1inch Audit Report

Date: November 5, 2020

The following report was written by Scott Bigelow, requested by the 1inch.exchange team.

This audit consists of both automated and manual review of the 1inch Solidity code, available here:

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/OneInchExchange.sol

Including relevant contracts `OneInchExchange.sol`, `OneInchCaller.sol`, `OneInchFlags.sol` at git commit `f9de98eec2d78797beb031202c79eacdcddc719c`.

`OneInchExchange.sol` from git commit `f9de98eec2d78797beb031202c79eacdcddc719c` corresponds to mainnet deployed address `0x11111125434b319222cdbf8c261674adb56f3ae`

This review of the 1inch system was done on behalf of Scott Bigelow Corp, LLC, as an attempt to identify concerns and potential flaws in the Solidity code. This review was done to the best of my ability, is written for the benefit of Solidity developers, and is not a guarantee that all flaws were discovered in this process.

Overview

`OneInchExchange` is a wrapper contract allowing a user to provide a list of arbitrary calls to execute sequentially, which are executed in the following order:

1.) Optionally, retrieve a user's tokens
2.) Execute a series of arbitrary calls (using unprivileged `OneInchCaller.sol`)
3.) Deliver "swapped" token to requested location

While 1inch functions as a DEX "aggregator", that functionality comes from a complex set of off-chain operations (from 1inch UI & backend tasks), combined with this simple iteration of arbitrary calls that take place on-chain (step #2). The 1inch UI/backend that generates these arbitrary calls is not in scope for this audit; this report exclusively focuses on the on-chain logic.

Keeping the on-chain portion simple, as is the case here, is a great design choice, but it also means that I cannot check what every future arbitrary call will do. I can, however, attempt to

ensure that the higher-level token input and token output requirements are met, and that approvals to the correct 1inch contract do not result in lost assets.

Summary of findings

The overall code quality of the project is good and the function interfaces and general logic is well-structured and easy to follow. Unit tests are present, but could cover more malicious cases.

During the course of the review, the git commit was switched from `c998b0ffb71dba509047d4595289ddefe63180a8` to `049b36363b2d0a0d1f2318862e734ec9ac2b630c`, after I reported the one critical finding below ("facadeCall"). The commit completely removed the vulnerable function, fixing the critical finding.

The commit was again switched to `43b4c6d1593ece0350edf793d06a20e214c87c50` after I made an architectural suggestion (outlined below) that the 1inch team adopted. The resulting code from this refactor is significantly smaller with a massively reduced attack surface and reduces the likelihood of a user accidentally approving the wrong contract.

Many initial security findings were no longer applicable after adopting the suggested architectural change.

Architecture design suggestion

What follows is the justification for a change to architectural design that was adopted by the 1inch team and is included in the most recent commit being reviewed.
The architecture of the system involves two main components;
1.) "OneInchExchange" contract, which executes the call batching and minimum-return logic
2.) "TokenSpender", which receives the ERC20 approvals for a user's tokens, retrieving these tokens when requested by OneInchExchange contract.

It is extremely important to ensure that not only is the logic around direct interaction between OneInchExchange and TokenSpender correct (only taking from msg.sender, only in the quantities specified), but also that any arbitrary call made from OneInchExchange does not allow interaction with TokenSpender. While I believe that both direct and arbitrary interactions are handled correctly, the overall architecture does expose some level of risk that I believe could be mitigated by making TokenSpender the first contract an EOA hits, ensuring less logic and inter-contract trust sits between reading msg.sender and withdrawing msg.sender's tokens.

Current call tree:

1. EOA
    a. OneInchExchange: passes msg.sender to TokenSpender via calldata
        i. TokenSpender: passes calldata msg.sender to Token TransferFrom
            1. Token: Transfers tokens on behalf of OneInchExchange's msg.sender, as requested from OneInchExchange
    b. OneInchExchange performs swap

Alternative:

1. EOA
    a. TokenSpender: passes the real msg.sender directly to Token TransferFrom
        i. Token: Transfers tokens from msg.sender to OneInchExchange
        ii. OneInchExchange - Hard coded into TokenSpender as the only destination to be called after a transfer.

This alternative architecture:

1.) Relies on a static value for call target after a token transfer, instead of relying on a blacklist conditional for where a calls should not go
2.) Does not pass around msg.sender in a contract-trusted way
3.) Makes TokenSpender the first contract interacted with, so users are less likely to accidentally approve OneInchExchange

There are drawbacks to this alternative approach, adding more logic to the most critical portion of the contracts, and it is a judgement call. I did not find a specific attack that this design would prevent, but I believe this design keeps the most vulnerable portion of the system simpler.

