Notional
Findings & Analysis Report

2021-10-01

Table of contents

Overview

About C4

Code 432n4 (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 Notional smart contract system written in Solidity. The code contest took place between August 25—September 8, 2021.

Wardens

11 Wardens contributed reports to the Notional code contest:

1. cmichel
2. leastwood
3. pauliax
4. tensors
5. gpersoon
6. Omik
7. Jmukesh
8. hrkrshnn
9. a_delamo
10. LSDan
11. ad3sh_

This contest was judged by ghoul.sol.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 38 unique vulnerabilities. All of the issues presented here are linked back to their original findings.

Of these vulnerabilities, 10 received a risk rating in the category of HIGH severity, 7 received a risk rating in the category of MEDIUM severity, and 21 received a risk rating in the category of LOW severity.

C4 analysis also identified 13 non-critical recommendations and 8 gas optimizations.

Scope

The code under review can be found within the C4 Notional code contest repository is comprised of 98 smart contracts written in the Solidity programming language and includes 13,208 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 (10)

[H-01] Self transfer can lead to unlimited mint

Submitted by Omik, also found by gpersoon

The implementation of the transfer function in `nTokenAction.sol` is different from the usual erc20 token transfer function.

This happens because it counts the incentive that the user gets, but with a self-transfer, it can lead to unlimited mint. In L278, it makes the amount negative, but in L279, it returns the value to an amount that is not negative. So, in the L281-282, it finalizes a positive value, only because the negative value is changed to the positive value.

You can interact with this transfer function through nTokenERC20Proxy.sol.

Recommend adding `(sender != recipient)`.

[H-02] DAO proposals can be executed by anyone due to vulnerable `TimelockController`

Submitted by cmichel, also found by leastwood

The `GovernorAlpha` inherits from a vulnerable `TimelockController`. This `TimelockController` allows an `EXECUTOR` role to escalate privileges and also gain the proposer role. See details on OZ and the fix here.

The bug is that `_executeBatch` checks if the proposal was scheduled only **after** the transactions have been executed. This allows inserting a call into the batch that schedules the batch itself, and the entire batch will succeed. As the custom `GovernorAlpha.executeProposal` function removed the original "queued state check" (`require(state(proposalId) == ProposalState.Queued`), the attack can be executed by anyone, even without the `EXEUCTOR_ROLE`.

Proof of concept:

1. Create a proposal using `propose`. The `calldata` will be explained in the next step. (This can be done by anyone passing the min `proposalThreshold`)
2. Call `executeProposal(proposalId, ...)` such that the following calls are made:

```
call-0: grantRole(TIME_LOCK_ADMIN, attackerContract)
call-1: grantRole(EXECUTOR, attackerContract)
call-2: grantRole(PROPOSER, attackerContract)
call-3: updateDelay(0) // such that _afterCall "isOperationReady
call-4: attackerContract.hello() // this calls timelock.schedule
// attackerContract is proposer & executor now and can directly
```

> ℹ I already talked to Jeff Wu about this and he created a test case for it confirming this finding

The impact is that, anyone who can create a proposal can become `Timelock` admin (proposer & executor) and execute arbitrary transactions as the DAO-controlled `GovernorAlpha`. Note that this contract has severe privileges and an attacker can now do anything that previously required approval of the DAO. For example, they could update the `globalTransferOperator` and steal all tokens.

Recommend updating the vulnerable contract to `TimelockController v3.4.2` as it currently uses `OpenZeppelin/openzeppelin-contracts@3.4.0-solc-0.7`

jeffywu (Notional) confirmed

[H-03] `CompoundToNotionalV2.notionalCallback` ERC20 return values not checked

Submitted by cmichel

Some tokens (like USDT) don't correctly implement the EIP20 standard and their `transfer` / `transferFrom` functions return `void`, instead of a success boolean. Calling these functions with the correct EIP20 function signatures will always revert. See `CompoundToNotionalV2.notionalCallback`'s `IERC20(underlyingToken).transferFrom` call.

Tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value. The fact that there is a `cToken` with `USDT` as the underlying this issue directly applies to the protocol.

We recommend using OpenZeppelin's `SafeERC20` versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.

jeffywu (Notional) confirmed

[H-04] Access restrictions on `CompoundToNotionalV2.notionalCallback` can be bypassed

Submitted by cmichel

The `CompoundToNotionalV2.notionalCallback` is supposed to only be called from the verified contract that calls this callback. But, the access restrictions can be circumvented by simply providing `sender = this`, as `sender` is a parameter of the function that can be chosen by the attacker.

```
function notionalCallback(
    address sender,
    address account,
    bytes calldata callbackData
) external returns (uint256) {
// @audit sender can be passed in by the attacker
require(sender == address(this), "Unauthorized callback");
```

An attacker can call the function passing in an arbitrary `account` whose tokens are then transferred to the contract. The `account` first has to approve this contract but this can happen with accounts that legitimately want to call the outer function and have to send a first transaction to approve the contract, but then an attacker front-runs the actual transaction.

It's at least a griefing attack: I can pass in a malicious `cTokenBorrow` that returns any token of my choice (through the `.underlying()` call) but whose `repayBorrowBehalf` is a no-op.

