Notional contest
Findings & Analysis Report

2022-03-10

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 code 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 code contest outlined in this document, C4 conducted an analysis of the Notional smart contract system written in Solidity. The code contest took place between January 27—February 2 2022.

Wardens

27 Wardens contributed reports to the Notional contest:

1. leastwood
2. ShippooorDAO
3. cmichel
4. TomFrenchBlockchain

5. gzeon
6. Dravee
7. gellej
8. Jujic
9. GeekyLumberjack
10. hyh
11. defsec
12. WatchPug (jtp and ming)
13. UncleGrandpa925
14. samruna
15. sirhashalot
16. Tomio
17. SolidityScan (cyberboy and zombie)
18. 0x1f8b
19. throttle
20. robee
21. IllIllI
22. PranavG
23. 0v3rf10w
24. cccz
25. camden

This contest was judged by pauliax. The judge also competed in the contest as a warden, but forfeited their winnings.

Final report assembled by liveactionllama and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities and 57 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity, 7 received a risk rating in the category of MEDIUM severity, and 8 received a risk rating in the category of LOW severity.

C4 analysis also identified 16 non-critical recommendations and 23 gas optimizations.

Scope

The code under review can be found within the C4 Notional contest repository, and is composed of 18 smart contracts written in the Solidity programming language and includes 1745 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.

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 (3)

[H-01] Treasury cannot claim COMP tokens & COMP tokens are stuck

Submitted by cmichel, also found by leastwood

The `TreasuryAction.claimCOMPAndTransfer` function uses pre- and post-balances of the `COMP` token to check which ones to transfer:

```
function claimCOMPAndTransfer(address[] calldata cTokens)
    external
    override
    onlyManagerContract
    nonReentrant
    returns (uint256)
{
    // Take a snasphot of the COMP balance before we claim
    // something we shouldn't.
    uint256 balanceBefore = COMP.balanceOf(address(this));

    COMPTROLLER.claimComp(address(this), cTokens);
    // NOTE: If Notional ever lists COMP as a collateral as
    // will never hold COMP balances directly. In this case
    // off of the contract.
    uint256 balanceAfter = COMP.balanceOf(address(this));
    uint256 amountClaimed = balanceAfter.sub(balanceBefore)
    // NOTE: the onlyManagerContract modifier prevents a tr
    COMP.safeTransfer(treasuryManagerContract, amountClaime
    // NOTE: TreasuryManager contract will emit a COMPHarve
    return amountClaimed;
}
```

Note that anyone can claim COMP tokens on behalf of any address (see `Comptroller.claimComp`). An attacker can claim COMP tokens on behalf of the contract and it'll never be able to claim any compound itself. The COMP claimed by the attacker are stuck in the contract and cannot be retrieved. (One can eventually get back the stuck COMP by creating a cCOMP market and then transferring it through `transferReserveToTreasury`.)

Recommended Mitigation Steps

Don't use pre-and post-balances, can you use the entire balance?

jeffywu (Notional) disagreed with severity and commented:

> Dispute as a high risk bug. Would categorize this as medium risk.

> There is no profit to be gained by doing this from the attacker besides denial of service. The protocol could simply upgrade to regain access to the tokens. We will fix this regardless.

pauliax (judge) commented:

> Very good find.

> It is a tough decision if this should be classified as High or Medium severity. An exploiter cannot acquire those assets, and the contracts are upgradeable if necessary, however, I think this time I will leave it in favor of wardens who both are experienced enough and submitted this as of high severity: *3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).*

[H-02] Cooldown and redeem windows can be rendered useless

Submitted by ShippooorDAO

Cooldown and redeem windows can be rendered useless.

Proof of Concept

- Given an account that has not staked sNOTE.
- Account calls sNOTE.startCooldown
- Account waits for the duration of the cooldown period. Redeem period starts.
- Account can then deposit and redeem as they wish, making the cooldown useless.
- Multiple accounts could be used to "hop" between redeem windows by transfering between them, making the redeem window effictively useless.

