	Use custom errors rather than `revert()` / `require()` strings to save gas	17
18	Functions guaranteed to revert when called by normal users can be marked `payable`	14

Total: 133 instances over 18 issues

[1] Avoid contract existence checks by using solidity version 0.8.10 or later

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**700 gas**), to check for contract existence for external calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value

There are 6 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

/// @audit registerToIssuanceModule()
239:                 try IDebtIssuanceModule(modules[i]).registerTo

/// @audit unregisterFromIssuanceModule()
```

```
256:                 try IDebtIssuanceModule(modules[i]).unregi

/// @audit allowance()
501:             if(IERC20(_sendToken).allowance(address(_setToken)

/// @audit getDecodedID()
639:             try IWrappedfCash(_fCashPosition).getDecodedID() r
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L239

```
File: notional-wrapped-fcash/contracts/wfCashERC4626.sol

/// @audit balanceOf()
212:             uint256 balanceBefore = IERC20(asset()).balanceOf(

/// @audit balanceOf()
219:             uint256 balanceAfter = IERC20(asset()).balanceOf(r
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L212

[2] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` variable when the value is accessed multiple times, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory

There are 5 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module
```

```
/// @audit positions[i] on line 395
396:                 address component = positions[i].component

/// @audit positions[i] on line 607
608:                 address component = positions[i].component

/// @audit positions[i] on line 619
620:                 address component = positions[i].component
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L396

```
File: notional-wrapped-fcash/contracts/wfCashLogic.sol

/// @audit assets[0] on line 133
134:                 assets[0].maturity,

/// @audit assets[0] on line 134
135:                 assets[0].assetType
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L134

[3] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 7 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

368:     function _deployWrappedfCash(uint16 _currencyId, uint4
```

```
376:      function _getWrappedfCash(uint16 _currencyId, uint40 _

418       function _mintFCashPosition(
419           ISetToken _setToken,
420           IWrappedfCashComplete _fCashPosition,
421           IERC20 _sendToken,
422           uint256 _fCashAmount,
423           uint256 _maxSendAmount
424       )
425       internal
426:      returns(uint256 sentAmount)

493       function _approve(
494           ISetToken _setToken,
495           IWrappedfCashComplete _fCashPosition,
496           IERC20 _sendToken,
497:          uint256 _maxAssetAmount

537       function _redeem(
538           ISetToken _setToken,
539           IWrappedfCashComplete _fCashPosition,
540           uint256 _fCashAmount,
541:          bool _toUnderlying

581       function _getUnderlyingAndAssetTokens(IWrappedfCashCom
582       internal
583       view
584:      returns(IERC20 underlyingToken, IERC20 assetToken)

596       function _getFCashPositions(ISetToken _setToken)
597       internal
598       view
599:      returns(address[] memory fCashPositions)
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L368

[4] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 2 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

238:          for(uint256 i = 0; i < modules.length; i++) {
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L238

```
File: index-coop-notional-trade-module/contracts/protocol/module

254:          for(uint256 i = 0; i < modules.length; i++) {
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L254

[5] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra chunk of byetes past the original 32 i[incurs an MSTORE](#) which costs 3 gas

There are 4 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

169:           require(_setToken.isComponent(address(_sendToken))
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L169

```
File: index-coop-notional-trade-module/contracts/protocol/module

199:           require(_setToken.isComponent(address(wrappedfCash
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L199

```
File: index-coop-notional-trade-module/contracts/protocol/module

378:           require(wrappedfCashAddress.isContract(), "Wrappec
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L378

```
File: index-coop-notional-trade-module/contracts/protocol/module

573:               require(_paymentToken == assetToken, "Token is
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L573

[6] `private` functions not called by the contract should be removed to save deployment gas

There is 1 instance of this issue:

```
File: notional-wrapped-fcash/contracts/wfCashERC4626.sol   #1

243:     function _safeNegInt88(uint256 x) private pure returns
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L243

[7] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted

There is 1 instance of this issue:

```
File: notional-wrapped-fcash/contracts/wfCashBase.sol   #1

/// @audit getToken()
16:   abstract contract wfCashBase is ERC777Upgradeable, IWrappe
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashBase.sol#L16

[8] Using `bool`s for storage incurs overhead

```
        // Booleans are more expensive than uint256 or any type that
        // word because each write operation emits an extra SLOAD to
        // slot's contents, replace the bits taken up by the boolean
        // back. This is the compiler's defense against contract upg
        // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27 Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from ‘false’ to ‘true’, after having been ‘true’ in the past

There are 3 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

112:       mapping(ISetToken => bool) public redeemToUnderlying;
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L112

```
File: index-coop-notional-trade-module/contracts/protocol/module

115:       mapping(ISetToken => bool) public allowedSetTokens;
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L115

```
File: index-coop-notional-trade-module/contracts/protocol/module

118:       bool public anySetAllowed;
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L118

[9] Use a more recent version of solidity

Use a solidity version of at least 0.8.0 to get overflow protection without `SafeMath` Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There is 1 instance of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

19:   pragma solidity 0.6.10;
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L19

[10] Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There is 1 instance of this issue:

```
File: notional-wrapped-fcash/contracts/wfCashERC4626.sol   #1

2:    pragma solidity ^0.8.0;
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L2

[11] It costs more gas to initialize non-`constant`/non-`immutable` variables to zero than to let the default of zero be applied

There are 7 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

48:       address internal constant ETH_ADDRESS = address(0);

238:          for(uint256 i = 0; i < modules.length; i++) {

254:          for(uint256 i = 0; i < modules.length; i++) {

393:          for(uint256 i = 0; i < positionsLength; i++) {

519:          uint32 minImpliedRate = 0;

605:          for(uint256 i = 0; i < positionsLength; i++) {

618:          for(uint256 i = 0; i < positionsLength; i++) {
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L48

[12] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves **6 gas per loop**

There are 5 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

238:          for(uint256 i = 0; i < modules.length; i++) {

254:          for(uint256 i = 0; i < modules.length; i++) {

393:          for(uint256 i = 0; i < positionsLength; i++) {

605:          for(uint256 i = 0; i < positionsLength; i++) {

618:          for(uint256 i = 0; i < positionsLength; i++) {
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L238

[13] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper

There are 2 instances of this issue:

```
File: notional-wrapped-fcash/contracts/wfCashLogic.sol   #1

116           require(
117               msg.sender == address(NotionalV2) &&
118               // Only accept the fcash id that corresponds t
119               _id == fCashID &&
120               // Protect against signed value underflows
121               int256(_value) > 0,
122               "Invalid"
123:          );
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L116-L123

```
File: notional-wrapped-fcash/contracts/wfCashLogic.sol   #2

129           require(
130               ac.hasDebt == 0x00 &&
131               assets.length == 1 &&
132               EncodeDecode.encodeERC1155Id(
133                   assets[0].currencyId,
134                   assets[0].maturity,
135                   assets[0].assetType
136               ) == fCashID
137:          );
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L129-L137

[14] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 53 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

158:          uint16 _currencyId,

159:          uint40 _maturity,
```

```
187:          uint16 _currencyId,
188:          uint40 _maturity,
368:      function _deployWrappedfCash(uint16 _currencyId, uint4
368:      function _deployWrappedfCash(uint16 _currencyId, uint4
376:      function _getWrappedfCash(uint16 _currencyId, uint40 _
376:      function _getWrappedfCash(uint16 _currencyId, uint40 _
519:          uint32 minImpliedRate = 0;
545:          uint32 maxImpliedRate = type(uint32).max;
639:          try IWrappedfCash(_fCashPosition).getDecodedID() r
639:          try IWrappedfCash(_fCashPosition).getDecodedID() r
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L158

```
File: notional-wrapped-fcash/contracts/wfCashBase.sol
35:       function initialize(uint16 currencyId, uint40 maturity
35:       function initialize(uint16 currencyId, uint40 maturity
83:       function getMaturity() public view override returns (u
93:       function getCurrencyId() public view override returns
98:       function getDecodedID() public view override returns (
98:       function getDecodedID() public view override returns (
103:      function decimals() public pure override returns (uint
109:      function getMarketIndex() public view override returns
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashBase.sol#L35