This will lead to any of the victim's approved tokens becoming stuck in the contract, essentially burning them:

```
// @audit using a malicious contract, this can be any token
address underlyingToken = CTokenInterface(cTokenBorrow).underlyi
bool success = IERC20(underlyingToken).transferFrom(account, adc
require(success, "Transfer of repayment failed");

// Use the amount transferred to repay the borrow
// @audit using a malicious contract, this can be a no-op
uint code = CErc20Interface(cTokenBorrow).repayBorrowBehalf(accc
```

Note that the assumption at the end of the function "// When this exits a free collateral check will be triggered" is not correct anymore but I couldn't find a way to make use of it to lead to an invalid account state.

Recommend fixing the authorization check.

jeffywu (Notional) confirmed

[H-05] Access restrictions on `NotionalV1ToNotionalV2.notionalCallback` can be bypassed

Submitted by cmichel, also found by pauliax

The `NotionalV1ToNotionalV2.notionalCallback` is supposed to only be called from the verified contract that calls this callback but the access restrictions can be circumvented by simply providing `sender = this` as `sender` is a parameter of the function that can be chosen by the attacker.

```
function notionalCallback(
    address sender,
    address account,
    bytes calldata callbackData
) external returns (uint256) {
    require(sender == address(this), "Unauthorized callback");
```

An attacker can call the function passing in an arbitrary `account` whose tokens can then be stolen. The `account` first has to approve this contract but this can happen with accounts that legitimately want to migrate their tokens and therefore have to send a first transaction to approve the contract, but then an attacker frontruns the actual migration transaction.

The attacker can steal the tokens by performing an attack similar to the following:

- first transaction is used to withdraw the victim's funds to the contract. This can be done by choosing `account=victim`, `v1RepayAmount=0`, `v1CollateralId=WBTC`, `v2CollateralId=DAI`. The `NotionalV1Erc1155.batchOperationWithdraw` (not part of this contest) will withdraw the victim's funds to this contract. Note that the attacker has to deposit the same `v2CollateralBalance = uint256(collateralBalance)` for the victim into the V2 version, but they can choose different cheaper collateral (for example, withdraw WBTC, deposit same amount of DAI).

- second transaction is now used to deposit the victim funds in the contract into the user's account. They use `account=attacker`, `v1DebtCurrencyId=WBTC`, `v1RepayAmount=amount` to deposit it into Notional V1. (They need to have a small `collateralBalance`, etc. to pass all checks).

Recommend fixing the authorization check.

[H-06] `TokenHandler.safeTransferOut` does not work on non-standard compliant tokens like USDT

Submitted by cmichel

The `TokenHandler.safeTransferOut` function uses the standard `IERC20` function for the transfer call and proceeds with a `checkReturnCode` function to handle non-standard compliant tokens that don't return a return value. However, this does not work, as calling `token.transfer(account, amount)` already reverts if the token does not return a return value, as `token`'s `IERC20.transfer` is defined to always return a `boolean`.

The impact is that, when using any non-standard compliant token like USDT, the function will revert. Deposits for these tokens are broken, which is bad as `USDT` is a valid underlying for the `cUSDT` cToken.

We recommend using OpenZeppelin's `SafeERC20` versions with the `safeApprove` function that handles the return value check as well as non-standard-compliant tokens.

jeffywu (Notional) confirmed

[H-07] `TokenHandler.safeTransferIn` does not work on non-standard compliant tokens like USDT

Submitted by cmichel

The `TokenHandler.safeTransferIn` function uses the standard `IERC20` function for the transfer call and proceeds with a `checkReturnCode` function to handle non-standard compliant tokens that don't return a return value. However, this does not work, as calling `token.transferFrom(account, amount)` already reverts if the token does not return a return value, as `token` 's `IERC20.transferFrom` is defined to always return a `boolean` .

When using any non-standard compliant token like USDT, the function will revert. Withdrawals for these tokens are broken, which is bad as `USDT` is a valid underlying for the `cUSDT` cToken.

We recommend using [OpenZeppelin's `SafeERC20`] versions with the `safeApprove` function that handles the return value check as well as non-standard-compliant tokens.

[jeffywu (Notional) confirmed]

[H-08] DOS by Frontrunning NoteERC20 `initialize()` Function

Submitted by leastwood

The `scripts/` folder outlines a number of deployment scripts used by the Notional team. Some of the contracts deployed utilize the ERC1967 upgradeable proxy standard. This standard involves first deploying an implementation contract and later a proxy contract which uses the implementation contract as its logic.

When users make calls to the proxy contract, the proxy contract will delegate call to the underlying implementation contract. `NoteERC20.sol` and `Router.sol` both implement an `initialize()` function which aims to replace the role of the `constructor()` when deploying proxy contracts. It is important that these proxy contracts are deployed and initialized in the same transaction to avoid any malicious front-running.

However, `scripts/deployment.py` does not follow this pattern when deploying `NoteERC20.sol` 's proxy contract. As a result, a malicious attacker could monitor the Ethereum blockchain for bytecode that matches the `NoteERC20` contract and front-run the `initialize()` transaction to gain ownership of the contract. This can be repeated as a Denial Of Service (DOS) type of attack, effectively preventing Notional's contract deployment, leading to unrecoverable gas expenses. See `deployment.py` **L44-L60**, and `deploy_governance.py` **L71-L105**.

As the `GovernanceAlpha.sol` and `NoteERC20.sol` are co-dependent contracts in terms of deployment, it won't be possible to deploy the governance contract before deploying and initializing the token contract. Therefore, it would be worthwhile to ensure the `NoteERC20.sol` proxy contract is deployed and initialized in the same transaction, or ensure the `initialize()` function is callable only by the deployer of the `NoteERC20.sol` contract. This could be set in the proxy contracts `constructor()`.

jeffywu (Notional) confirmed

[H-09] Potential DOS in Contracts Inheriting `UUPSUpgradeable.sol`

Submitted by leastwood

There are a number of contracts which inherit `UUPSUpgradeable.sol`, namely; `GovernanceAction.sol`, `PauseRouter.sol` and `NoteERC20.sol`.