Could be used for voting power attacks using flash loan if voting process is not monitored https://www.coindesk.com/tech/2020/10/29/flash-loans-have-made-their-way-to-manipulating-protocol-elections/

Tools Used

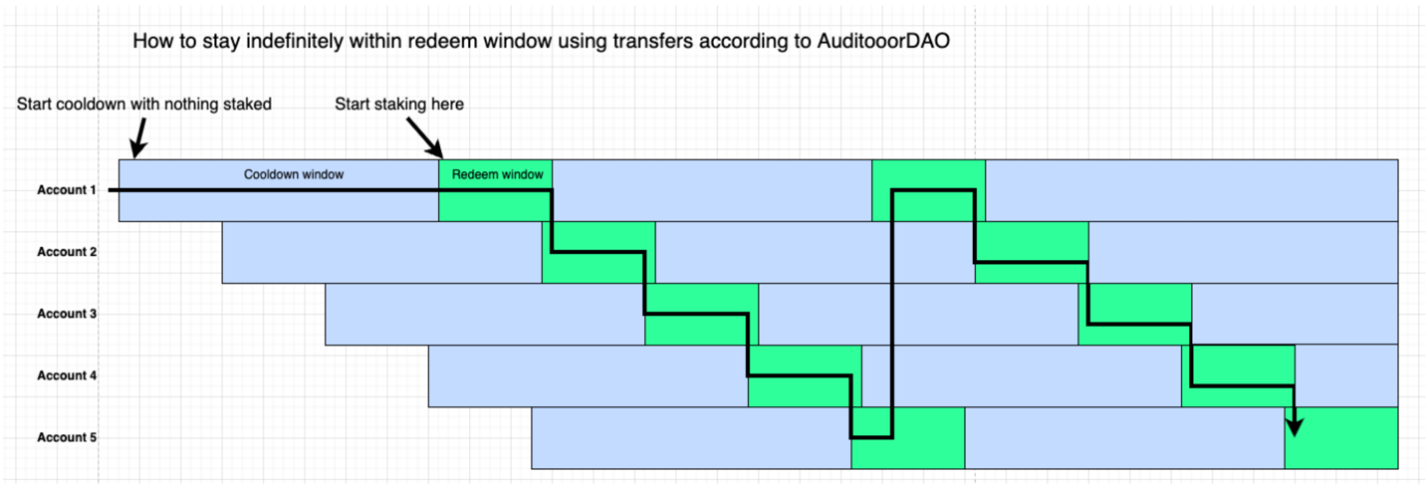
- VS Code

Recommended Mitigation Steps

A few ways to mitigate this problem: Option A: Remove the cooldown/redeem period as it's not really preventing much in current state. Option B: Let the contract start the cooldown on mint, and bind the cooldown/redeem window to the amount that was minted at that time by the account. Don't make sNOTE.startCooldown() available externally. Redeem should verify amount of token available using this new logic.

jeffywu (Notional) confirmed and commented:

> Propose to increase the severity of this [from Low] to High.

> This image is a better way to understand the potential attack.
> 
>

How to stay indefinitely within redeem window using transfers according to AuditooorDAO
> Start cooldown with nothing staked
> Start staking here
> Cooldown window
> Redeem window
> Account 1
> Account 2
> Account 3
> Account 4
> Account 5

pauliax (judge) increased severity to high and commented:

> Great find. Agree with the sponsor, the severity can be upgraded because it destroys the cooldown/redeem protection.

> Could this be mitigated by including an amount (up to the whole user's balance) when starting a cooldown, and then redeem can't withdraw more than specified during the cooldown init?

jeffywu (Notional) commented:

> We've prevented this by refactoring how the redemption window is defined.

[H-03] A Malicious Treasury Manager Can Burn Treasury Tokens By Setting `makerFee` To The Amount The Maker Receives

Submitted by leastwood

The treasury manager contract holds harvested assets/ `COMP` from Notional which are used to perform `NOTE` buybacks or in other areas of the protocol. The manager account is allowed to sign off-chain orders used on 0x to exchange tokens to `WETH` which can then be deposited in the Balancer LP and distributed to `sNOTE` holders.

However, `_validateOrder` does not validate that `takerFee` and `makerFee` are set to zero, hence, it is possible for a malicious manager to receive tokens as part of a swap, but the treasury manager contract receives zero tokens as `makerFee` is set to the amount the maker receives. This can be abused to effectively burn treasury tokens at no cost to the order taker.

