Uniswap Audit

CONSENSYS Diligence

1 Summary

Uniswap is a decentralized exchange hosted on the main Ethereum blockchain. It enables users to trade any ERC20 token for ETH, or for another ERC20 token. It has no native token, and no fees are charged by Uniswap's creators. Thus it can be considered a public good.

From December 10 to January 11 (excluding holidays) four members of the ConsenSys Diligence team conducted a security audit on the Uniswap system.

Unsiwap is written in Vyper, whereas the vast majority of contracts have been written in Solidity. To our knowledge, this is the first security audit conducted on a Vyper codebase.

We

1.1 Audit Dashboard

Audit Details

- **Project Name:** Uniswap
- **Client Name:** Uniswap
- **Client Contact:** Hayden Adams
- **Auditors:** John Mardlin, Gonçalo Sá, Dean Pierce, Sergii Kravchenko, Daniel Luca
- **GitHub :** ConsenSys/Uniswap-audit-internal-2018-12
- **Languages:** Vyper
- **Date:** January 11th 2019

Number of issues per severity

	Minor	Medium	Major	Critical

	Minor	Medium	Major	Critical
Open	5	1	1	0
Closed	0	0	0	0

1.2 Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

Security: Identifying security related issues within each contract and within the system of contracts. **Sound Architecture:** Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices. **Code Correctness and Quality:** A full review of the contract source code. The primary areas of focus include:

- Correctness
- Readability
- Sections of code with high complexity
- Quantity and quality of test coverage

1.3 System Overview

Documentation

The following documentation was available to the audit team, and provides the necessary context for this report:

- White Paper
- Docs
- Runtime Verifications Formal Specification of Market Maker Model

Summary of Formal Verification work done by Runtime Verification

We were provided with a report by Runtime Verification (RV) Inc. describing some properties of Uniswap exchange. This report contains a formal specification of the Uniswap exchange model (constant product market maker model) that includes a description of price discovery including fees, tokens trading and adding/removing liquidity.

Since the implementation is restricted to integer arithmetic, the main purpose of the RV report is to analyze the difference between the theoretical model and the implemented model. This report only analyzes rounding errors that are made on the transition to integer arithmetics. It proves that no attack can be made to benefit from rounding errors, and that every rounding error is made in favour of the liquidity pool.

The RV report does not cover all possible attacks on the actual smart contract system, such as frontrunning, reentrancy, etc.

Scope

Contract file name	SHA1 Hash
uniswap_exchange.vy	9b058dc847040594bcac502effab5bda0de5fa3c
uniswap_factory.vy	97d49145ec4fc6aa31099cb51c0c2f69b6e487b7

1.4 Key Observations

On Auditing Vyper

Vyper has the disadvantage of being a newer language with a smaller community, thus having fewer security analysis tools available. Fortunately, Vyper's design philosophy priortizes both auditability and legibility. The LLL IR from the compiler made it easier to see which opcodes are used to achieve the written instructions.

Despite Vyper's emphasis on legibility, we observed some unituitive output from the compiler. In particular, the built in function `create_with_code_of(address)` does not use the code of the input address, but rather creates a contract which delegates it's logic to the input address.

On the quality and preparedness of Uniswap

Uniswap's documentation is thorough and well written. The codebase contains natspec comments on each function. The code is written defensively, with frequent assert statements to revert calls with invalid input.

We found the test coverage to be incomplete. Including untested behavior, and sections of code which were untested. The majority of the tests are positive test cases, meaning that the tests confirm that the system works with an expected sequence of actions and inputs. The test suite should be expanded to include more negative scenarios to ensure that the safe checks within the contract system are working correctly.

1.5 Recommendations

The issues in our report do not necesessitate replacing the system as it currently stands.

An important consideration in our recommendation is that the Uniswap system has been live on the main Ethereum network for several months, and holds over $300,000 in its liquidity pools. This provides a natural incentive to attack the system, suggesting that no known vulnerabilities currently exist.

Our primary recommendation is to extend the test suite to cover 100% of the code, and to include testing to ensure for undesirable behavior.