All these contracts are deployed using a proxy pattern whereby the implementation contract is used by the proxy contract for all its logic. The proxy contract will make delegate calls to the implementation contract. This helps to facilitate future upgrades by pointing the proxy contract to a new and upgraded implementation contract.

However, if the implementation contract is left uninitialized, it is possible for any user to gain ownership of the `onlyOwner` role in the implementation contract for `NoteERC20.sol`. Once the user has ownership they are able to perform an upgrade of the implementation contract's logic contract and delegate call into any arbitrary contract, allowing them to self-destruct the proxy's implementation contract. Consequently, this will prevent all `NoteERC20.sol` interactions until a new implementation contract is deployed.

Initial information about this issue was found here.

Consider the following scenario:

- Notional finance deploys their contracts using their deployment scripts. These deployment scripts leave the implementation contracts uninitialized. Specifically the contract in question is `NoteERC20.sol`.
- This allows any arbitrary user to call `initialize()` on the `NoteERC20.sol` implementation contract.
- Once a user has gained control over `NoteERC20.sol`'s implementation contract, they can bypass the `_authorizeUpgrade` check used to restrict upgrades to the `onlyOwner` role.
- The malicious user then calls `UUPSUpgradeable.upgradeToAndCall()` shown here which in turn calls this function. The new implementation contract then points to their own contract containing a self-destruct call in its fallback function.
- As a result, the implementation contract will be self-destructed due to the user-controlled delegate call shown here, preventing all future calls to the `NoteERC20.sol` proxy contract until a new implementation contract has been deployed.

Recommend considering initializing the implementation contract for `NoteERC20.sol` and checking the correct permissions before deploying the proxy contract or performing any contract upgrades. This will help to ensure the implementation contract cannot be self-destructed.

jeffywu (Notional) acknowledged and disagreed with severity:

> Acknowledged, I don't think this should be categorized high risk because the worst case is a denial of service and a redeployment of the ERC20 contract. As it stands, we've already successfully deployed our ERC20 contract so this is a non-issue.

> I would categorize as `0 (Non-critical)`

adamavenir (organizer) commented:

> Warden leastwood added this proof of concept to illustrate the vulnerability https://gist.github.com/leastwood/b23d9e975883c817780116c2ceb785b8

jeffywu (Notional) commented:

> Ok I retract my previous statement, I misread the issue description. Up to you guys but do you want to pay out a full amount to someone who is reporting issues discovered elsewhere? OZ has already called initialize on our deployed contract for us.

adamavenir (organizer) commented:

> @jeffywu (Notional) I think the question is whether the issue is valid based on the original code base. Given your initial response and change after his proof of concept, my read was there was value here in what he reported. Is that a correct understanding?

jeffywu (Notional) commented:

> There was value added here but perhaps not at the same level as the other high risk issues.

adamavenir (organizer) commented:

> @jeffywu (Notional) Thanks for the input. As per our rules, awards are determined strictly based on the judge's assessment of the validity and severity, so we'll see how our judge chooses to score this.

[H-10] Liquidity token value can be manipulated

Submitted by cmichel

The liquidity token value (`AssetHandler.getLiquidityTokenValue`) is the sum of the value of the individual claims on cash (underlying or rather cTokens) and fCash. The amount to redeem on each of these is computed as the LP token to redeem relative to the total LP tokens, see `AssetHandler.getCashClaims` / `AssetHandler.getHaircutCashClaims` :

```
    // @audit token.notional are the LP tokens to redeem
    assetCash = market.totalAssetCash.mul(token.notional).div(market
    fCash = market.totalfCash.mul(token.notional).div(market.totalLi
```

This means the value depends on the **current market reserves** which can be manipulated. You're essentially computing a spot price (even though the individual values use a TWAP price) because you use the current market reserves which can be manipulated.

See the "How do I tell if I'm using spot price?" section here.

> However, by doing this you're actually incorporating the spot price because you're still dependent on the reserve balances of the pool. This is an extremely subtle detail, and more than one project has been caught by it. You can read more about this footgun in this writeup by @cmichelio.

The value of an LP token is computed as `assetCashClaim + assetRate.convertFromUnderlying( presentValue(fCashClaim) )`, where `(assetCashClaim, fCashClaim)` depends on the current market reserves which can be manipulated by an attacker via flashloans. Therefore, an attacker trading large amounts in the market can either increase or decrease the value of an LP token.

If the value decreases, they can try to liquidate users borrowing against their LP tokens / nTokens. If the value increases, they can borrow against it and potentially receive an under-collateralized borrow this way, making a profit.

The exact profitability of such an attack depends on the AMM as the initial reserve manipulation and restoring the reserves later incurs fees and slippage. In constant-product AMMs like Uniswap it's profitable and several projects have already been exploited by this, like warp.finance. However, Notional Finance uses a more complicated AMM and the contest was too short for me to do a more thorough analysis. It seems like a similar attack could be possible here as described by the developers when talking about a different context of using TWAP oracles:

> "Oracle rate protects against short term price manipulation. Time window will be set to a value on the order of minutes to hours. This is to protect fCash valuations from market manipulation. For example, a trader could use a flash loan to dump a large amount of cash into the market and depress interest rates. Since we value fCash in portfolios based on these rates, portfolio values will decrease and they may then be liquidated." - Market.sol L424

Recommend not using the current market reserves to determine the value of LP tokens. Also, think about how to implement a TWAP oracle for the LP tokens themselves, instead of combining it from the two TWAPs of the claimables.