Proof of Concept

https://github.com/0xProject/0x-monorepo/blob/0571244e9e84b9ad778bccb99b837dd6f9baaf6e/contracts/exchange/contracts/src/MixinExchangeCore.sol#L196-L250

https://github.com/0xProject/0x-monorepo/blob/0571244e9e84b9ad778bccb99b837dd6f9baaf6e/contracts/exchange-libs/contracts/src/LibFillResults.sol#L59-L91

https://github.com/code-423n4/2022-01-notional/blob/main/contracts/utils/EIP1271Wallet.sol#L147-L188

```
    function _validateOrder(bytes memory order) private view {
        (
            address makerToken,
            address takerToken,
            address feeRecipient,
            uint256 makerAmount,
            uint256 takerAmount
        ) = _extractOrderInfo(order);
        // No fee recipient allowed
        require(feeRecipient == address(0), "no fee recipient allowe

        // MakerToken should never be WETH
        require(makerToken != address(WETH), "maker token must not b

        // TakerToken (proceeds) should always be WETH
        require(takerToken == address(WETH), "taker token must be WE

        address priceOracle = priceOracles[makerToken];

        // Price oracle not defined
        require(priceOracle != address(0), "price oracle not definec

        uint256 slippageLimit = slippageLimits[makerToken];

        // Slippage limit not defined
        require(slippageLimit != 0, "slippage limit not defined");

        uint256 oraclePrice = _toUint(
            AggregatorV2V3Interface(priceOracle).latestAnswer()
        );

        uint256 priceFloor = (oraclePrice * slippageLimit) /
            SLIPPAGE_LIMIT_PRECISION;

        uint256 makerDecimals = 10**ERC20(makerToken).decimals();

        // makerPrice = takerAmount / makerAmount
        uint256 makerPrice = (takerAmount * makerDecimals) / makerAm

        require(makerPrice >= priceFloor, "slippage is too high");
    }
```

Recommended Mitigation Steps

Consider checking that `makerFee == 0` and `takerFee == 0` in `EIP1271Wallet._validateOrder` s.t. the treasury manager cannot sign unfair orders which severely impact the `TreasuryManager` contract.

jeffywu (Notional) confirmed and commented:

> Confirmed, we will fix this.

pauliax (judge) commented:

> Good job warden for identifying this issue with 0x integration.

Medium Risk Findings (7)

[M-01] Usage of deprecated ChainLink API in `EIP1271Wallet`

Submitted by cmichel, also found by 0x1f8b, defsec, leastwood, pauliax, sirhashalot, TomFrenchBlockchain, UncleGrandpa925, and WatchPug

The Chainlink API (`latestAnswer`) used in the `EIP1271Wallet` contract is deprecated:

> This API is deprecated. Please see API Reference for the latest Price Feed API. Chainlink Docs

This function does not error if no answer has been reached but returns 0. Besides, the `latestAnswer` is reported with 18 decimals for crypto quotes but 8 decimals for FX quotes (See Chainlink FAQ for more details). A best practice is to get the decimals from the oracles instead of hard-coding them in the contract.

Recommended Mitigation Steps

Use the `latestRoundData` function to get the price instead. Add checks on the return data with proper revert messages if the price is stale or the round is uncomplete, for example:

```
(uint80 roundID, int256 price, , uint256 timeStamp, uint80 answe
require(answeredInRound >= roundID, "...");
require(timeStamp != 0, "...");
```

pauliax (judge) commented:

> Valid finding. I am hesitating whether this should be low or medium but decided to leave it as a medium because the likeliness is low but the impact would be huge, and all the wardens submitted this with a medium severity. Also: "Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements."

[M-02] `sNOTE.sol#_mintFromAssets()` Lack of slippage control

Submitted by WatchPug, also found by cmichel, hyh, pauliax, TomFrenchBlockchain, and UncleGrandpa925

https://github.com/code-423n4/2022-01-notional/blob/d171cad9e86e0d02e0909eb66d4c24ab6ea6b982/contracts/sNOTE.sol#L195-L209

```
    BALANCER_VAULT.joinPool{value: msgValue}(
        NOTE_ETH_POOL_ID,
        address(this),
        address(this), // sNOTE will receive the BPT
        IVault.JoinPoolRequest(
            assets,
            maxAmountsIn,
            abi.encode(
                IVault.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT,
                maxAmountsIn,
                0 // Accept however much BPT the pool will give us
            ),
            false // Don't use internal balances
        )
    );
```

The current implementation of `mintFromNOTE()` and `mintFromETH()` and `mintFromWETH()` (all are using `_mintFromAssets()` with `minimumBPT` hardcoded to `0`) provides no parameter for slippage control, making it vulnerable to front-run attacks.