Findings:

Critical:

[FIXED] facadeCall() could allow an attacker to withdraw approved user assets

https://github.com/CryptoManiacsZone/1inch-contract/blob/c998b0ffb71dba509047d4595289ddefe63180a8/contracts/OneInchExchange.sol#L146-L150

The `facadeCall` function allows a user to save their msg.sender address to storage prior to calling the swap function, in order to retrieve this saved msgSender value later and retrieve their own TokenSpender-approved assets. While the user controls which arbitrary calls are made during a swap, an attacker could create a malicious token or leverage an ERC777 callback to call into the OneInchExchange contract during this facadeCall, while msgSender value is set to the victim's address. By intentionally creating profitable arbitrage opportunities, an attacker

could cause 1inch aggregate orders to route through malicious callbacks, allowing them to drain all user-approved assets.

In this case, the assets at stake are not limited to what is being exchanged, but to the entire balance of all tokens approved to the TokenSpender.

Saving “msgSender” in this way is very similar to an attack on tx.origin.

Fixed: This issue was resolved by completely removing the facadeCall function in this PR: https://github.com/CryptoManiacsZone/1inch-contract/pull/19/files#diff-bdd7933169078432d0aac20255e988b5L144-L151

Medium:

[FIXED] ChiSpender vulnerable to signature replayable

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/ChiSpender.sol#L35-L39
ChiSpender.burnChi() has a flag for burning Chi from a sender based on a signature. While the signature is tied directly to the data used to execute the swap, if a user uses the signature-based BURN_FROM_SPONSOR, another user could copy the exact same data and, if the transaction doesn’t fail, burn that same sponsor’s chi tokens. Since the data would need to be exact, it is unlikely to benefit the attacker directly, but could still deduct chi tokens from the victim.

Fixed, completely removed signature-based chi burning
https://github.com/CryptoManiacsZone/1inch-contract/pull/24

Low:

No deadline on swap()

https://github.com/CryptoManiacsZone/1inch-contract/blob/626e4f0856fb951ea417b9f4a2201366a17c650c/contracts/OneInchExchange.sol#L45-L52

While swap() contains parameters that ensure the user’s transaction provides a constant amount of source for a minimum amount of destination tokens, a transaction that sits in the pending queue for long periods of time becomes a “free option”: one where if the price moves against them, it doesn’t execute (since the “minReturnAmount” will not be satisfied), but if it

moves in their favor, they could be committing to exchange tokens at a much lower price than the market is offering. This is especially profitable for miners. As an example, a user offers to buy DAI at a rate of 400 DAI per ETH, but the transaction is not provided sufficient gas to mine for a long time, long enough for the price of ETH to rise to 500 DAI per ETH. A miner could order transactions in such a way to ensure the user is only given 400 DAI per ETH.

If the arbitrary calls inside the transaction are hitting an on-chain order book or AMM, an attacker could order transactions such that they receive the majority of the benefit of this “mispriced” order.

(It is possible to add this deadline functionality with an arbitrary call to a time-checking contract that reverts)

Attacker could take “remainingAmount” tokens or excess return if malicious code is part of call list

Similar to the facadeCall issue above, it is possible for an attacker to re-enter the OneInchCaller contract in the middle of a swap, allowing the “slippage” from a swap (a user providing more of the source token than necessary to receive the minReturn amount of destination token to compensate for market movement) to be taken by a malicious callback or token. To pull off the attack, the attacker would need to convince 1inch to route part of the request through their malicious trade AND quickly feed fill information into their malicious system, such that it does not remove too much or else the transaction will revert.

Any time a contract enters malicious code in the middle of a multi-hop trade, the malicious code could accomplish this same attack in a number of ways that are not specific to 1inch (such as modifying AMM prices). This is a common issue with flexible routers.

Gifting CHI to pending transaction could cause transaction failure in specific circumstances

If a user submits a transaction to `OneInchExchange.discountedSwap()` method, their Ethereum client will perform an estimate based on their current quantity of CHI. If a user does not have enough CHI to fully refund their transaction, an attacker could give them CHI that would be burned due, requiring extra gas leading to an out-of-gas error for their transaction.

[FIXED] Function signature blacklist only prevents known transfer functions

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/OneInchExchange.sol#L157-L161

Each arbitrary call is checked to make sure it is not calling any variant of “transferFrom”, attempting to prevent token loss if someone accidentally approves the 1inch main contract, instead of TokenSpender. This should help with the most common functions that rely on approvals, but other function could act on this approval. For instance, an accidental 1inch-approval could call “burnFrom()” to destroy incorrectly approved user assets.

