Notional Finance

Date	July 2022
Auditors	George Kobakhidze, Chingiz Mardanov, Sergii Kravchenko

1 Executive Summary

This report presents the results of our engagement with **Notional Finance** to review **Strategy Vaults**.

The review was conducted over 5 weeks, from **07/04/2022** to **08/05/2022** by **George Kobakhidze**, **Chingiz Mardanov** and **Sergii Kravchenko**. A total of 8 person-weeks were spent.

2 Scope

Our review focused on the commit hash `6212a09576ff18eea8b3291667617b724a8e9b1b` for `contracts-v2` and `82eeb5b10285f98553d7b55a94c407ff7e09e818` for `strategy-vaults`. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Notional Finance** team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently with the intended functionality, and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Recommendations

3.1 Unnecessary code handling transfer fee logic

Resolution

Remediated per Notional's team notes in commit by removing unnecessary code and adding an additional require statement

Description

There is code in `VaultConfiguration. transferUnderlyingToVaultDirect` that handles tokens with transfer fees when transferring borrow currencies to the vault:

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L330-L355

```
function transferUnderlyingToVaultDirect(
    VaultConfig memory vaultConfig,
    address transferFrom,
    uint256 depositAmountExternal
) internal returns (uint256) {
    if (depositAmountExternal == 0) return 0;

    Token memory assetToken = TokenHandler.getAssetToken(vaultConfig.borrowCurrencyId);
    Token memory underlyingToken = assetToken.tokenType == TokenType.NonMintable ?
        assetToken :
        TokenHandler.getUnderlyingToken(vaultConfig.borrowCurrencyId);

    address vault = vaultConfig.vault;
    if (underlyingToken.tokenType == TokenType.Ether) {
        require(msg.value == depositAmountExternal, "Invalid ETH");
        // Forward all the ETH to the vault
        GenericToken.transferNativeTokenOut(vault, msg.value);

        return msg.value;
    } else if (underlyingToken.hasTransferFee) {
        // In this case need to check the balance of the vault before and after
        uint256 balanceBefore = underlyingToken.balanceOf(vault);
        GenericToken.safeTransferFrom(underlyingToken.tokenAddress, transferFrom, vault, depositAmountExternal);
        uint256 balanceAfter = underlyingToken.balanceOf(vault);

        return balanceAfter.sub(balanceBefore);
```

However, Notional Strategy Vaults are hardcoded not to allow borrow currencies with transfer fees, either for primary or secondary currencies.

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L164

```
require(!assetToken.hasTransferFee && !underlyingToken.hasTransferFee);
```

contracts-v2/contracts/external/actions/VaultAction.sol:L67

```
require(!assetToken.hasTransferFee && !underlyingToken.hasTransferFee);
```

Recommendation

Consider removing this code to clarify and emphasize the intent of these vaults not to have borrow currencies with transfer fees.

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

4.1 Trust Model

While the Notional Vault system and attached strategy vaults work autonomously on many levels, there are several trust assumptions that need to be kept intact for the system to operate correctly. These include the rest of Notional smart contracts and libraries not in scope of this audit, the Notional team & governance managing the configurations for the whitelisted vaults, and the deployers of the strategy vaults' proxies and implementations.

Other Notional Smart Contracts

Files in scope of this audit interact with almost all of the core Notional system. In fact there are files that are crucial to the vaults logic or were implemented specifically for the use in vaults and yet are not in scope of the audit. Such files include but are not limited to:

- strategy-vaults/contracts/trading/*
- strategy-vaults/contracts/vaults/balancer/*
- strategy-vaults/contracts/vaults/Balancer2TokenVault.sol

Similarly, Notional core contracts such as those that help with operating on the Notional AMM, trading on external DEXs, asset rate calculation, retrieval of system storage, various utils and many other aspects also perform critical operations in the context of Notional Strategy Vaults. Finally, the architecture of the Notional systems of contracts allow them to be upgradeable as well, so the functionality of these interconnected contracts could change.

We approach this audit with an assumption that those modules and the rest of the core Notional system are implemented correctly.

Owner Privileged Actor

Additionally, as the Strategy Vaults functionality effectively allows new arbitrary code access to the core Notional system for borrowing and lending, there is a whitelisting mechanism that only allows authorized vault implementations to interact with the Notional system. The actions of whitelisting as well as managing parameters for the Strategy Vaults are being done by the privileged 'owner' actor. In particular, the owner would be able to whitelist new Strategy Vaults, enable or disable vaults, update their borrow currencies and their limits, change minimum collateral ratios and do other critical functions. While it is likely that the owner would end up being NOTE governance, it is important to note possible attack vectors like compromised privileged access that could jeopardize the security of user funds.