Recommendation

Consider adding a `minAmountOut` parameter for these functions.

[jeffywu (Notional) confirmed](#)

[pauliax (judge) commented](#):

> Great find, slippage should be configurable and not hardcoded to 0.

[M-03] No upper limit on `coolDownTimeInSeconds` allows funds to be locked sNOTE owner

Submitted by TomFrenchBlockchain, also found by defsec, Dravee, and Jujic

Inability for sNOTE holders to exit the pool in the case of ownership over SNOTE contract being compromised/malicious.

Proof of Concept

sNOTE works on a stkAAVE model where users have to wait a set cooldown period before being able to reclaim the underlying tokens. This cooldown period can be set to an arbitrary uint32 value in seconds by the owner of the sNOTE contract.

https://github.com/code-423n4/2022-01-notional/blob/d171cad9e86e0d02e0909eb66d4c24ab6ea6b982/contracts/sNOTE.sol#L94-L97

Below in the `startCooldown()` function, it's possible for the owner of the sNOTE contract to choose a value for `coolDownTimeInSeconds` which always causes this function to revert (`_safe32` will always revert if `coolDownTimeInSeconds = type(uint32).max`).

https://github.com/code-423n4/2022-01-notional/blob/d171cad9e86e0d02e0909eb66d4c24ab6ea6b982/contracts/sNOTE.sol#L217-L226

Should ownership over sNOTE become compromised then all of the users' assets may be locked indefinitely.

Recommended Mitigation Steps

Provide a sensible upper limit to `coolDownTimeInSeconds` of, say, a month. This will give plenty of time for NOTE governance to withdraw funds in the event of a shortfall while giving confidence that a user's funds can't be locked forever.

pauliax (judge) commented:

> Valid concern. I was thinking if this should be left as of medium or low severity, but decided this time in favor of wardens: *"Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements."*

[M-04] MAX*SHORTFALL*WITHDRAW limit on BTP extraction is not enforced

Submitted by gellej, also found by gzeon

The function `extractTokensForCollateralShortfall()` allows the owner of the sNote contract to withdraw up to 50% of the total amount of BPT.

Presumably, this 50% limit is in place to prevent the owner from "rug-pulling" the sNote holders (or at least to give them a guarantee that their loss is limited to 50% of the underlying value).

However, this limit is easily circumvented as the function can simply be called a second, third and fourth time, to withdraw almost all of the BPT.

As the contract does not enforce this limit, the bug requires stakers to trust the governance to not withdraw more than 50% of the underlying collateral. This represents a higher risk for the stakers, which may also result in a larger discount on sNote wrt its BPT collateral (this is why I classified the bug as medium risk - users may lose value - not from an exploit, but from the lack of enforcing the 50% rule)

Proof of Concept

See above. The code affected is here: https://github.com/code-423n4/2022-01-notional/blob/main/contracts/sNOTE.sol#L100

Recommended Mitigation Steps

Rewrite the logic and enforce a limit during a time period - i.e. do not allow to withdraw over 50% *per week* (or any time period that is longer than the cooldown period, so that users have time to withdraw their collateral)

jeffywu (Notional) confirmed

pauliax (judge) commented:

> Great find, 50% withdrawal limit can be bypassed by invoking the function multiple times.

[M-05] `sNOTE` Holders Are Not Incetivized To Vote On Proposals To Call `extractTokensForCollateralShortfall`

Submitted by leastwood

As `sNOTE` have governance voting rights equivalent to the token amount in `NOTE`, users who stake their `NOTE` are also able to vote on governance proposals. In the event a majority of `NOTE` is staked in the `sNOTE` contract, it doesn't seem likely that stakers would be willing to vote on a proposal which liquidates a portion of their staked position.

Hence, the protocol may be put into a state where stakers are unwilling to vote on a proposal to call `extractTokensForCollateralShortfall`, leaving Notional insolvent as stakers continue to dump their holdings.