2 Threat Model

2.1 Overview

Uniswap is a decentralized exchange, which, from the start, gives it a large number of potential adversaries with strong incentives to take advantage of the system. Here we examine the various malicious actors, and the potential impact they may have on the system.

2.2 Detail

Malicious Ethereum Attacker

The contracts are live on mainnet, giving anyone the ability to poke at them. The functionality in these contracts is fairly minimal, and the implementation in vyper has likely mitigated many of the classic implementation errors, so it seems like an attacker is going to have the most luck attacking extra features in the attached ERC20 token.

Malicious Trader

Two key areas of attack for malicious traders would be trying to stack rounding issues, and skimming trades via front running. Rounding issues are unlikely to be much of a problem because rounding always favors the liquidity providers, but is likely to be a key attack vector for a while. Specifics and potential mitigations are discussed in **3.1**.

Malicious Miner

The key advantage that a Malicious Miner has is the ability to front run transactions much more reliably. Hopefully the social incentives for being a fair pool operator will preclude any of the major pools from participating in this sort of activity, but the threat exists.

Malicious Liquidity Provider

A large liquidity provider primarily has the advantage when performing rounding attacks. Since rounding errors favor the liquidity provider, someone may temporarily take something like a 95% stake in the liquidity pool, and then attempt to stack rounding issues to drain funds from the liquidity pool. We have so far been unable to figure out a sequence of events that would be favorable to the attacker.

Malicious Exchange Creator

Importantly, anyone can create an exchange, and can set it to any ERC20 token they like, which could lead to a few types of attacks. An attacker may attempt to impersonate a popular token by naming it similarly, though as long as the default exchanges are statically added to the frontend manually exposure should be limited.

More interestingly, the exchange creator could register a well known legitimate token, but initialize the liquidity pool in such a way that the token can never be used on the platform.

There is also nothing in Uniswap to ensure a token address provided to the the Factory, is compliant with ERC20. However, Exchange contracts are created via a template, so the Exchange code can't be tampered with. The ERC20 tokens could be contracts designed to attack Uniswap, or particularly vulnerable contracts might be added more prone to attack.

Malicious Web Attacker

Since the front-facing portion of Uniswap is hosted on a website, anyone who gains access to the DNS, hosting, or Cloudflare account could deploy a malicious Uniswap frontend that could, among other things, redirect transactions meant for the Exchanges to a wallet controlled by the attacker. An attacker may also go after one of the many dependencies pulled in by NPM to inject themselves into the deployment process.

3 Issue Overview

The following table contains all the issues discovered during the audit. The issues are ordered based on their severity. More detailed description on the levels of severity can be found in Appendix 2. The table also contains the Github status of any discovered issue.

Chapter	Issue Title	Issue Status	Severity
3.1	Liquidity pool can be stolen in some tokens (e.g. ERC-777)	Open	Major
3.2	Frontrunners can skim ~2.5% from every transaction.	Open	Medium
3.3	Gaps in test coverage	Open	Minor
3.4	Consider using transferFrom() in removeLiquidity() function	Open	Minor
3.5	Different 'deadline' behaviour	Open	Minor
3.6	Redundant checks in factory contract	Open	Minor
3.7	The factory contract should use a constructor	Open	Minor

4 Issue Detail

3.1 Liquidity pool can be stolen in some tokens (e.g. ERC-777) (#29)

Severity	Status	Link	Remediation Comment
Major	Open	issues/29	The issue is currently under review

Description

If token allows making reentrancy on `transferFrom(address from, address to, uint tokens)` function by someone except the recipient, then all the liquidity funds might be stolen. For example, if token calls callback function of `from` address. It's irrelevant if reentrancy is done before or after the balances update.

Attack

Let's imagine we have a token that calls a callback function of `from` address on `transferFrom(address from, address to, uint tokens)` and allows `from` address to make a reentrancy. We will consider the case when reentrancy is made after the token balances are updated. If token balances are updated after the reentrancy (e.g. ERC-777), the algorithm is even easier and requires fewer funds to steal liquidity pool.