Proxy Implementation Deployer

The Strategy Vaults may also utilize upgradeable proxy and Beacon style deployments. Some of the vault code also performs external delegatecalls to the Notional system, such as those in CrossCurrencyfCashVault that call the Notional Trading Handler to perform swaps against an external DEX. The address of the Notional system through a proxy is set in the constructor of the implementation contract BaseStrategyVault. As a result, the implementation contracts would then need to be deployed by a trusted actor to make sure the address for the Notional Proxy is correct and doesn't point to a malicious contract.

4.2 Additional Notes

Past the audit completion an additional issue was found by the Notional's team during the internal audit. If an account attempts to roll forward with `fCashToBorrow == 0` and `costToRepay > 0` it will short circuit within `_borrowAndTransfer` and cause a cash deficit within Notional. The fix to it can be found here: Commit

5 Contracts V2

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 VaultConfig.setVaultConfig doesn't check all critical arguments Medium

Resolution

Remediated per Notional's team notes in commit by adding the following checks:

- Checks to ensure borrow currency and secondary currencies cannot change once set
- Check to ensure `liquidationRate` does not exceed `minCollateralRatioBPS`

Check for `maxBorrowMarketIndex` was not added. The Notional team will review this parameter on a case-by-case basis as for some vaults borrowing idiosyncratic fCash may not be an issue

Description

The Notional Strategy Vaults need to get whitelisted and have specific Notional parameters set in order to interact with the rest of the Notional system. This is done through `VaultAction.updateVault()` where the `owner` address can provide a `VaultConfigStorage calldata vaultConfig` argument to either whitelist a new vault or change an existing one. While this is to be performed by a trusted privileged actor (the `owner`), and it could be assumed they are careful with their updates, the contracts themselves don't perform enough checks on the validity of the parameters, either in isolation or when compared against the existing vault state. Below are examples of arguments that should be better checked.

`borrowCurrencyId`

The `borrowCurrencyId` parameter gets provided to `TokenHandler.getAssetToken()` and `TokenHandler.getUnderlyingToken()` to retrieve its associated `TokenStorage` object and verify that the currency doesn't have transfer fees.

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L162-L164

```
Token memory assetToken = TokenHandler.getAssetToken(vaultConfig.borrowCurrencyId);
Token memory underlyingToken = TokenHandler.getUnderlyingToken(vaultConfig.borrowCurrencyId);
require(!assetToken.hasTransferFee && !underlyingToken.hasTransferFee);
```

However, these calls retrieve data from the mapping from storage which returns an empty struct for an unassigned currency ID. This would pass the check in the last require statement regarding the transfer fees and would successfully allow to set the currency even if isn't actually registered in Notional. The recommendation would be to check that the returned `TokenStorage` object has data inside of it, perhaps by checking the decimals on the token.

In the event that this is a call to update the configuration on a vault instead of whitelisting a whole new vault, this would also allow to switch the borrow currency without checking that the existing borrow and lending accounting has been cleared. This could cause accounting issues. A check for existing debt before swapping the borrow currency IDs is recommended.

`liquidationRate` and `minCollateralRatioBPS`

To ensure that the system doesn't have bad debt, it employs a liquidation engine that depends on a few parameters, in particular the vault's `liquidationRate` that incentivises liquidators and `minCollateralRatioBPS` that determines when an account can be liquidated. `minCollateralRationBPS+100%` (since the collateral ratio is calculated starting from `0%` not `100%`) would need to be greater than `liquidationRate` (that is calculated from `100%`) or the system could run into problems liquidating small accounts entering the vault. There is an edge case during liquidation where if the account is below the minimum collateral ratio but doesn't have to be liquidated fully, the leftover position from that account would be too small for liquidators to profitably liquidate (due to gas costs) as per another configuration parameter `minAccountBorrowSize`. In this edge case, the system would set the whole account to be liquidated and determine that the liquidator deposit required would be equal to that account's total debt, which would be normally seen as `vaultAccount.fCash`. The liquidator in this case would roughly receive as much value as `vaultAccount.fCash*liquidationRate` denominated in that vault account's `vaultAccount.vaultShares`, which is the existing assets of that vault account. In fact the liquidator gets:

contracts-v2/contracts/external/actions/VaultAccountAction.sol:L274-L283

```
uint256 vaultSharesToLiquidator;
{
    vaultSharesToLiquidator = vaultAccount.tempCashBalance.toUint()
        .mul(vaultConfig.liquidationRate.toUint())
        .mul(vaultAccount.vaultShares)
        .div(vaultShareValue.toUint())
        .div(uint256(Constants.RATE_PRECISION));
}

vaultAccount.vaultShares = vaultAccount.vaultShares.sub(vaultSharesToLiquidator);
```