```
File: notional-wrapped-fcash/contracts/proxy/WrappedfCashFactory

15:       event WrapperDeployed(uint16 currencyId, uint40 maturi

15:       event WrapperDeployed(uint16 currencyId, uint40 maturi

21:       function _getByteCode(uint16 currencyId, uint40 maturi

21:       function _getByteCode(uint16 currencyId, uint40 maturi

26:       function deployWrapper(uint16 currencyId, uint40 matur

26:       function deployWrapper(uint16 currencyId, uint40 matur

39:       function computeAddress(uint16 currencyId, uint40 matu

39:       function computeAddress(uint16 currencyId, uint40 matu
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/proxy/WrappedfCashFactory.sol#L15

```
File: notional-wrapped-fcash/contracts/wfCashERC4626.sol

18:           uint16 currencyId = getCurrencyId();

31:           (uint16 currencyId, uint40 maturity) = getDecodedI

31:           (uint16 currencyId, uint40 maturity) = getDecodedI

100:              (uint16 currencyId, uint40 maturity) = getDecc

100:              (uint16 currencyId, uint40 maturity) = getDecc

120:              (uint16 currencyId, uint40 maturity) = getDecc

120:              (uint16 currencyId, uint40 maturity) = getDecc
```

```
139:             (uint16 currencyId, uint40 maturity) = getDecc
139:             (uint16 currencyId, uint40 maturity) = getDecc
157:             (uint16 currencyId, uint40 maturity) = getDecc
157:             (uint16 currencyId, uint40 maturity) = getDecc
243:     function _safeNegInt88(uint256 x) private pure returns
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L18

```
File: notional-wrapped-fcash/contracts/wfCashLogic.sol
29:           uint88 fCashAmount,
31:           uint32 minImpliedRate
43:           uint88 fCashAmount,
45:           uint32 minImpliedRate
52:           uint88 fCashAmount,
54:           uint32 minImpliedRate,
169:          uint32 maxImpliedRate
187:          uint32 maxImpliedRate
222:              uint16 currencyId = getCurrencyId();
261:          uint16 currencyId,
263:          uint88 assetInternalCashClaim,
279:          uint32 maxImpliedRate
315:      function _safeUint88(uint256 x) internal pure returns
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L29

[15] `abi.encode()` is less efficient than `abi.encodePacked()`

There are 3 instances of this issue:

```
File: notional-wrapped-fcash/contracts/proxy/WrappedfCashFactory

23:             return abi.encodePacked(type(nBeaconProxy).creatic
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/proxy/WrappedfCashFactory.sol#L23

```
File: notional-wrapped-fcash/contracts/wfCashERC4626.sol   #2

230             bytes memory userData = abi.encode(
231                 RedeemOpts({
232                     redeemToUnderlying: true,
233                     transferfCash: false,
234                     receiver: receiver,
235                     maxImpliedRate: 0
236                 })
237:            );
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L230-L237

```
File: notional-wrapped-fcash/contracts/wfCashLogic.sol   #3

159:            bytes memory data = abi.encode(opts);
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L159

[16] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

There is 1 instance of this issue:

```
File: notional-wrapped-fcash/contracts/proxy/WrappedfCashFactory

12:       bytes32 public constant SALT = 0;
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/proxy/WrappedfCashFactory.sol#L12

[17] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hitby avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

There are 17 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

169:           require(_setToken.isComponent(address(_sendToken))

199:           require(_setToken.isComponent(address(wrappedfCash

227:               require(allowedSetTokens[_setToken], "Not allo
```

```
234:          require(_setToken.isInitializedModule(getAndValida

269:          require(_setToken.isInitializedModule(address(_deb

280:          require(controller.isSet(address(_setToken)) || al

378:          require(wrappedfCashAddress.isContract(), "Wrappec

449:          require(sentAmount <= _maxSendAmount, "Overspent")

