OpenZeppelin | security

Wido Comet Collateral Swap Contracts

Compound

October 16, 2023

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-09-20 To 2023-09-29
Languages	Solidity

Total Issues	11 (5 resolved, 3 partially resolved)
Critical Severity Issues	2 (2 resolved)
High Severity Issues	1 (0 resolved, 1 partially resolved)
Medium Severity Issues	2 (0 resolved, 1 partially resolved)
Low Severity Issues	3 (1 resolved, 1 partially resolved)
Notes & Additional Information	3 (2 resolved)

Scope

We audited the `widolabs/wido-contracts` repository at commit d96ef2d.

In scope were the following contracts:

```
contracts
├── compound
│   ├── WidoCollateralSwap_Aave.sol
│   └── WidoCollateralSwap_ERC3156.sol
├── libraries
│   └── LibCollateralSwap.sol
└── interfaces
    ├── IComet.sol
    ├── IERC3156.sol
    └── IWidoCollateralSwap.sol
```

System Overview

Wido is a set of smart contracts and APIs that enable swaps between any token, focused on non-liquid tokens like farms, vaults or pools. The project under review in this report adds a new functionality to the Wido set of contracts. The new contracts offer users the ability to replace their collateral assets on Compound V3 (hereafter referred to as "Comet") with different ones and swap their now withdrawn, original assets using the Wido Router contracts. To provide this service, it utilizes ERC-3156-compliant flash loan providers and has a specific implementation to utilize Aave's flash loan services. This results in greater integration between the two DeFi protocols.

Security Model and Trust Assumptions

The greater space of Wido contracts (e.g. the router and token manager) was not requested for review and remained out of scope for this audit. Therefore, we did not verify their correct behavior and instead assumed they behaved as intended.

During the process of using the Wido Comet collateral swapper, the user temporarily grants full permission for the swapper contract to manage the funds of the user on a Comet instance. This permission is revoked immediately after a call to the specified Comet instance's `withdrawFrom` method. So rather than users granting the swapper contract permission via `approve`, the user will call the swapper contract having signed two EIP-712-compliant digests (one for granting and revoking access) that the swapper will then use to call Comet's `allowBySig` method.

Critical Severity

C-01 Unexpected Entry Point Leads to User Impersonation

Both `WidoCollateralSwap_Aave.executeOperation` and `WidoCollateralSwap_ERC3156.onFlashLoan` check if the message sender is an authorized flash loan provider but neither check to see if the loan originated from the swap contract itself. This is important because these flash loan callbacks are expecting the respective `swapCollateral` function to have encoded the correct `msg.sender` within the `params` or `data` parameter. By allowing anyone to initiate a flash loan from the correct provider and then call into the `executeOperation`/`onFlashLoan` functions, a malicious user is able to impersonate another by providing whatever bytes they want for those sensitive parameters. An example would be front-running a user who has exposed their signatures to force them to swap to different assets. Indeed, arbitrageurs would be incentivized to perform this attack.

Consider adding a `require` statement to enforce that `initiator == address(this)`.

***Update**: Resolved in [pull request #35](#) at commit [f6aa4b7](#).*

C-02 Unrestricted Execution with `_swap` Function

The `_swap` function is designed to allow users to exchange collateral removed from a Comet market, ensuring there are enough assets to repay the flash loan. However, the swap process, executed on [line 85](#) of `LibCollateralSwap.sol`, heavily relies on user inputs. This dependence creates a vulnerability, where an attacker might inject malicious inputs into the contract, thereby potentially initiating unrestricted calls to any other smart contract within the blockchain. Combined with other issues found (see C01, H01, M02, and L03) this leads to scenarios where calling users can have all of their Comet positions stolen.

Below is an illustration of this attack, demonstrating how an attacker can exploit the signatures provided by any user, along with the contract's manager-level access over the user's Comet funds, to steal the user's collateral:

- Alice holds a debt position in Comet and she wishes to exchange a portion of her collateral for a new one
- An attacker sees Alice's public transaction and her signatures, and makes a new transaction. They keep Alice's signatures and pass in a malicious contract's address as the Comet address to gain control whenever Comet is called from the protocol
- The attacker front-runs Alice's transaction
- The attacker takes a flash loan for the contract, triggering `executeOperation`
- The attacker gains control on line 121 of `LibCollateralSwap.sol`, where the contract has manager access over the user's Comet funds
- Attacker re-enters the collateral swap function, this time aiming for line 85 of `LibCollateralSwap.sol` providing malicious inputs to steal all of Alice's unlocked funds. They specify an official Comet address as `swap.router` and `transferAssetFrom(address,address,address,uint256)` as `swap.callData` to steal the user's unlocked collateral.

As a side note, the attacker can go further and unlock all of the user's Comet collateral with a flash loan before the whole process, to effectively steal all of their collateral, close the debt position and come out with a profit since the position is over-collateralized.

Consider sanitizing the swap input to ensure users are calling the correct swap contract. Also, you can utilize OpenZeppelin's reenterancy guard to ensure the flow is executed once at a time.

***Update**: Resolved in pull request #38 at commit e093542.*

High Severity

H-01 Comet Address Is Not Confirmed

`performCollateralSwap` allows a user to specify an address that will be used as a Comet implementation. The address, however, is not confirmed to be an official Comet market, allowing a user to run any arbitrary contract in its place. An attacker could input a malicious contract address or lead others to do so and gain control over the transaction at critical points of the execution to steal from users. See line 121 of the `LibCollateralSwap.sol` for an example of where execution control would be transferred.

Consider storing valid Comet addresses within the contract or confirming Comet addresses by calling the Comet Configurator's `getConfiguration` method.