In `tokenToTokenInput` we have the following 2 lines of code

```
assert self.token.transferFrom(buyer, self, tokens_sold)
tokens_bought: uint256 =
Exchange(exchange_addr).ethToTokenTransferInput(min_tokens_bought, deadline, recipient,
value=wei_bought)
```

Attacker(`buyer`) can make reentrancy on the first line here.

1. Assume we have an exchange with a token that worth equally to ETH with liquidity pool equals (100 tokens, 100 ETH)
2. An attacker creates a fake Exchange (it will be the second exchange in `tokenToToken` transfers) that will receive ETH from the first exchange and behave like a normal exchange.
3. The attacker can buy 50 ETH for 100 tokens by using `tokenToTokenInput` function.
4. New liquidity pool should be (200 tokens, 50 ETH) but since the attacker makes reentrancy on `assert self.token.transferFrom(buyer, self, tokens_sold)` it will still be (200 tokens, 100 ETH).
5. While making reentrancy the attacker can buy 49.999 ETH for about 200 tokens using `tokenToEthSwapInput` .
6. After that, the liquidity pool should look like (400 tokens, 0.001 ETH)
7. Now the attacker can buy all the tokens for a very small amount of ETH.

This is not a very accurate algorithm that does not include fees and gas cost, but logic stays the same.

Remediation

Add mutex to all functions that make trades in order to prevent reentrancy.

3.2 Frontrunners can skim ~2.5% from every transaction. (#30)

Severity	Status	Link	Remediation Comment
Medium	Open	issues/30	The issue is currently under review

The default client contains a constant called `ALLOWED_SLIPPAGE` which states how much the price is allowed to change. This constant is set to 0.025 in `uniswap-frontend/src/pages/Swap/index.js:367`. This means that the price can go up ~2.5% from what the buyer expects, and the order will still get executed.

Any user on the Ethereum network has the ability to watch for new transactions being sent to the network. When the attacker sees a large victim transaction that they want to front run come in, they can create a similar transaction that would move the market up by no more than 2.5%. They then increase their gas fees to ensure that their order gets executed first. The attacker transaction executes, raising the price of the asset, and then the victim transaction executes at the higher price. The attacker is then free to exit the position immediately, pocketing the difference, having never exposed themselves to any risk.

Sophisicated front-runners will likely call these transactions from their own contract addresses to make sure they end up with the prices they expect, and don't collide with other front-runners.

This is a hard problem in general, and front-running of some form is always to be expected. By reducing the `ALLOWED_SLIPPAGE` (maybe even to 0), you can reduce the amount that front-runners can get for free from every transaction. The slippage value should probabally also be exposed to the user so they can opt in to a 0% slippage value if they choose.

Even with zero slippage, the attacker can still make a large buy order, watch the victims order fail, wait until they make a new order, and then exit their position. Doing this does carry the additional risk that the victim may be unhappy with the new price, which eliminates the attackers opportunity for zero risk skimming.

Bancor uses a front-running mitigation where there was a maximum gas value, so a user using the maximum gas value cannot be front-run by an attacker increasing the gas for their transaction. This approach might be worth looking into.

3.3 Gaps in test coverage (#32)

Severity	Status	Link	Remediation Comment

Severity	Status	Link	Remediation Comment
Minor	**Open**	issues/32	The issue is currently under review

Description

The test suite could be improved in certain areas. Different types of testing parameters and testing methodologies could be used to further extend its coverage.

Remediation

Specific areas that were identified as lacking proper coverage:

- Additional ERC20 contracts with edge case parameters set (e.g.: `decimals = 0 | 18 | MAX_UINT8`, `totalSupply = 0 | MAX_UINT256`, ...) as opposed to only testing with the `HAY` token
- Stress tests with some blackbox fuzzing with random amounts and parameters for ETH <-> ERC20 trades and ERC20 <-> ERC20 trades instead of fixed amounts.
- Do unit testing in exchange functions with proper state reset between each test as opposed to doing only end-to-end tests with persisting state.
- In the end-to-end tests more scenarios could be added to test for adverse conditions like big slippage.