Where `vaultAccount.tempCashBalance` has the liquidator deposit, which in this case would be the account's debt and equal to `vaultAccount.fCash`. However, since we know that this account is being liquidated, we know that `fCash*(1+minCollateralRationBPS) >= vaultShareValue`. Similarly, assuming the liquidation rate was set incorrectly as defined in the beginning of this section, i.e. `liquidationRate > (1+minCollateralRationBPS)`, we can determine that `fCash*(liquidationRate) > vaultShareValue` as well. Therefore, we will get some number `vaultSharesToLiquidator=X*vaultAccount.vaultShares`, where `X=(vaultAccount.tempCashBalance*vaultConfig.liquidationRate)/(vaultShareValue)` and `X>1`, so the result will be `vaultSharesToLiquidator>vaultAccount.vaultShares`, which will cause a revert once the liquidator shares get subtracted from that vault account's vault share balance. This will cause the account to remain in the system until the account is possibly insolvent , potentially causing bad debt. The recommendation would be to check that the liquidation rate is less than the minimum collateral ratio, of course in the appropriate denomination (i.e. do `minCollateral+1`) and precision.

`maxBorrowMarketIndex`

The current Strategy Vault implementation does not allow for idiosyncratic cash because it causes issues during exits as there are no active markets for the account's maturity. Therefore, the configuration shouldn't be set with `maxBorrowMarketIndex >=3` as that would open up the 1 Year maturity for vault accounts that could cause idiosyncratic fCash. The recommendation would be to add that check.

`secondaryBorrowCurrencies`

Similarly to the `borrowCurrencyId`, there are few checks that actually determine that the `secondaryBorrowCurrencies[]` given are actually registered in Notional. This is, however, more inline with how some vaults are supposed to work as they may have no secondary currencies at all, such as when the `secondaryBorrowCurrencies[]` id is given as `0`. In the event that this is a call to update the configuration on a vault instead of whitelisting a whole new vault, this would also allow to switch the secondary borrow currency without checking that the existing borrow and lending accounting has been cleared. For example, the `VaultAction.updateSecondaryBorrowCapacity()` function could be invoked on the new set of secondary currencies and simply increase the borrow there. This could cause accounting issues. A check for existing debt before swapping the borrow currency IDs is recommended.

5.2 Handle division by 0 Medium

Resolution

Remediated per Notional's team notes in commit by adding the following checks:

- Check to account for div by zero in settle vault account
- Short circuit to ensure debtSharesToRepay is never zero. Divide by zero may still occur but this would signal a critical accounting issue

The Notional team also acknowledged that the contract will revert when `vaultShareValue = 0`. The team decided to not make any changes related to that since liquidation will not accomplish anything for an account with no vault share value.

Description

There are a few places in the code where division by zero may occur but isn't handled.

Examples

If the vault settles at exactly 0 value with 0 remaining strategy token value, there may be an unhandled division by zero trying to divide claims on the settled assets:

contracts-v2/contracts/internal/vaults/VaultAccount.sol:L424-L436

```
int256 settledVaultValue = settlementRate.convertToUnderlying(residualAssetCashBalance)
    .add(totalStrategyTokenValueAtSettlement);

// If the vault is insolvent (meaning residualAssetCashBalance < 0), it is necessarily
// true that totalStrategyTokens == 0 (meaning all tokens were sold in an attempt to
// repay the debt). That means settledVaultValue == residualAssetCashBalance, strategyTokenClaim == 0
// and assetCashClaim == totalAccountValue. Accounts that are still solvent will be paid from the
// reserve, accounts that are insolvent will have a totalAccountValue == 0.
strategyTokenClaim = totalAccountValue.mul(vaultState.totalStrategyTokens.toInt())
    .div(settledVaultValue).toUint();

assetCashClaim = totalAccountValue.mul(residualAssetCashBalance)
    .div(settledVaultValue);
```

If a vault account is entirely insolvent and its `vaultShareValue` is zero, there will be an unhandled division by zero during liquidation:

contracts-v2/contracts/external/actions/VaultAccountAction.sol:L274-L281

```
uint256 vaultSharesToLiquidator;
{
    vaultSharesToLiquidator = vaultAccount.tempCashBalance.toUint()
        .mul(vaultConfig.liquidationRate.toUint())
        .mul(vaultAccount.vaultShares)
        .div(vaultShareValue.toUint())
        .div(uint256(Constants.RATE_PRECISION));
}
```

If a vault account's secondary debt is being repaid when there is none, there will be an unhandled division by zero:

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L661-L666

```
VaultSecondaryBorrowStorage storage balance =
    LibStorage.getVaultSecondaryBorrow()[vaultConfig.vault][maturity][currencyId];
uint256 totalfCashBorrowed = balance.totalfCashBorrowed;
uint256 totalAccountDebtShares = balance.totalAccountDebtShares;

fCashToLend = debtSharesToRepay.mul(totalfCashBorrowed).div(totalAccountDebtShares).toInt();
```

While these cases may be unlikely today, this code could be reutilized in other circumstances later that could cause reverts and even disrupt operations more frequently.

Recommendation

Handle the cases where the denominator could be zero appropriately.