Fixed, new architecture does not rely on a blacklist at all

[FIXED] BytesPatcher library contains integer overflow

If an offset is greater than `2^256-32` , the “patch” would overwrite values just prior to the target “data” location.
https://github.com/CryptoManiacsZone/1inch-contract/blob/3eec86535f0afc966f4100d99bd0f332e23f126e/contracts/helpers/BytesPatcher.sol
This is mitigated by calls into BytesPatcher coming from a 253-bit extraction of a uint256, so numbers should not be close enough to `2^256` to cause an overflow.

Even so, an overflow here would be disastrous, allowing arbitrary memory writes. I recommend adding an overflow check here.

Fixed here:
https://github.com/CryptoManiacsZone/1inch-contract/pull/22/files#diff-47af76dba68a1fd59793a00ce63fb19940f889e7167952eb92d515530d873bd3L8
And furthermore, the new architecture helps mitigate the impact of overflow as an attack vector to retrieve approved assets.

[FIXED] Blacklist call to claimTokens function signature

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/OneInchExchange.sol#L157-L161
There is no specific vulnerability I can find related to makeCall() being able to interact with TokenSpender, but if somehow OneInchExchange could make an arbitrary call to TokenSpender, it would be disastrous for every approved TokenSpender balance. An additional check for the claimTokens function signature would add as second layer protection against some other vulnerability that violated `desc.target != address(tokenSpender)`

Fixed: New architecture does not rely on a blacklist at all

Chi tokens could be burned by attacker using tx.origin flag

Any user who has approved Chi tokens to the OneInchExchange contract (either for exchange or to be used as a gas token) could have their tokens burned by an attacker who is able to convince the victim to execute a seemingly unrelated transaction. This burning of Chi would not benefit the attacker, would require significant setup from the attacker to pull off, and the impact would be limited by transaction and block gas limits.

Imagine a scenario where an attacker AirDrops $10 of their own malicious token to a victim who has approved Chi to 1inch. The victim sees this token worth $10 and decides to sell the token on Uniswap. In the transfer function of the malicious token, it calls into 1inch swap function, with the “burn chi” flag and the “use tx.origin” flag. OneInchExchange would call Chi.freeFromUpTo() with the victim’s address.

This attack is limited by how much gas a victim gives to a transaction that enters a malicious contract, does not benefit the attacker or transfer Chi tokens to the attacker.

Note

Possible for anyone to receive misplaced tokens

Any token or ETH that finds its way onto the OneInchContract outside of a swap transaction is immediately claimable by anyone who asks. An airdrop (like UNI) would go to whoever asked for it first. An accidental transfer of an asset directly to the OneInchExchange contract would be immediately claimable by a bot monitoring token balances. This is *not a flaw* with the system as it is not designed to hold onto assets outside of a swap.

Bitwise flag inspection uses 3 different styles

I believe the bit-math is handled correctly, but there are 3 distinct ways bitwise flags are parsed.

Breakout functions:

```
    function _isPartialFill(uint256 flags) private pure returns
(bool) {
        return flags & _PARTIAL_FILL == _PARTIAL_FILL;
    }
```

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/OneInchExchange.sol#L184-L190

Inline inspection:

```
  if (chiSourceFlags & _BURN_FROM_MSG_SENDER ==
_BURN_FROM_MSG_SENDER) {
    chiSource = msgSender;
  } else if (chiSourceFlags & _BURN_FROM_TX_ORIGIN ==
_BURN_FROM_TX_ORIGIN) {
```

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/ChiSpender.sol#L33-L40

.high()/.low() combinations:

```
  if (skipMaskAndOffset.high(1) != 1) {
   // skip value patching flag is not set
    desc.value = desc.value.add(amount);
  }
```

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/extensions/PatcherExtension.sol#L83-L86

Note: this was reduced down to 2 different styles for bit math in final commit

Gas Savings:

These are possible ideas for saving gas, but they all come with drawbacks. Skipping these suggestions is perfectly reasonable and does not decrease the quality or security of the contract.

[FIXED] Function signature checking byte-by-byte

https://github.com/CryptoManiacsZone/1inch-contract/blob/049b36363b2d0a0d1f2318862e734ec9ac2b630c/contracts/OneInchExchange.sol#L157-L163

Every arbitrary call that is made looks byte-by-byte through the input to ensure it isn't calling a blacklisted function signature. The code avoids dropping into assembly, but ends up extracting single bytes of data to make many comparisons while the EVM is capable of evaluating the entire function signature in a single instruction.

As written, the difference in gas savings is not significant and the way it is written is clearer.

Fixed : New architecture has no need to check function signatures