Proof of Concept

https://github.com/code-423n4/2022-01-notional/blob/main/contracts/sNOTE.sol#L99-L129

```
    function extractTokensForCollateralShortfall(uint256 requestedWi
        uint256 bptBalance = BALANCER_POOL_TOKEN.balanceOf(address(t
        uint256 maxBPTWithdraw = (bptBalance * MAX_SHORTFALL_WITHDRA
        // Do not allow a withdraw of more than the MAX_SHORTFALL_WI
        // revert here since there may be a delay between when gover
        // the withdraw actually occurs.
        uint256 bptExitAmount = requestedWithdraw > maxBPTWithdraw ?

        IAsset[] memory assets = new IAsset[](2);
        assets[0] = IAsset(address(WETH));
        assets[1] = IAsset(address(NOTE));
        uint256[] memory minAmountsOut = new uint256[](2);
        minAmountsOut[0] = 0;
        minAmountsOut[1] = 0;

        BALANCER_VAULT.exitPool(
            NOTE_ETH_POOL_ID,
            address(this),
            payable(owner), // Owner will receive the NOTE and WETH
            IVault.ExitPoolRequest(
                assets,
                minAmountsOut,
                abi.encode(
                    IVault.ExitKind.EXACT_BPT_IN_FOR_TOKENS_OUT,
                    bptExitAmount
                ),
                false // Don't use internal balances
            )
        );
    }
```

Recommended Mitigation Steps

Consider redesigning this mechanism to better align stakers with the health of the protocol. It might be useful to allocate a percentage of generated fees to an insurance fund which will be used to cover any collateral shortfall events. This fund can be staked to generate additional yield.

jeffywu (Notional) acknowledged and commented:

> Acknowledged, however, there are technical difficulties with programmatic collateral shortfall detection at this moment. We will look to develop a method that

> allows for programmatic detection in the future (these issues have been discussed with the warden).

pauliax (judge) commented:

> A hypothetical but valid concern.

[M-06] `getVotingPower` Is Not Equipped To Handle On-Chain Voting

Submitted by leastwood

As `NOTE` continues to be staked in the `sNOTE` contract, it is important that Notional's governance is able to correctly handle on-chain voting by calculating the relative power `sNOTE` has in terms of its equivalent `NOTE` amount.

`getVotingPower` is a useful function in tracking the relative voting power a staker has, however, it does not utilise any checkpointing mechanism to ensure the user's voting power is a snapshot of a specific block number. As a result, it would be possible to manipulate a user's voting power by casting a vote on-chain and then have them transfer their `sNOTE` to another account to then vote again.

Proof of Concept

https://github.com/code-423n4/2022-01-notional/blob/main/contracts/sNOTE.sol#L271-L293

```
    function getVotingPower(uint256 sNOTEAmount) public view returns
        // Gets the BPT token price (in ETH)
        uint256 bptPrice = IPriceOracle(address(BALANCER_POOL_TOKEN)
        // Gets the NOTE token price (in ETH)
        uint256 notePrice = IPriceOracle(address(BALANCER_POOL_TOKEN

        // Since both bptPrice and notePrice are denominated in ETH,
        // this formula to calculate noteAmount
        // bptBalance * bptPrice = notePrice * noteAmount
        // noteAmount = bptPrice/notePrice * bptBalance
        uint256 priceRatio = bptPrice * 1e18 / notePrice;
        uint256 bptBalance = BALANCER_POOL_TOKEN.balanceOf(address(t
        // Amount_note = Price_NOTE_per_BPT * BPT_supply * 80% (80/2
        uint256 noteAmount = priceRatio * bptBalance * 80 / 100;

        // Reduce precision down to 1e8 (NOTE token)
        // priceRatio and bptBalance are both 1e18 (1e36 total)
        // we divide by 1e28 to get to 1e8
        noteAmount /= 1e28;

        return (noteAmount * sNOTEAmount) / totalSupply();
    }
```

Recommended Mitigation Steps

Consider implementing a `getPriorVotingPower` function which takes in a `blockNumber` argument and returns the correct balance at that specific block.

jeffywu (Notional) confirmed

pauliax (judge) commented:

> Great find, voting power snapshots would also make the system more resilient to manipulation, e.g. by using flashloans.

[M-07] `_validateOrder` Does Not Allow Anyone To Be A Taker Of An Off-Chain Order

Submitted by leastwood

The `EIP1271Wallet` contract intends to allow the treasury manager account to sign off-chain orders in 0x on behalf of the `TreasuryManager` contract, which holds harvested assets/ `COMP` from Notional. While the `EIP1271Wallet._validateOrder` function mostly prevents the treasury manager from exploiting these orders, it does not ensure that the `takerAddress` and `senderAddress` are set to the zero address. As a result, it is possible for the manager to have sole rights to an off-chain order and due to the flexibility in `makerPrice`, the manager is able to extract value from the treasury by maximising the allowed slippage.