T-Woodward (Notional) confirmed and disagreed with severity :

> It is true that a flash loan could be used to manipulate the value of a liquidity token's cash and fCash claims. This issue can potentially cause accounts to be liquidated which shouldn't be, but not for the reasons stated in this issue. I'll explain what actually can go wrong, and why the fix is simple and non-invasive.

> First, to restate the issue: The manipulator could borrow or lend a large amount to a liquidity pool, which would change the amount of cash and fCash sitting in that pool and the corresponding cash and fCash claims of a liquidity token associated with that pool. This could change the liquidity token's net value within the space of a transaction despite the fact that the oracleRate used to value fCash is lagged and manipulation resistant.

> But it is not true that this manipulation could decrease the value of a liquidity token - in fact it could only increase a liquidity token's value. By borrowing or lending a large amount using a flash loan, the interest rate that the attacker would receive would deviate from the oracleRate in favor of the liquidity provider. If the attacker executed a large lend order, the interest rate on the loan would be significantly below the oracleRate. This would mean that the liquidity providers had borrowed at a below-market rate and that the net value of that trade would be positive for them. Conversely if the attacker executed a large borrow order, the interest rate on the loan would be significantly above the oracleRate. Again, this would mean that the net value of that trade would be positive for the liquidity providers because they would effectively be lending at an above-market rate. In either case, the value of the liquidity token would increase, not decrease.

> However, even though the value of a liquidity token could only increase during such an attack, the collateral value of the liquidity token could decrease once the haircuts were applied in the free collateral calculation. The reason for this is that fCash claims are effectively double-haircut (once by the liquidity token haircut and once by the fCash haircut), whereas cash claims are only haircut once (by the liquidity token haircut). This means that even though the attack would increase the value of the liquidity token without haircuts, once you consider the haircuts applied in the free collateral calculation, the collateral value of the liquidity token

> can be decreased and accounts could become undercollateralized and eligible for liquidation.

> **Remediation:** The immediate remediation for this issue is to restrict providing liquidity to the nToken account exclusively. In the longer term, we will plan to add TWAPs to determine the collateral value of liquidity token cash and fCash claims. This immediate remediation will be fine for now though, and will not degrade the system for two reasons:

1. The team does not anticipate users providing liquidity directly outside of the nToken (we don't even offer a way to do it within the UI for example). Only nToken holders receive NOTE incentives, not direct liquidity providers.
2. The nToken accounts are safe from this attack because the maximum amount that an attacker could temporarily decrease the collateral value of liquidity tokens could never be enough to cause the nToken accounts to become undercollateralized, and therefore they would never be at risk of liquidation due to this attack. The TLDR here is that this attack can't actually decrease the collateral value of liquidity tokens all that much, and so for an account to be vulnerable they would have to be running quite close to minimum collateralization. This will never happen for the nToken because it doesn't borrow, it just provides liquidity and always maintains very healthy collateralization levels.

ghoul-sol (judge) commented:

> Again, I gave it some thought and I think that this is high risk. Keeping as is.

Medium Risk Findings (7)

[M-01] TokenHandler.sol, L174 - `.transfer` is bad practice

Submitted by JMukesh, also found by tensors

The use of `.transfer` in `TokenHandler.sol` L174 to send ether is now considered bad practice as gas costs can change which would break the code.

See stop using soliditys transfer now, and istanbul hardfork eips increasing gas costs and more.

Recommend using `call` instead, and make sure to check for reentrancy.

jeffywu (Notional) commented:

> This should be `0 (Non critical)`. ETH transfers still work as of today. This is a best practice recommendation. In the case that there is a hard fork that breaks ETH transfers the protocol can be upgraded to support the use of `call()`

ghoul-sol (judge) commented:

> I aline with warden on this one. `.transfer` will not work if the account is a smart contract. Keeping as is.

EDITORS NOTE: *Additional conversation regarding this vulnerability can be found* here

[M-02] `.latestRoundData()` does not update the oracle - `ExchangeRate.sol`

*Submitted by a*delamo, also found by tensors, JMukesh, cmichel and defsec_

The method `.latestRoundData()` on an oracle returns the latest updated price from the oracle, but this is not the current price of an asset. To get an accurate current price you need to query it by calling the oracle and waiting for a callback to fulfill the request.

Inaccurate price data could quickly lead to a large loss of funds. Suppose the price of an asset changes downward 5% but your oracle is not updated. A user could deposit funds (credited with an extra 5% since the oracle isn't updated), wait until `.latestRoundData()` updates (or update it himself) and becomes accurate. He then withdraws to the same asset he put in for an extra 5%. `ExchangeRate.sol` L84

Recommend not fetching the latest price (having to call the oracle to update the price instead), and then waiting for the callback.

[M-03] Allowance checks not correctly implemented

Submitted by cmichel

The `nTokenAction` implements two token approvals, the `nTokenWhitelist` which is always used first, and the `nTokenAllowance` which is checked second. If the `nTokenWhitelist` does *not* have enough allowance for the transfer, the transaction fails, even in the case where `nTokenAllowance` still has enough allowance.

