CERTIK

PROFESSIONAL SERVICES

1inch Exchange

Aggregation Protocol v3

Security Assessment

March 15th, 2021

By:
Alex Papageorgiou @ CertiK
alex.papageorgiou@certik.org

Camden Smallwood @ CertiK
camden.smallwood@certik.org

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	1inch Exchange: Aggregation Protocol v3
Description	1inch exchange's Aggregation Protocol v3 smart contract release.
Platform	Ethereum; Solidity; Yul
Codebase	GitHub Repository
Commits	1. d3def083b875d3e04faf2caee758a1c4aaf43b7d

Audit Summary

Delivery Date	Mar. 15, 2021
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	Mar. 01, 2021 - Mar. 08, 2021

Vulnerability Summary

● **Total Informational**	*2 - 0 Resolved*

Executive Summary

The CertiK team audited a subset of the 1inch Exchange Aggregation Protocol v2.1 smart contracts over the course of March 1st to March 8th. The code is very well-written with concern shown for optimization and improving gas costs. One of the main concerns expressed by the 1inch team was to verify the optimized assembly code in the newly-written `OneInchUnoswap` contract. We annotated the assembly code of the `OneInchUnoswap` implementation in full and worked through some queries with the 1inch team.

The calculation block within the `swap` assembly function appears to replicate the `getAmountOut` function of the `UniswapV2Library`, however, the Uniswap counterpart uses literals whilst the 1inch implementation uses a dynamic variable and a constant. We queried whether we should assume that the dynamic numerator is meant to act as a sort of fee being set arbitrarily, to which the 1inch team noted that users would be able to bypass such fees by interfacing directly with the `unoswap` function. Additionally, we pointed out that the denominator appears to be set to 1000000000 whereas the Uniswap counterpart is set to 1000. The 1inch team pointed out that there are existing Uniswap forks which utilize different fixed or even dynamic fee models and the modification was designed to support those forks as well.

The `revertWithReason` assembly implementation appears to be conforming to the EIP838 draft and particularly the implementation that has been denoted in Solidity 0.8.X and up. We found the EIP mechanism manually implemented somewhat unconventionally. The 0.8.X style denotes that the data should be ABI encoded similarly to a function call ("This data consists of a 4-byte selector and subsequent ABI-encoded data.") quoting the documentation. The 1inch team's implementation stores the 4-byte signature in a full 32 byte slot and then proceeds to store the rest in the consequent 32 byte slots instead of following the ABI convention by storing i.e. at location 0x4. Additionally, even if one conducts AND operations to construct the ABI encoded expected value, the offsets are off by a margin of 1 byte on the second instruction of `revertWithReason` as well as all `revertWithReason` invocation message arguments. For reference, the EIP838 was based on the following encoding principle which seems to also differ to your implementation i.e. data offset is different: https://github.com/ethereum/EIPs/issues/838#issuecomment-458919375

The 1inch team stated that they are following the convention and supplied a breakdown of the data to demonstrate that in memory it lies down as follows:

```
0x08c379a0 - selector
0x0000000000000000000000000000000000000000000000000000000000000020 - data
offset
0x0000000000000000000000000000000000000000000000000000000000000011 -
string length
0x696e76616c6964206d73672e76616c756500000000000000000000000000000000 -
actual string

Total data size is 4 + 32 + 32 + 17 = 85 which is equal to 0x55
```

We also pointed out that it may be more optimal to hard-code the len of the full payload to the highest one passed to the function (i.e. 0x5A) and allow off-chain processes to deal with the surplus 0 bits in the error message, which should reduce the gas cost in these invocations. The 1inch team declined this optimization, stating that leaving trailing zeros might be messy and they would prefer to keep the more correct implementation.

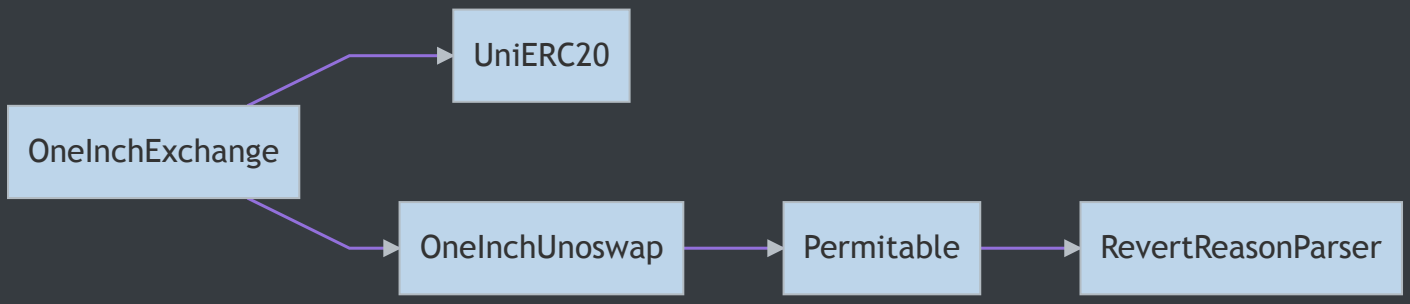
Within the `OneInchExchange` contract, we noticed that the `discountSwap` function is allowed to execute when the contract is paused, which is undocumented. This may be unintentional as the regular `swap` function does contain the `whenNotPaused` modifier. See OIE-01 for more information.