***Update**: Partially resolved in pull request #36 at commit e933cdc. While storage space for a comet market address has been added, consider verifying the input address during construction via Comet's Configurator.*

Medium Severity

M-01 Lack of swap sanitization could lead to user's collateral being stolen

During the collateral swap process, the swap function is supposed to exchange user's collateral to make up enough funds and repay the flash loan. However, the function never checks that the swap's output token is the same as the flash loan borrowed asset, and the contract only re-invests the surplus amount of the borrowed asset back into the user's Comet. In case users make the mistake of calling `swapCollateral` by swapping into the the wrong asset (other than the flash loan or deposit asset), the tokens can be stuck in the contract and malicious users can steal the tokens.

For example:

- Alice sends a transaction that takes flash loan in `USDC`, but by mistake swaps the removed collateral into `USDT`. This should cause the transaction to fail, since there is not enough funds to repay the flash loan.
- An attacker sees the transaction, and sends enough `USDC` to the protocol just before Alice's transaction to make her transaction pass.
- After Alice's transaction passes, there are some amount of `USDT` left in the protocol from Alice's swap
- The attacker now creates a vacuous transaction, specifying `USDT` as their flash loan asset. Since line 65 of `LibCollateralSwap.sol` simply transfers all of the borrowed asset in the contract to the caller, the attacker will get all of the `USDT` that Alice wanted to swap for (less the value of the loan required to swap).

Consider checking the swap destination token type is the same as the flash loan token type during the swap process to mitigate this attack vector.

***Update**: Partially resolved in pull request #37 at commit d66b097. While the code checks if the swap has utilized the flash loan asset, the larger Wido swap contracts were out of scope and so the same attack vector may remain should the swap attempt to swap for multiple tokens.*

M-02 Unnecessary fee structuring

During the collateral swap the flash loan fee is first subtracted from the flash loan amount which is then deposited into Comet. This is to ensure that the protocol has enough to pay the flash loan fee. But the swap process itself also relies on the user to pick a large enough amount to withdraw so as to cover the flash loan amount with a token swap. Since the contract checks for the required amount to be repaid, removing the fee first only adds to the complexity of the calculation with which the user has to decide the withdrawal amount. This could be a difficult task given the process involves unpredictable market exchange rate and slippage caused by available liquidity between the two swapping collaterals.

Consider removing the fee deduction to use the full flash loan for the Comet collateral deposit. Further consider offering helper functions to assist with the calculation of the appropriate final collateral amount.

***Update**: Acknowledged, not resolved. The Wido team stated:*

> *We have an off-chain SDK to assist with the calculation.*

Low Severity

L-01 Missing Docstrings

Throughout the codebase, there are several parts that do not have docstrings. For example:

- Line 14 of `WidoCollateralSwap_Aave.sol`
- Line 12 of `WidoCollateralSwap_ERC3156.sol`
- Line 9 of `LibCollateralSwap.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well (even copied interfaces). When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #34 at commit 9a24c01.*

L-02 Exposed signatures can be lead to unanticipated permissions

To call `swapCollateral`, a user must provide signatures for allowing the contract permission to move the user's funds on Comet. Exposing this signature means that a front-runner can call Comet's `allowBySig` with the signature ahead of the `swapCollateral` transaction. The swap transaction will fail because of a bad nonce and the swap contract will have complete permission for the users' account. This could be entirely unexpected and lead to a user unaware of permissions for their own account.

Consider requiring users to call Comet's `allow` function for the swap contract prior to interacting with it.

Update: *Acknowledged, not resolved. The Wido team stated:*

> *To make the UX simpler for the user, we are keeping the signatures.*

L-03 Failing Tests and Public `env` Variables

Multiple test cases are failing in the provided repo. Tests are an essential part of the development and security cycle. Consider fixing test cases and ensuring that all tests pass and that the coverage is high.

There is also a `.env` file included in the repo which contains information that might be sensitive, such as API keys. Consider removing these from the public repository and providing a `.env.example` as the template instead.

Update: *Partially resolved in pull request #33 at commit 371963b. The testing suite was updated but we were unable to run them successfully.*

Notes & Additional Information

N-01 Aave Pool Address is Immutable

The `WidoCollateralSwap_Aave` contract utilizes a `POOL` variable to call Aave flash loans. This variable is marked as immutable, meaning that when the Aave pool address changes, this entire contract will need to be redeployed. Consider making this variable mutable so it can be updated in the future.

***Update**: Acknowledged, not resolved. The Wido team stated:*

> *We are okay with deploying a new contract when the Aave Pool address changes.*

N-02 Unused Imports

Throughout the codebase, there are imports that are unused and could be removed. For instance:

- Import `IComet` of `WidoCollateralSwap_Aave.sol`
- Import `IComet` of `WidoCollateralSwap_ERC3156.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

***Update**: Resolved in pull request #32 at commit f34c83b.*

N-03 Using `int`/`uint` instead of `int256`/`uint256`

Within `IComet.sol`, there are multiple instances where `int`/`uint` is used as opposed to `int256`/`uint256`. For instance:

- On line 6 of `IComet.sol`
- On line 8 of `IComet.sol`

In favor of explicitness, consider replacing all instances of `int`/`uint` with `int256`/ `uint256`.

Update: *Resolved in [pull request #32](#) at commit [f34c83b](#).*

Conclusion

Two critical issues and one high were reported along with multiple minor issues. While the Wido team has resolved all major issues we've reported, the Wido router and token manager contracts were out of scope and thus are assumed to be working properly.

It must be noted that the reliance on Wido's out-of-scope contracts have a major impact on the security of this integration which includes a user-specified, non-sanitized call to the Wido Router contract. Compound users are cautioned to proceed at their own risk as this feature is dependent on the security of the Wido protocol which OpenZeppelin has not reviewed.