Transfers that have sufficient allowance fail in certain cases.

Recommend that, instead of reverting if the `nTokenWhitelist` allowance is not enough, default to the `nTokenAllowance` case.

Something like this:

```
uint256 requiredAllowance = amount;

uint256 allowance = nTokenWhitelist[from][spender];
// use whitelist allowance first
if (allowance > 0) {
    uint256 min = amount < allowance ? amount : allowance;
    requiredAllowance -= min;
    allowance = allowance.sub(min);
    nTokenWhitelist[from][spender] = allowance;
}

// use currency-specific allowance now
if(requiredAllowance > 0)
    // This is the specific allowance for the nToken.
    allowance = nTokenAllowance[from][spender][currencyId];
    require(allowance >= requiredAllowance, "Insufficient allowa
    allowance = allowance.sub(requiredAllowance);
    nTokenAllowance[from][spender][currencyId] = allowance;
}
```

[jeffywu (Notional) confirmed]

[M-04] `CompoundToNotionalV2.enableToken` ERC20 missing return value check

Submitted by cmichel

The `enableToken` function performs an `ERC20.approve()` call but does not check the `success` return value. Some tokens do **not** revert if the approval failed, returning `false` instead.

The impact is that, tokens that don't actually perform the approve and return `false` are still counted as a correct approve.

Recommend using [OpenZeppelin's `SafeERC20`]() versions with the `safeApprove` function that handles the return value check as well as non-standard-compliant tokens.

[jeffywu (Notional) confirmed]()

[M-05] `nTokenERC20Proxy` emits events even when not success

Submitted by cmichel

The `nTokenERC20Proxy` functions emit events all the time, even if the return value from the inner call returns `false`, indicating an unsuccessful action.

An off-chain script scanning for `Transfer` or `Approval` events can be tricked into believing that an unsuccessful transfer was indeed successful. This happens in the `approve`, `transfer` and `transferFrom` functions.

Recommend only emitting events on `success`.

[jeffywu (Notional) confirmed]()

[M-06] `TokenHandler.setToken` ERC20 missing return value check

Submitted by cmichel

The `setToken` function performs an `ERC20.approve()` call but does not check the `success` return value. Some tokens do **not** revert if the approval failed but return `false` instead.

The impact is that tokens that don't actually perform the approve and return `false` are still counted as a correct approve.

We recommend using OpenZeppelin's `SafeERC20` versions with the `safeApprove` function that handles the return value check as well as non-standard-compliant tokens.

jeffywu (Notional) confirmed

[M-07] Attackers can force liquidations by borrowing large amounts of an asset.

Submitted by tensors

Consider an attacker who borrows enough to greatly increase the oracle rate. It is claimed that arbitrageurs will come in and fix this discrepancy before the attacker has a chance to profit off of his price manipulation:

> "Over the next 1 hour, the effect of the new 12% interest rate will be averaged into the previous 6% rate. This forces the borrower to hold their position and gives an opportunity for other traders to lend to the market to bring the interest rate back down to its previous 6% level."

In my opinion, this incentive is not sufficient to prevent an attack. This assumes that:

1. There is sufficient volume to notice a manipulation like this
2. The arbitrageurs would be willing to deploy capital for a short amount of for a slightly increased rate
3. The arbitrageurs would now know that this is a manipulation, and not a natural market movement (For example, lets say an asset lending rate goes up 10% in value, is it being manipulated or is the rate actually worth 10% more for some reason? An arbitrageur needs to make this before he deploys capital). Since

notional is the only market to offer something like this, it is difficult to discern what the response should be.

4. The arbitrageurs don't join in on the attack, and try to liquidate accounts with the attacker

Proof of concept is based off of the formula and text here.

Uncertain what the recommendation should be.

T-Woodward (Notional) acknowledged and disagreed with severity:

> This is a fair point. With a timeWindow value that is under an hour, it's not clear whether capital will flow in to take advantage of artificially inflated yields before the oracleRate catches up to the lastImpliedRate and accounts are potentially eligible for liquidation at this early stage in Notional's lifespan before there are lots of eyes on the system and its market dynamics are widely understood.

> However, it's worth noting that there are several mitigating circumstances which make such an attack unlikely:

- The necessary conditions for a profitable attack to exist are quite narrow
- The capital required to execute such an attack is substantial
- The sunk cost of executing this attack prior to any payoff is significant
- The payoff of the attack is not guaranteed

> First let's examine what this attack would have to look like.

- With the current fCash market parameters, interest rates top out at about 25%, so the most that an attacker could push interest rates from their natural level is ~20% (assuming a starting interest rate of 5%).
- With the current active maturities (longest dated maturity is one year), a 20% change in interest rates could decrease the collateral value of an account's fCash by a maximum of ~20%.
- Pushing the interest rate to the top of the range requires the attacker to borrow all of the cash sitting in the liquidity pool. If you assume there is $100M sitting in the pool, then the attacker would have to borrow $100M. At worst, if they execute the borrow all in one shot, their realized borrow rate would be 25%. At

best, if they execute the borrow in pieces (increasing the time taken and risk of the attack), their realized borrow rate would be ~15%. This implies that the attacker has placed at least $15M at risk (his total interest owed) as there is no guarantee that he can exit his position in profit or at cost.