5.3 Increasing a leveraged position in a vault with secondary borrow currency will revert Minor

Resolution

Per Notional team's notes, they have rearranged if statement to ensure that increasing an existing position will work. The proposed solution was skipped as it creates issues with the `_repayDuringRoll` method which will attempt to lend on the current maturity. Commit

Description

From the client's specifications for the strategy vaults, we know that accounts should be able to increase their leveraged positions before maturity. This property will not hold for the vaults that require borrowing a secondary currency to enter a position. When an account opens its position in such vault for the first time, the `VaultAccountSecondaryDebtShareStorage.maturity` is set to the maturity an account has entered. When the account is trying to increase the debt position, an accounts current maturity will be checked, and since it is not set to 0, as in the case where an account enters the vault for the first time, nor it is smaller than the new maturity passed by an account as in case of a rollover, the code will revert.

Examples

contracts-v2/contracts/external/actions/VaultAction.sol:L226-L228

```
if (accountMaturity != 0) {
    // Cannot roll to a shorter term maturity
    require(accountMaturity < maturity);
```

Recommendation

In order to fix this issue, we recommend that `<` is replaced with `<=` so that account can enter the vault maturity the account is already in as well as the future once.

5.4 Secondary Currency debt is not managed by the Notional Controller Minor

Resolution

Remediated per Notional's team notes in commit by adding valuation for secondary borrow within the vault.

Description

Some of the Notional Strategy Vaults may allow for secondary currencies to be borrowed as part of the same strategy. For example, a strategy may allow for USDC to be its primary borrow currency as well as have ETH as its secondary borrow currency.

In order to enter the vault, a user would have to deposit `depositAmountExternal` of the primary borrow currency when calling `VaultAccountAction.enterVault()`. This would allow the user to borrow with leverage, as long as the `vaultConfig.checkCollateralRatio()` check on that account succeeds, which is based on the initial deposit and borrow currency amounts. This collateral ratio check is then performed throughout that user account's lifecycle in that vault, such as when they try to roll their maturity, or when liquidators try to perform collateral checks to ensure there is no bad debt.

However, in the event that the vault has a secondary borrow currency as well, that additional secondary debt is not calculated as part of the `checkCollateralRatio()` check. The only debt that is being considered is the `vaultAccount.fCash` that corresponds to the primary borrow currency debt:

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L313-L319

```
function checkCollateralRatio(
    VaultConfig memory vaultConfig,
    VaultState memory vaultState,
    VaultAccount memory vaultAccount
) internal view {
    (int256 collateralRatio, /* */) = calculateCollateralRatio(
        vaultConfig, vaultState, vaultAccount.account, vaultAccount.vaultShares, vaultAccount.fCash
```

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L278-L292

```
function calculateCollateralRatio(
    VaultConfig memory vaultConfig,
    VaultState memory vaultState,
    address account,
    uint256 vaultShares,
    int256 fCash
) internal view returns (int256 collateralRatio, int256 vaultShareValue) {
    vaultShareValue = vaultState.getCashValueOfShare(vaultConfig, account, vaultShares);

    // We do not discount fCash to present value so that we do not introduce interest
    // rate risk in this calculation. The economic benefit of discounting will be very
    // minor relative to the added complexity of accounting for interest rate risk.

    // Convert fCash to a positive amount of asset cash
    int256 debtOutstanding = vaultConfig.assetRate.convertFromUnderlying(fCash.neg());
```

Whereas the value of strategy tokens that belong to that user account are being calculated by calling `IStrategyVault(vault).convertStrategyToUnderlying()` on the associated strategy vault:

contracts-v2/contracts/internal/vaults/VaultState.sol:L314-L324

```
function getCashValueOfShare(
    VaultState memory vaultState,
    VaultConfig memory vaultConfig,
    address account,
    uint256 vaultShares
) internal view returns (int256 assetCashValue) {
    if (vaultShares == 0) return 0;
    (uint256 assetCash, uint256 strategyTokens) = getPoolShare(vaultState, vaultShares);
    int256 underlyingInternalStrategyTokenValue = _getStrategyTokenValueUnderlyingInternal(
        vaultConfig.borrowCurrencyId, vaultConfig.vault, account, strategyTokens, vaultState.maturity
    );
```

contracts-v2/contracts/internal/vaults/VaultState.sol:L296-L311

```
function _getStrategyTokenValueUnderlyingInternal(
    uint16 currencyId,
    address vault,
    address account,
    uint256 strategyTokens,
    uint256 maturity
) private view returns (int256) {
    Token memory token = TokenHandler.getUnderlyingToken(currencyId);
    // This will be true if the the token is "NonMintable" meaning that it does not have
    // an underlying token, only an asset token
    if (token.decimals == 0) token = TokenHandler.getAssetToken(currencyId);

    return token.convertToInternal(
        IStrategyVault(vault).convertStrategyToUnderlying(account, strategyTokens, maturity)
    );
}
```

From conversations with the Notional team, it is assumed that this call returns the strategy token value subtracted against the secondary currencies debt, as is the case in the `Balancer2TokenVault` for example. In other words, when collateral ratio checks are performed, those strategy vaults that utilize secondary currency borrows would need to calculate the value of strategy tokens already accounting for any secondary debt. However, this is a dependency for a critical piece of the Notional controller's strategy vaults collateral checks.