We determined that the `receive` function ensures that the sender of funds is not an EOA and pointed out that a more suitable check would be to ensure that the depositor of ETH is the WETH address, as it should not intend to receive ETH deposits from other contracts. This would prevent accidental deposits on the contract side as well. The 1inch team stated that allowing only WETH to deposit ETH in the `receive` function seems valid, but this check is in place only for fool protection. They also highlighted the `rescueFunds` function in the `OneInchExchange` contract, which allows them to rescue ETH accidentally sent to it. Granted funds are rescuable from the contract, we determined that the existing check is sufficient, but have noted that the `rescueFunds` function only allows the contract owner to retrieve the

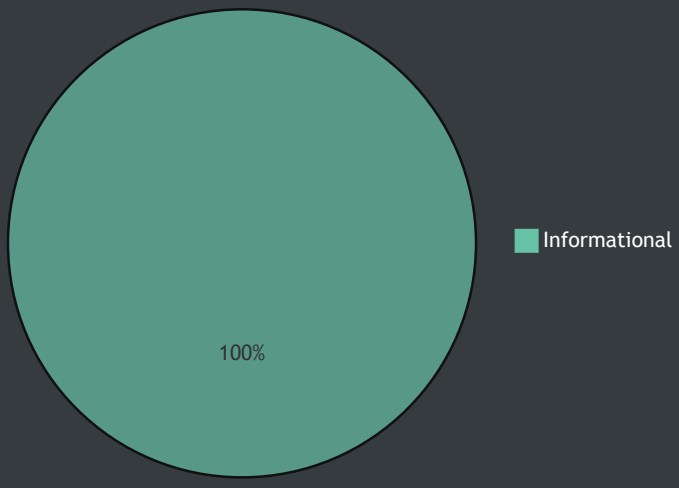
funds for themself. An improvement to the implementation would be to include a recipient address in order to send the funds back to their original owner directly. See OIE-02 for more information.

Files In Scope

ID	Contract	Location
OIE	OneInchExchange	contracts/OneInchExchange.sol
OIU	OneInchUnoswap	contracts/OneInchUnoswap.sol
PER	Permitable	contracts/helpers/Permitable.sol
RRP	RevertReasonParser	contracts/helpers/RevertReasonParser.sol
UNI	UniERC20	contracts/helpers/UniERC20.sol



Findings



ID	Title	Type	Severity	Resolved
OIE-01	Discounted swap can be performed while paused	Implementation	Informational	ⓘ
OIE-02	Contract owner can only rescue funds for themself	Implementation	Informational	ⓘ

OIE-01: Discounted swap can be performed while paused

Type	Severity	Location
Implementation	Informational	contracts/OneInchExchange.sol L46

Description:

The `discountedSwap` function in the `OneInchExchange` contract does not incorporate the `whenNotPaused` modifier as the `swap` function does.

Recommendation:

While this behavior is not explicitly defined in documentation in comments, we suggest clarifying if the `discountedSwap` function in the `OneInchExchange` contract should be allowed to execute when the contract is paused.

OIE-02: Contract owner can only rescue funds for themself

Type	Severity	Location
Implementation	Informational	contracts/OneInchExchange.sol L142-L144

Description

The `rescueFunds` function allows the owner of the contract to transfer an arbitrary amount from an arbitrary `IERC20` token to themself in order to retrieve funds accidentally sent to it:

```
function rescueFunds(IERC20 token, uint256 amount) external onlyOwner {
  token.uniTransfer(msg.sender, amount);
}
```

Recommendation:

Consider incorporating a recipient address in order to transfer funds that were accidentally transfered to the contract back to their original owner instead of limiting it to only the owner of the contract.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Arithmetic

Arithmetic exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.