- In order for this to be a profitable attack, the attacker needs to offset their borrowing by executing an fCash liquidation that would allow them to lend at the artificially inflated interest rate. This means that there must be an account, or set of accounts, which have borrowed against their fCash, are 20% away from under collateralization, and have enough fCash such that the attacker can offset his borrow by liquidating their fCash. In other words, there would need to be $100M+ of fCash held in the system that is being used as collateral and is close to liquidation.
- These conditions are possible, but improbable. It would imply that the amount of outstanding loans that are being borrowed against (and are close to under collateralization) is greater than the total liquidity in the pool. This strikes me as unlikely for two reasons:
- For there to be more fCash outstanding than liquidity in the pool, there would have to be a lot of two-way trading. We expect this to happen eventually, but in the initial stages we would be very surprised by this because lenders do not earn NOTE incentives, only liquidity providers earn NOTE incentives. It would be very surprising to see more lending than liquidity given this fact. We expect more lending than liquidity once Notional is more mature, but by then there will be more eyes on the system which would make this attack less likely as arbitragers and/or automated yield aggregators really would come in to push rates back down to their natural level.
- Only a percentage of fCash will actually be used as collateral, and probably not a large percentage. So if you specifically need fCash used as collateral (and close to liquidation) to be a multiple of liquidity in the pool, this would imply that the amount of total outstanding fCash would be like an order of magnitude greater than the liquidity in the pool. That strikes me as highly unlikely.

> Ultimately we don't think this is a significant risk because the necessary conditions are unlikely to obtain in the early stages of Notional's growth, the attack involves substantial sunk costs and capital committed, and the risk of an unsuccessful attack is significant. To further mitigate against this unlikely risk, we will skew the timeWindow to a longer time / larger value.

ghoul-sol (judge) commented:

> This attack requires a lot of things aligned and for that single reason I'll give it medium risk.

Low Risk Findings (21)

[L-01] Incorrect event parameters in `transferFrom` function

Submitted by pauliax, also found by JMukesh

Different parameter are being set in `Approval` event in `transferFrom()`

```
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external override returns (bool) {
        (bool success, uint256 newAllowance) =
            proxy.nTokenTransferFrom(currencyId, msg.sender, from, t

        // Emit transfer events here so they come from the correct c
        emit Transfer(from, to, amount);

    // here first parameter should be owner and second should be spe
    //   as mentioned in ntokenErc20.sol that is :
    // event Approval(address indexed owner, // address indexed sper

        emit Approval(msg.sender, from, newAllowance);

        return success;
    }
```

The impact is that, this error may negatively impact off-chain tools that are monitoring critical transfer events of the token. See `nTokenERC20Proxy.sol` L100.

jeffywu (Notional) disagreed with severity:

> Duplicate #55, dispute the categorization. This should be Low Risk.

ghoul-sol (judge) commented:

> External services might be affected but it’s not clear how significant it would be. Most of the time events are not critical. Making this low risk.

[L-02] `Address.isContract` with no check of returned value

Submitted by JMukesh, also found by pauliax

The function `activateNotional` calls `Address.isContract(...)` but does not check the returned value, thus making this call pretty much useless:

```
    Address.isContract(address(notionalProxy_));
```

Recommend wrapping this in a require statement.

jeffywu (Notional) confirmed

[L-03] lack of input validation of arrays

Submitted by JMukesh

```
    function migrateBorrowFromCompound(
        address cTokenBorrow,
        uint256 cTokenRepayAmount,
        uint16[] memory notionalV2CollateralIds,
        uint256[] memory notionalV2CollateralAmounts,
        BalanceActionWithTrades[] calldata borrowAction
    ) ;
```

if the array length of `notionalV2CollateralId` , `notionalV2CollateralAmounts` and `borrowAction` is not equal, it can lead to an error. See `CompoundToNotionalV2.sol` L24.

Recommend checking the input array length.

jeffywu (Notional) confirmed

[L-04] No Transfer Ownership Pattern

Submitted by leastwood, also found by JMukesh

The current ownership transfer process involves the current owner calling `NoteERC20.transferOwnership()`. This function checks that the new owner is not the zero address and proceeds to write the new owner's address into the owner's state variable. If the nominated EOA account is not a valid account, it is entirely possible the owner may accidentally transfer ownership to an uncontrolled account, breaking all functions with the `onlyOwner()` modifier. See `NoteERC20.sol` **L123-L127**.

Recommend considering implementing a two step process where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of ownership to fully succeed. This ensures the nominated EOA account is a valid and active account.

jeffywu (Notional) confirmed

[L-05] `initialize` functions can be front-run

Submitted by cmichel

The `initialize` function that initializes important contract state can be called by anyone.

Occurences:

- `NoteERC20.initialize`
- `Router.initialize`

The attacker can initialize the contract before the legitimate deployer, hoping that the victim continues to use the same contract. In the best case for the victim, they notice it and have to redeploy their contract costing gas.

Recommend using the constructor to initialize non-proxied contracts. For initializing proxy contracts, recommend deploying contracts using a factory contract that immediately calls `initialize` after deployment, or make sure to call it immediately after deployment and verify the transaction succeeded.

jeffywu (Notional) confirmed

[L-06] `ERC1155Action` returns `false` on `supportsInterface` with the real ERC1155 interface

Submitted by cmichel

As the return value of `ERC1155.balanceOf` was changed to a signed integer, the `nERC1155Interface` does not implement the `ERC1155` interface and the `supportsInterface` call will return false if people call it with the actual `ERC1155` interface ID.