Therefore, even though the strategy vaults' code and logic would be vetted before their whitelisting into the Notional system, they would still remain an external dependency with relatively arbitrary code responsible for the liquidation infrastructure that could lead to bad debt or incorrect liquidations if the vaults give inaccurate information, and thus potential loss of funds.

Recommendation

Specific strategy vault implementations using secondary borrows were not in scope of this audit. However, since the core Notional Vault system was, and it includes secondary borrow currency functionality, from the point of view of the larger Notional system it is recommended to include secondary debt checks within the Notional controller contract to reduce external dependency on the strategy vaults' logic.

5.5 Vaults are unable to borrow single secondary currency Minor

Resolution
Remediated per Notional's team notes.

Description

As was previously mentioned some strategies require borrowing one or two secondary currencies. All secondary currencies have to be whitelisted in the `VaultConfig.secondaryBorrowCurrencies`. Borrow operation on secondary currencies is performed in the `borrowSecondaryCurrencyToVault(...)` function. Due to a `require` statement in that function, vaults will only be able to borrow secondary currencies if both of the currencies are whitelisted in `VaultConfig.secondaryBorrowCurrencies`. Considering that many strategies will have just one secondary currency, this will prevent those strategies from borrowing any secondary assets.

Examples

contracts-v2/contracts/external/actions/VaultAction.sol:L214

```
require(currencies[0] != 0 && currencies[1] != 0);
```

Recommendation

We suggest that the `&&` operator is replaced by the `||` operator. Ideally, an additional check will be performed that will ensure that values in argument arrays `fCashToBorrow`, `maxBorrowRate`, and `minRollLendRate` are passed under the same index as the whitelisted currencies in `VaultConfig.secondaryBorrowCurrencies`.

contracts-v2/contracts/external/actions/VaultAction.sol:L202-L208

```
function borrowSecondaryCurrencyToVault(
    address account,
    uint256 maturity,
    uint256[2] calldata fCashToBorrow,
    uint32[2] calldata maxBorrowRate,
    uint32[2] calldata minRollLendRate
) external override returns (uint256[2] memory underlyingTokensTransferred) {
```

5.6 An account roll may be impossible if the vault is already at the maximum borrow capacity. Minor

Resolution

Remediated per Notional's team notes in commit by adding the ability for accounts to deposit during a roll vault position call to offset any additional cost that would put them over the maximum borrow capacity.

Description

One of the actions allowed in Notional Strategy Vaults is to roll an account's maturity to a later one by borrowing from a later maturity and repaying that into the debt of the earlier maturity.

However, this could cause an issue if the vault is at maximum capacity at the time of the roll. When an account performs this type of roll, the new borrow would have to be more than the existing debt simply because it has to at least cover the existing debt and pay for the borrow fees that get added on every new borrow. Since the whole vault was already at max borrow capacity before with the old, smaller borrow, this process would revert at the end after the new borrow as well once the process gets to `VaultAccount.updateAccountfCash` and `VaultConfiguration.updateUsedBorrowCapacity`:

contracts-v2/contracts/internal/vaults/VaultConfiguration.sol:L243-L257

```
function updateUsedBorrowCapacity(
    address vault,
    uint16 currencyId,
    int256 netfCash
) internal returns (int256 totalUsedBorrowCapacity) {
    VaultBorrowCapacityStorage storage cap = LibStorage.getVaultBorrowCapacity()[vault][currencyId];

    // Update the total used borrow capacity, when borrowing this number will increase (netfCash < 0),
    // when lending this number will decrease (netfCash > 0).
    totalUsedBorrowCapacity = int256(uint256(cap.totalUsedBorrowCapacity)).sub(netfCash);
    if (netfCash < 0) {
        // Always allow lending to reduce the total used borrow capacity to satisfy the case when the max borrow
        // capacity has been reduced by governance below the totalUsedBorrowCapacity. When borrowing, it cannot
        // go past the limit.
        require(totalUsedBorrowCapacity <= int256(uint256(cap.maxBorrowCapacity)), "Max Capacity");
```

The result is that users won't able to roll while the vault is at max capacity. However, users may exit some part of their position to reduce their borrow, thereby reducing the overall vault borrow capacity, and then could execute the roll. A bigger problem would occur if the vault configuration got updated to massively reduce the borrow capacity, which would force users to exit their position more significantly with likely a much smaller chance at being able to roll.

Recommendation

Document this case so that users can realise that rolling may not always be an option. Perhaps consider adding ways where users can pay a small deposit, like on `enterVault`, to offset the additional difference in borrows and pay for fees so they can remain with essentially the same size position within Notional.