485:          require(receivedAmount >= _minReceiveAmount, "Not

573:              require(_paymentToken == assetToken, "Token is
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L169

```
File: notional-wrapped-fcash/contracts/wfCashBase.sol

37:           require(cashGroup.maxMarketIndex > 0, "Invalid cur

40            require(
41                DateTime.isValidMaturity(cashGroup.maxMarketIr
42                "Invalid maturity"
43:           );
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashBase.sol#L37

```
File: notional-wrapped-fcash/contracts/wfCashERC4626.sol

23:           require(underlyingExternal > 0, "Must Settle");
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashERC4626.sol#L23

```
File: notional-wrapped-fcash/contracts/wfCashLogic.sol

57:           require(!hasMatured(), "fCash matured");

116           require(
117               msg.sender == address(NotionalV2) &&
118               // Only accept the fcash id that corresponds t
119               _id == fCashID &&
120               // Protect against signed value underflows
121               int256(_value) > 0,
122               "Invalid"
123:          );

211:          require(opts.receiver != address(0), "Receiver is

225:              require(0 < cashBalance, "Negative Cash Balanc
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/notional-wrapped-fcash/contracts/wfCashLogic.sol#L57

[18] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 14 instances of this issue:

```
File: index-coop-notional-trade-module/contracts/protocol/module

156       function mintFCashPosition(
157           ISetToken _setToken,
158           uint16 _currencyId,
```

```
159             uint40 _maturity,
160             uint256 _mintAmount,
161             address _sendToken,
162             uint256 _maxSendAmount
163         )
164             external
165             nonReentrant
166             onlyManagerAndValidSet(_setToken)
167:            returns(uint256)

185         function redeemFCashPosition(
186             ISetToken _setToken,
187             uint16 _currencyId,
188             uint40 _maturity,
189             uint256 _redeemAmount,
190             address _receiveToken,
191             uint256 _minReceiveAmount
192         )
193             external
194             nonReentrant
195             onlyManagerAndValidSet(_setToken)
196:            returns(uint256)

210:        function redeemMaturedPositions(ISetToken _setToken) p

219         function initialize(
220             ISetToken _setToken
221         )
222             external
223             onlySetManager(_setToken, msg.sender)
224:            onlyValidAndPendingSet(_setToken)

219         function initialize(
220             ISetToken _setToken
221         )
222             external
223             onlySetManager(_setToken, msg.sender)
224:            onlyValidAndPendingSet(_setToken)

246:        function removeModule() external override onlyValidAnc

268:        function registerToModule(ISetToken _setToken, IDebtIs

279:        function updateAllowedSetToken(ISetToken _setToken, bo

289:        function updateAnySetAllowed(bool _anySetAllowed) exte
```

```
294         function setRedeemToUnderlying(
295             ISetToken _setToken,
296             bool _toUnderlying
297         )
298         external
299:        onlyManagerAndValidSet(_setToken)

309:        function moduleIssueHook(ISetToken _setToken, uint256

317:        function moduleRedeemHook(ISetToken _setToken, uint256

326         function componentIssueHook(
327             ISetToken _setToken,
328             uint256 _setTokenAmount,
329             IERC20 _component,
330             bool _isEquity
331:        ) external override onlyModule(_setToken) {

338         function componentRedeemHook(
339             ISetToken _setToken,
340             uint256 _setTokenAmount,
341             IERC20 _component,
342             bool _isEquity
343:        ) external override onlyModule(_setToken) {
```

https://github.com/code-423n4/2022-06-notional-coop/blob/6f8c325f604e2576e2fe257b6b57892ca181509a/index-coop-notional-trade-module/contracts/protocol/modules/v1/NotionalTradeModule.sol#L156-L167

jeffywu (Notional) commented:

> Well organized report

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-

risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top