Not all users of the contract might care about the `balance` function and call `supportsInterface` with the original EIP1155 interface. The contract will still deny the *[content missing]*

It is indeed debatable if this contract should be considered implementing ERC1155 and what the correct return value of `supportsInterface(ERC1155.interface)` should be for compatibility. Users need to be aware that this contract is not standard compliant and that the `supportsInterface` call will fail.

jeffywu (Notional) confirmed

[L-07] ERC1155 has reentrancy possibilities.

Submitted by cmichel, also found by tensors

ERC1155 tokens have a callback on transfer, making reentrancy a possibility. I haven't been able to find any reentrancy, but having extra external function calls isn't safe. If it's necessary to use an ERC1155 there is nothing you can do about it, but otherwise consider just using an ERC20.

See `actions/ERC1155Action` sol.

Recommend confirming that using tokens with callbacks is really necessary for the protocol to function.

jeffywu (Notional) disputed and disagreed with severity:

> Callbacks are required as part of the ERC1155 spec. Duplicate #62. Severity should be Low or Non Critical.

ghoul-sol (judge) commented:

> Duplicate of #62 ergo low risk

[L-08] Open TODOs in `ERC1155Action`

Submitted by cmichel

The `ERC1155Action._checkPostTransferEvent` has open TODOs:

```
    // TODO: retrieve revert string
    require(status, "Call failed");
```

Open TODOs can hint at programming or architectural errors that still need to be fixed.

Recommend resolving the TODO and bubble up the error.

jeffywu (Notional) confirmed

[L-09] Router calls to `nTokenAction.nTokenTransferApprove` fail

Submitted by cmichel

The `Router` forwards `nTokenTransferApprove` calls to the `nTokenAction` implementation. However, these always fail due to the `msg.sender == nTokenAddress` check.

This call failing seems to be the intended behavior but it shouldn't even be forwarded in the Router.

Recommend removing `sig == nTokenAction.nTokenTransferApprove.selector` from the `getRouterImplementation` as it indicates that this is a valid function call.

jeffywu (Notional) disputed:

> Calling approve on the nToken will forward the call to the Router which will then delegate call to the `nTokenTransferApprove` method. This is the intended functionality and will pass the require statement because the delegate call does not change the `msg.sender`

[L-10] Unclear decimals value in `cTokenAggregator`

Submitted by cmichel

The `cTokenAggregator.decimals` value is set to `18` but `cTokens` only have `8` decimals. It's unclear what this `decimals` field refers to.

If it should refer to the `cToken` decimals, it's wrong and should be set to `8`. This value is not used inside the contract but it's `public` and anyone can read it.

jeffywu (Notional) confirmed:

> Decimals refers to the decimals in the exchange rate, but we should add a comment here. Agree it is confusing.

[L-11] Governor average block time is not up-to-date

Submitted by cmichel

The `GovernorAlpha.MIN_VOTING_PERIOD_BLOCKS = 6700` value indicates an average block time of 12.8956s which was correct a year ago, but at the moment a more accurate block time would be 13.2s, see blocktime.

Recommend using a `MIN_VOTING_PERIOD_BLOCKS` of `6545`.

jeffywu (Notional) confirmed

[L-12] NoteERC20 missing initial ownership event

Submitted by cmichel

The `NoteERC20.initialize` function does not emit an initial `OwnershipTransferred` event.

Recommend emitting `OwnershipTransferred(address(0), owner_)` in `initialize`.

jeffywu (Notional) confirmed

[L-13] `TokenHandler.transfer` wrong branch order

Submitted by cmichel

The `TokenHandler.transfer` should handle the `if (token.tokenType == TokenType.Ether)` case first, as if the token type is `Ether` but `netTransferExternal <= 0` it treats the token as an `ERC20` token and tries to call `ERC20` functions on it.

Luckily, trying to call ERC20 functions on the invalid token address will revert which is the desired behavior.

We still recommend reordering the branches and adding a `netTransferExternal <= 0` check. The code becomes cleaner and it's more obvious that the transaction will fail.

jeffywu (Notional) confirmed

[L-14] `DateTime.isValidMarketMaturity` bounds should be tighter

Submitted by cmichel

`DateTime.isValidMarketMaturity` can be called with a `maxMarketIndex < 10` but the inner `DateTime.getTradedMarket(i)` function will revert for any values `i > 7`.

The impact is that "Valid" `maxMarketIndex` values above 7 will break and return with an error.

Recommend that the upper bound on `maxMarketIndex` should be set to `7`.

jeffywu (Notional) confirmed

[L-15] `DateTime.getMarketIndex` bounds should be tighter

Submitted by cmichel

`DateTime.getMarketIndex` can be called with a `maxMarketIndex < 10` but the inner `DateTime.getTradedMarket(i)` function will revert for any values `i > 7`.

"Valid" `maxMarketIndex` values above 7 will break and return with an error.

The upper bound on `maxMarketIndex` should be set to `7`.

jeffywu (Notional) confirmed

[L-16] Double check for "birthday" collision

Submitted by gpersoon

The function `getRouterImplementation` of `Router.sol` checks the selectors of functions and calls the appropriate function. Selectors are only 4 bytes long, so there is a theoretical probability of a collision (e.g. two functions having the same selector).

This is comparable to the "birthday attack". The probability of a collision when you have 93 different functions is 10^−6. Due to the structure of the `Router.sol`, the solidity compiler does not prevent collisions

Recommend double checking (perhaps via a continuous integration script / github workflow), that there are no collisions of the selectors.