By setting `takerAddress` to the zero address, any user can be the taker of an off-chain order. By setting `senderAddress` to the zero address, anyone is allowed to access the exchange methods that interact with the order, including filling the order itself. Hence, these two order addresses can be manipulated by the manager to effectively restrict order trades to themselves.

Proof of Concept

https://github.com/0xProject/0x-monorepo/blob/0571244e9e84b9ad778bccb99b837dd6f9baaf6e/contracts/exchange-libs/contracts/src/LibOrder.sol#L66

```
    address takerAddress;    // Address that is allowed to fill the c
```

https://github.com/0xProject/0x-monorepo/blob/0571244e9e84b9ad778bccb99b837dd6f9baaf6e/contracts/exchange/contracts/src/MixinExchangeCore.sol#L196-L250

https://github.com/0xProject/0x-monorepo/blob/0571244e9e84b9ad778bccb99b837dd6f9baaf6e/contracts/exchange/contracts/src/MixinExchangeCore.sol#L354-L374

https://github.com/code-423n4/2022-01-notional/blob/main/contracts/utils/EIP1271Wallet.sol#L147-L188

```
    function _validateOrder(bytes memory order) private view {
        (
            address makerToken,
            address takerToken,
            address feeRecipient,
            uint256 makerAmount,
            uint256 takerAmount
        ) = _extractOrderInfo(order);

        // No fee recipient allowed
        require(feeRecipient == address(0), "no fee recipient allowe

        // MakerToken should never be WETH
        require(makerToken != address(WETH), "maker token must not b
        // TakerToken (proceeds) should always be WETH
        require(takerToken == address(WETH), "taker token must be WE

        address priceOracle = priceOracles[makerToken];

        // Price oracle not defined
        require(priceOracle != address(0), "price oracle not definec

        uint256 slippageLimit = slippageLimits[makerToken];

        // Slippage limit not defined
        require(slippageLimit != 0, "slippage limit not defined");

        uint256 oraclePrice = _toUint(
            AggregatorV2V3Interface(priceOracle).latestAnswer()
        );

        uint256 priceFloor = (oraclePrice * slippageLimit) /
            SLIPPAGE_LIMIT_PRECISION;

        uint256 makerDecimals = 10**ERC20(makerToken).decimals();

        // makerPrice = takerAmount / makerAmount
        uint256 makerPrice = (takerAmount * makerDecimals) / makerAn

        require(makerPrice >= priceFloor, "slippage is too high");
    }
```

Tools Used

Manual code review. Discussions with Notional team.

Recommended Mitigation Steps

Consider adding `require(takerAddress == address(0), "manager cannot set taker");` and `require(senderAddress == address(0), "manager cannot set sender");` statements to `_validateOrder`. This should allow any user to fill an order and prevent the manager from restricting exchange methods to themselves.

jeffywu (Notional) confirmed

pauliax (judge) commented:

> Great find, I like when wardens understand and identify issues with integrated external protocols.

Low Risk Findings (8)

- [L-01] Missing validation check in totalSupply() *Submitted by SolidityScan, also found by cmichel and Dravee*
- [L-02] setReserveCashBalance can only set less reserves *Submitted by GeekyLumberjack*
- [L-03] No upper limit check on swap fee Percentage *Submitted by samruna, also found by Jujic*
- [L-04] `getVotingPower` Truncates Result Leading To Inaccuracies In Voting Power *Submitted by leastwood, also found by hyh*
- [L-05] `makerPrice` assumes oracle price is always in 18 decimals *Submitted by cmichel*
- [L-06] Users Can Game `sNOTE` Minting If Buybacks Occur Infrequently *Submitted by leastwood, also found by cmichel*
- [L-07] `extractTokensForCollateralShortfall` Can Be Frontrun By Non-Stakers *Submitted by leastwood*
- [L-08] Conversions between sNOTE and BPT when burning cause less sNOTE to be burned than expected *Submitted by TomFrenchBlockchain, also found by gellej and gzeon*

Non-Critical Findings (16)