5.7 Rollover might introduce economically impractical deposits of dust into a strategy Minor

Resolution

Acknowledged with a note from the Notional's team:

"This is true, however, vaults with secondary borrows may need to execute logic in order to roll positions forward. We will opt to not do any handling for dust amounts on the vault controller side and allow each vault to set its own dust thresholds."

Description

During the rollover of the strategy position into a longer maturity, several things happen:

- Funds are borrowed from the longer maturity to pay off the debt and fees of the current maturity.
- Strategy tokens that are associated with the current maturity are moved to the new maturity.
- Any additional funds provided by the account are deposited into the strategy into a new longer maturity.

In reality, due to the AMM nature of the protocol, the funds borrowed from the new maturity could exceed the debt the account has in the current maturity, resulting in a non-zero `vaultAccount.tempCashBalance`. In that case, those funds will be deposited into the strategy. That would happen even if there are no external funds supplied by the account for the deposit.

It is possible that the dust in the temporary account balance will not cover the gas cost of triggering a full deposit call of the strategy.

Examples

contracts-v2/contracts/internal/vaults/VaultState.sol:L244-L246

```
uint256 strategyTokensMinted = vaultConfig.deposit(
    vaultAccount.account, vaultAccount.tempCashBalance, vaultState.maturity, additionalUnderlyingExternal, vaultData
);
```

Recommendation

We suggest that additional checks are introduced that would check that on rollover `vaultAccount.tempCashBalance + additionalUnderlyingExternal > 0` or larger than a certain threshold like `minAccountBorrowSize` for example.

5.8 Significantly undercollateralized accounts will revert on liquidation Minor

Resolution

Remediated per Notional's team notes in commit by updating the calculations within `calculateDeleverageAmount`

Description

The Notional Strategy Vaults utilise collateral to allow leveraged borrowing as long as the account passes the `checkCollateralRatio` check that ensures the overall account value is at least `minCollateralRatio` greater than its debts. If the account doesn't have sufficient collateral, it goes through a liquidation process where some of the collateral is sold to liquidators for the account's borrowed currency in attempt to improve the collateral ratio. However, if the account is severely undercollateralised, the entire account position is liquidated and given over to the liquidator:

contracts-v2/contracts/internal/vaults/VaultAccount.sol:L282-L289

```
int256 depositRatio = maxLiquidatorDepositAssetCash.mul(vaultConfig.liquidationRate).div(vaultShareValue);
// Use equal to so we catch potential off by one issues, the deposit amount calculated inside the if statement
// below will round the maxLiquidatorDepositAssetCash down
if (depositRatio >= Constants.RATE_PRECISION) {
    maxLiquidatorDepositAssetCash = vaultShareValue.divInRatePrecision(vaultConfig.liquidationRate);
    // Set this to true to ensure that the account gets fully liquidated
    mustLiquidateFullAmount = true;
```

Here, the liquidator will need to deposit exactly `maxLiquidatorDepositAssetCash=vaultShareValue/liquidationRate` in order to get all of account's assets, i.e. all of `vaultShareValue` in the form of `vaultAccount.vaultShares`. In fact, later this deposit will be set in `vaultAccount.tempCashBalance`:

contracts-v2/contracts/external/actions/VaultAccountAction.sol:L361-L380

```
int256 maxLiquidatorDepositExternal = assetToken.convertToExternal(maxLiquidatorDepositAssetCash);

// NOTE: deposit amount external is always positive in this method
if (depositAmountExternal < maxLiquidatorDepositExternal) {
    // If this flag is set, the liquidator must deposit more cash in order to liquidate the account
    // down to a zero fCash balance because it will fall under the minimum borrowing limit.
    require(!mustLiquidateFull, "Must Liquidate All Debt");
} else {
    // In the other case, limit the deposited amount to the maximum
    depositAmountExternal = maxLiquidatorDepositExternal;
}

// Transfers the amount of asset tokens into Notional and credit it to the account's temp cash balance
int256 assetAmountExternalTransferred = assetToken.transfer(
    liquidator, vaultConfig.borrowCurrencyId, depositAmountExternal
);

vaultAccount.tempCashBalance = vaultAccount.tempCashBalance.add(
    assetToken.convertToInternal(assetAmountExternalTransferred)
);
```

Then the liquidator will get:

contracts-v2/contracts/external/actions/VaultAccountAction.sol:L274-L281

```
uint256 vaultSharesToLiquidator;
{
    vaultSharesToLiquidator = vaultAccount.tempCashBalance.toUint()
        .mul(vaultConfig.liquidationRate.toUint())
        .mul(vaultAccount.vaultShares)
        .div(vaultShareValue.toUint())
        .div(uint256(Constants.RATE_PRECISION));
}
```