- jeffywu (Notional) confirmed

[L-17] `notionalCallback` returns no value

Submitted by pauliax

The function `notionalCallback` (in `NotionalV1ToNotionalV2` and `CompoundToNotionalV2`) declares to return uint, however, no actual value is returned.

Recommend either removing the return declaration or returning the intended value (I assume it may return a value that it gets from `depositUnderlyingToken` / `depositAssetToken`). Otherwise, it may confuse other protocols that later may want to integrate with you.

jeffywu (Notional) confirmed

[L-18] `NotionalV1ToNotionalV2` should reject ETH transfers from others than WETH

Submitted by pauliax

The contract `NotionalV1ToNotionalV2` has an empty receive function which allows it to receive Ether. I suppose this was needed to receive ETH when withdrawing from WETH. As there is no way to send out accidentally sent ETH from this contract, I suggest adding an auth check to this receive function to only accept ETH from WETH contract.

```
    require(msg.sender == address(WETH), "Not WETH");
```

jeffywu (Notional) confirmed

[L-19] No checks on target variable

Submitted by tensors

Lack of checks on target could lead to loss of funds in `Reservoir.sol` L50.

Recommend requiring that target is non-zero.

jeffywu (Notional) confirmed

[L-20] Some `TradingActions` do not have front-running protections

Submitted by tensors

Some of the actions in `TradingAction.sol` can be front-run. Since there are no slippage protections, its unclear how bad this problem can be. See `TradingAction.sol` L334.

An example is `_settleCashDebt()`. This goes through `_getfCashSettleAmount()` which uses an `impliedRate` variable. This can be manipulated by a frontrunner. Add checks that exist on the other trade types.

Recommend adding `minAmountOut` / `minAmountCredited` as function variables to protect against frontrunning. For example. `_executeLiquidityTrade` has such protections in place.

T-Woodward (Notional) acknowledged but disagreed with severity:

> I don't think this is a particularly serious issue in practice, given that _getfCashSettleAmount is calculated off the oraclePrice which is manipulation-resistant, but it is still a valid concern.

> **Remediation:** Add slippage guards similar to executing a normal lend/borrow on the AMM

jeffywu (Notional) commented:

> Severity should be `1 Low Risk`. It is true that you can potentially front run settle cash debt but we will not fix this as there is no room in the calldata for an additional parameter. The likelihood of manipulating the oracle rate for a given time window is low.

ghoul-sol (judge) commented:

> As the sponsor said, there is no immediate risk of losing funds and it's not clear how this could be manipulated is the oracle price is a base to calculate the settlement amount. Making this low risk.

[L-21] `NoteERC20.getPriorVotes` includes current unclaimed incentives

Submitted by cmichel

The `NoteERC20.getPriorVotes` function is supposed to return the voting strength of an account at a specific block in the past. This should be a static value but it directly includes the *current* unclaimed incentives due to the `getUnclaimedVotes(account)` call.

Users that didn't even have tokens at the time of proposal creation (but are now interested in voting on the proposal), can farm unclaimed incentives and impact the outcome of the proposal.

Adding checkpoints for all unclaimed incentives would be the correct solution but was probably not done because it'd cost too much gas. It also needs to be ensured that incentives cannot be increased through flash-loaning of assets.

T-Woodward (Notional) acknowledged and disagreed with severity:

> It is true that getPriorVotes returns an inaccurate value for the reason you have stated.

> In practice, this issue is not very severe though. Any meaningful attack would require an enormous amount of capital, and could only gain access to at most a relatively small number of votes. The attack would be to provide liquidity at the moment a proposal is introduced, accrue incentives for the duration of the voting

> period, and then vote for them. So the maximum damage that could be done would assume that you have provided all liquidity on the platform (not gonna happen) for the entire voting period (5 days). Given that the highest annual emission rate for NOTE incentives is 20M, this would imply that the maximum amount of incentives earned in the five day period is 277,778 NOTE. This is .277% of the total NOTE supply. So the worst case is still not really a big deal.

> Claimable incentives can't be manipulated via flash loans.

> **Remediation:** We'll remove the `getUnclaimedVotes` function. It is a vestige of an earlier design that we have since moved away from anyway, so there's no real impact on the system or reason not to remove it.

ghoul-sol (judge) commented:

> per sponsor comment, the impact is indeed low

Non-Critical Findings (13)

- [N-01] proposal get defeated even if `forVotes == againstVotes` in `GovernorAlpha.sol`
- [N-02] Erc20 Race condition
- [N-03] Used a fixed or pragma that spans only a single `0.x.*`
- [N-04] Lack of Zero Address Validation
- [N-05] Use pragma abicoder v2
- [N-06] lack of require message
- [N-07] Replacing the assembly `extcodesize` checks for versions `>0.8.1`
- [N-08] Check if address is a contract
- [N-09] Total supply dependency on decimals
- [N-10] unsafe cast from int to uint can lead to incentive abuse
- [N-11] Use of `msg.value` in batch action
- [N-12] Recommend adding a `nonReentrant` modifier to external functions
- [N-13] `initialize()` function of `router.sol` can be reinitialize

Gas Optimizations (8)

- [G-01] Gas optimization on `_INT256_MIN`
- [G-02] Upgrade to at least 0.8.4
- [G-03] Caching length in for loops
- [G-04] `StorageLayoutV1` Gas Optimisations
- [G-05] Unused variables
- [G-06] uint is always >= 0
- [G-07] Cache values that are accessed more than once
- [G-08] Gas optimization: Can put require and variable declaration inside the if statement.

Disclosures