- [N-01] safeApprove of openZeppelin is deprecated *Submitted by robee, also found by sirhashalot*
- [N-02] `approve()` return value not checked *Submitted by sirhashalot, also found by 0x1f8b, cmichel, PranavG, robee, and SolidityScan*
- [N-03] Multiple Missing zero address checks *Submitted by SolidityScan, also found by 0v3rf10w, cccz, hyh, Jujic, and robee*
- [N-04] Require with empty message *Submitted by robee*

- [N-05] Improper Contract Upgrades Can Lead To Loss Of Contract Ownership *Submitted by leastwood, also found by robee*
- [N-06] _getToken not resilient to errors *Submitted by 0x1f8b*
- [N-07] TreasuryManager and sNOTE events aren't indexed *Submitted by hyh*
- [N-08] `StorageId` enums may never be shuffled *Submitted by cmichel*
- [N-09] Incorrect comment on cooldown check *Submitted by camden, also found by hyh*
- [N-10] Comment refers to NOTE when it means WETH *Submitted by TomFrenchBlockchain*
- [N-11] `TreasuryAction.sol` : `modifier onlyOwner()` 's revert message is confusing *Submitted by Dravee*
- [N-12] `TreasuryAction.sol:transferReserveToTreasury()` : Missing @return comment *Submitted by Dravee*
- [N-13] Consider making contracts Pausable *Submitted by Jujic, also found by hyh*
- [N-14] Inclusive conditions *Submitted by pauliax, also found by cmichel and Dravee*
- [N-15] Oracle Time Interval Is Small *Submitted by defsec*
- [N-16] Missing parameter validation *Submitted by cmichel*

Gas Optimizations (23)

- [G-01] Gas: Places where both the `return` statement and a named `returns` are used *Submitted by Dravee, also found by Jujic and robee*
- [G-02] Prefix (`++i`), rather than postfix (`i++`), increment/decrement operators should be used in for-loops *Submitted by IllIllI, also found by defsec, Dravee, robee, and throttle*
- [G-03] Remove unnecessary super._beforeTokenTransfer() *Submitted by sirhashalot*
- [G-04] Revert string > 32 bytes *Submitted by sirhashalot, also found by Jujic*
- [G-05] Unused state variables *Submitted by pauliax, also found by gzeon, Jujic, samruna, ShippooorDAO, SolidityScan, throttle, and WatchPug*

- [G-06] Unnecessary inheritance messing with inheritance tree. *Submitted by TomFrenchBlockchain*
- [G-07] Gas: When a function use the `onlyOwner` modifier, use `msg.sender` instead of `owner` *Submitted by Dravee*
- [G-08] Initialisation of zero entries in arrays is unnecessary *Submitted by TomFrenchBlockchain, also found by Jujic and throttle*
- [G-09] Placement of require statement *Submitted by Jujic*
- [G-10] Gas: Use Custom Errors instead of Revert Strings to save Gas *Submitted by Dravee*
- [G-11] Gas in `Bitmap.sol:getMSB()` : unnecessary arithmetic operation *Submitted by Dravee*
- [G-12] Gas in `TreasuryManager.sol` : Inline function `_investWETHToBuyNOTE()` *Submitted by Dravee*
- [G-13] `BalanceHandler.sol:getBalanceStorage()` : `store` is used only once and shouldn't get cached *Submitted by Dravee*
- [G-14] `mintFromNOTE` , `mintFromETH` and `mintFromWETH` can be merged into two functions to give users better experience. *Submitted by TomFrenchBlockchain*
- [G-15] Gas: `reserveInternal.subNoNeg(bufferInternal)` can be unchecked *Submitted by cmichel*
- [G-16] Double _requireAccountNotInCoolDown *Submitted by Tomio, also found by TomFrenchBlockchain*
- [G-17] Optimization on _redeemAndTransfer *Submitted by Tomio, also found by pauliax*
- [G-18] considered changing it to storage *Submitted by Tomio*
- [G-19] Gas Optimization: Unnecessary comparison *Submitted by gzeon*
- [G-20] coolDown.redeemWindowEnd serves no purpose *Submitted by TomFrenchBlockchain*
- [G-21] Require statement on nonzero pool address is impossible to fail *Submitted by TomFrenchBlockchain*
- [G-22] `_investWETHToBuyNOTE` is unnecessarily roundabout. *Submitted by TomFrenchBlockchain*

- [G-23] Gas: Missing checks for non-zero transfer value calls *Submitted by Dravee, also found by Jujic*

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