And if (except for precision and conversions) `vaultAccount.tempCashBalance=maxLiquidatorDepositAssetCash=vaultShareValue/liquidationRate`, then `vaultSharesToLiquidator = (vaultAccount.tempCashBalance * liquidationRate * vaultAccount.vaultShares) / (vaultShareValue)` becomes `vaultSharesToLiquidator = ((vaultShareValue/liquidationRate)* liquidationRate * vaultAccount.vaultShares) / (vaultShareValue) = vaultAccount.vaultShares`

In other words, the liquidator needed to deposit exactly `vaultShareValue/liquidationRate` to get all `vaultAccount.vaultShares`. However, the liquidator deposit (what would be in `vaultAccount.tempCashBalance`) needs to cover all of that account's debt, i.e. `vaultAccount.fCash`. At the end of the liquidation process, the vault account has its fCash and tempCash balances updated:

contracts-v2/contracts/external/actions/VaultAccountAction.sol:L289-L290

```
int256 fCashToReduce = vaultConfig.assetRate.convertToUnderlying(vaultAccount.tempCashBalance);
vaultAccount.updateAccountfCash(vaultConfig, vaultState, fCashToReduce, vaultAccount.tempCashBalance.neg());
```

contracts-v2/contracts/internal/vaults/VaultAccount.sol:L77-L88

```
function updateAccountfCash(
    VaultAccount memory vaultAccount,
    VaultConfig memory vaultConfig,
    VaultState memory vaultState,
    int256 netfCash,
    int256 netAssetCash
) internal {
    vaultAccount.tempCashBalance = vaultAccount.tempCashBalance.add(netAssetCash);

    // Update fCash state on the account and the vault
    vaultAccount.fCash = vaultAccount.fCash.add(netfCash);
    require(vaultAccount.fCash <= 0);
```

While the `vaultAccount.tempCashBalance` gets cleared to 0, the `vaultAccount.fCash` amount only gets to `vaultAccount.fCash = vaultAccount.fCash.add(netfCash)`, and `netfCash=fCashToReduce = vaultConfig.assetRate.convertToUnderlying(vaultAccount.tempCashBalance)`, which, based on the constraints above essentially becomes:

```
vaultAccount.fCash=vaultAccount.fCash+vaultConfig.assetRate.convertToUnderlying(assetToken.convertToExternal(vaultShareValue/vaultConfig.liquidationRate))
```

However, later this account is set on storage, and, considering it is going through 100% liquidation, the account will necessarily be below minimum borrow size and will need to be at `vaultAccount.fCash==0`.

contracts-v2/contracts/internal/vaults/VaultAccount.sol:L52-L62

```
function setVaultAccount(VaultAccount memory vaultAccount, VaultConfig memory vaultConfig) internal {
    mapping(address => mapping(address => VaultAccountStorage)) storage store = LibStorage
        .getVaultAccount();
    VaultAccountStorage storage s = store[vaultAccount.account][vaultConfig.vault];

    // The temporary cash balance must be cleared to zero by the end of the transaction
    require(vaultAccount.tempCashBalance == 0); // dev: cash balance not cleared
    // An account must maintain a minimum borrow size in order to enter the vault. If the account
    // wants to exit under the minimum borrow size it must fully exit so that we do not have dust
    // accounts that become insolvent.
    require(vaultAccount.fCash == 0 || vaultConfig.minAccountBorrowSize <= vaultAccount.fCash.neg(), "Min Borrow");
```

The case where vaultAccount.fCash>0 is taken care of by taking any extra repaid value and assigning it to the protocol, zeroing out the account's balances:

contracts-v2/contracts/external/actions/VaultAccountAction.sol:L293

```
if (vaultAccount.fCash > 0) vaultAccount.fCash = 0;
```

The case where `vaultAccount.fCash < 0` is however not addressed, and instead the process will revert. This will occur whenever the vaultShareValue discounted with the liquidation rate is less than the fCash debt after all the conversions between external and underlying accounting. So, whenever the below is true, the account will not be liquidate-able. `fCash>vaultShareValue/liquidationRate`

This is an issue because the account is still technically solvent even though it is undercollateralized, but the current implementation would simply revert until the account is entirely insolvent (still without liquidation options) or its balances are restored enough to be liquidated fully.

Consider implementing a dynamic liquidation rate that becomes smaller the closer the account is to insolvency, thereby encouraging liquidators to promptly liquidate the accounts.

6 Strategy Vaults

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Strategy vault swaps can be frontrun Minor

Resolution
Acknowledged with a note from the Notional's team: "This is a large part of the diligence process for writing strategies"

Description

Some strategy vaults utilize borrowing one currency, swapping it for another, and then using the new currency somewhere to generate yield. For example, the CrossCurrencyfCash strategy vault could borrow USDC, swap it for DAI, and then deposit that DAI back into Notional if the DAI lending interest rates are greater than USDC borrowing interest rates. However, during vault settlement the assets would need to be swapped back into the original borrow currency.

Since these vaults control the borrowed assets that go only into white-listed strategies, the Notional system allows users to borrow multiples of their posted collateral and claim the yield from a much larger position. As a result, these strategy vaults would likely have significant funds being borrowed and managed into these strategies.

However, as mentioned above, these strategies usually utilize a trading mechanism to swap borrowed currencies into whatever is required by the strategy, and these trades may be quite large. In fact, the `BaseStrategyVault` implementation contains functions that interact with Notional's trading module to assist with those swaps:

strategy-vaults/contracts/vaults/BaseStrategyVault.sol:L100-L127

```
/// @notice Can be used to delegate call to the TradingModule's implementation in order to execute
/// a trade.
function _executeTrade(
    uint16 dexId,
    Trade memory trade
) internal returns (uint256 amountSold, uint256 amountBought) {
    (bool success, bytes memory result) = nProxy(payable(address(TRADING_MODULE))).getImplementation()
        .delegatecall(abi.encodeWithSelector(ITradingModule.executeTrade.selector, dexId, trade));
    require(success);
    (amountSold, amountBought) = abi.decode(result, (uint256, uint256));
}

/// @notice Can be used to delegate call to the TradingModule's implementation in order to execute
/// a trade.
function _executeTradeWithDynamicSlippage(
    uint16 dexId,
    Trade memory trade,
    uint32 dynamicSlippageLimit
) internal returns (uint256 amountSold, uint256 amountBought) {
    (bool success, bytes memory result) = nProxy(payable(address(TRADING_MODULE))).getImplementation()
        .delegatecall(abi.encodeWithSelector(
            ITradingModule.executeTradeWithDynamicSlippage.selector,
            dexId, trade, dynamicSlippageLimit
        )
    );
    require(success);
    (amountSold, amountBought) = abi.decode(result, (uint256, uint256));
}
```

Although some strategies may manage stablecoin <-> stablecoin swaps that typically would incur low slippage, large size trades could still suffer from low on-chain liquidity and end up getting frontrun and "sandwiched" by MEV bots or other actors, thereby extracting maximum amount from the strategy vault swaps as slippage permits. This could be especially significant during vaults' settlements, that can be initiated by anyone, as lending currencies may be swapped in large batches and not do it on a per-account basis. For example with the CrossCurrencyfCash vault, it can only enter settlement if all strategy tokens (lending currency in this case) are gone and swapped back into the borrow currency:

strategy-vaults/contracts/vaults/CrossCurrencyfCashVault.sol:L141-L143

```
if (vaultState.totalStrategyTokens == 0) {
    NOTIONAL.settleVault(address(this), maturity);
}
```

As a result, in addition to the risk of stablecoins' getting off-peg, unfavorable market liquidity conditions and arbitrage-seeking actors could eat into the profits generated by this strategy as per the maximum allowed slippage. However, during settlement the strategy vaults don't have the luxury of waiting for the right conditions to perform the trade as the borrows need to repaid at their maturities.

So, the profitability of the vaults, and therefore users, could suffer due to potential low market liquidity allowing high slippage and risks of being frontrun with the chosen strategy vaults' currencies.

Recommendation

Ensure that the currencies chosen to generate yield in the strategy vaults have sufficient market liquidity on exchanges allowing for low slippage swaps.

6.2 Cross currency strategy should not have same lend and borrow currencies Minor

Description

Cross currency strategy currently takes lend and borrow currencies as the initialization arguments. Due to the way strategy and `TradingModule` are implemented, the strategy will not operate correctly if lend and borrow currencies are the same. Despite those arguments being passed exclusively by the Notional team, there is still a possibility of incorrect arguments being used.

Examples

strategy-vaults/contracts/vaults/CrossCurrencyfCashVault.sol:L77-L82

```
function initialize(
    string memory name_,
    uint16 borrowCurrencyId_,
    uint16 lendCurrencyId_,
    uint64 settlementSlippageLimit_
) external initializer {
```

Recommendation

We suggest adding a `require` check in the initialization function of the `CrossCurrencyfCashVault.sol` that will ensure that lend and borrow currencies are different.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts-v2/contracts/internal/vaults/VaultAccount.sol	63bb4fb2afb562ff8c9a9a11568eb5714f66a3de
contracts-v2/contracts/internal/vaults/VaultConfiguration.sol	7d6e51045f77a56ba8a8a755d5be1b883f36953f
contracts-v2/contracts/internal/vaults/VaultState.sol	48154f31af67fd8619462546086d6f8ab48cade8
contracts-v2/contracts/external/actions/VaultAccountAction.sol	604fdcfbdc888e628784db02975fc168d2e9c12d
contracts-v2/contracts/external/actions/VaultAction.sol	ae26284c351c3f341d39da392b66c0ed5b00a722
strategy-vaults/contracts/vaults/BaseStrategyVault.sol	b512776162107362b69272ff03a587055c67351e
strategy-vaults/contracts/vaults/CrossCurrencyfCashVault.sol	1c128207d38fe662e6eafb376280f0dc154ff4dd

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.