3.4 Consider using transferFrom() in removeLiquidity() function (#31)

Severity	Status	Link	Remediation Comment
Minor	**Open**	issues/31	The issue is currently under review

Description

To prevent issues like the BNB issue[1] from coming up in the future, consider using the same transfer function to add and remove liquidity. Hopefully this will ensure that if any non-compliant tokens get added in the future, it will be much more unlikely to get into a state where liquidity can be added, but not removed.

[1] https://twitter.com/UniswapExchange/status/1072286773554876416

3.5 Different 'deadline' behaviour (#25)

Severity	Status	Link	Remediation Comment
Minor	**Open**	issues/25	The issue is currently under review

Description

Many functions in `Exchange` contract have `deadline` as a parameter. This parameter has the same description in every function. But some functions use `>` operator when checking for a deadline

```
assert deadline > block.timestamp
```

while other functions use `>=` operator.

Remediation

Change all `>` operators to `>=` when working with deadline parameter. It does not really affect anything but keeps code more beautiful and consistent.

3.6 Redundant checks in factory contract (#24)

Severity	Status	Link	Remediation Comment
Minor	**Open**	issues/24	The issue is currently under review

Description

In `uniswap_factory.vy` there are some redundant assertions:

contracts/uniswap_factory.vy:L15

```
    assert template != ZERO_ADDRESS
```

and

contracts/uniswap_factory.vy:L21

```
    assert self.exchangeTemplate != ZERO_ADDRESS
```

Due to the fact that the Vyper compiler actually asserts that the code size at the specified exchange address (through the use of `EXTCODESIZE`) is bigger than `0`, here:

contracts/uniswap_factory.vy:L24

```
    Exchange(exchange).setup(token)
```

This meaning that the zeroth address as the exchange, having no code at all, would make the call to `setup()` fail.

Remediation

Remove the two assertions.

3.7 The factory contract should use a constructor (#23)

Severity	Status	Link	Remediation Comment
Minor	Open	issues/23	The issue is currently under review

Description

The factory uses a public function for initialization instead of a constructor which might make it a target for griefing attacks.

Since there is no way to actually deploy the contract and call the `initializeFactory()` method within the same transaction without an additional contract designed for the effect (which seems to be inexistent in the current codebase) an attacker could grief deployments of the Uniswap factory by always front-running the initializing transaction after deployment.

Possibly even more worrisome would be a front-running transaction that would underhandedly change the exchange template code to something very similarly benign but that was, for example, an underhandedly backdoored version of the original.

Remediation

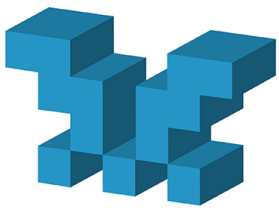
The easiest solution would be to turn the initialization method into the constructor of said contract. Another possible remediation would be to introduce a "factory deployer" contract that executes both message calls in a single transaction.

5 Tool based analysis

The issues found using tool based analysis have been reviewed and the relevant issues have been listed in chapter 3 - Issues. Fewer tools support code written with Vyper than Solidity, the following were included in our analysis.

5.1 Mythril Classic

The Mythril Classic uses concolic analysis to detect various types of issues. The tool was used for automated vulnerability discovery for all audited contracts and libraries. More details on MythX's current vulnerability coverage can be found here.



The raw output of the Mythril Classic vulnerability scan for each contract:

- uniswap_exchange.vy
- uniswap_factory.vy

5.2 Harvey Fuzzer

Harvey is a grey box fuzzer designed specifically for the EVM.

The raw output of Harvey's analysis for each contract:

- uniswap_exchange.vy
- uniswap_factory.vy

6 Test Coverage Measurement

Testing is implemented using eth-tester. 21 tests are included in the test suite and they all pass.

Specific sections of the code where necessary test coverage is missing are included in chapter 3 - Issues.

It's important to note that "100% test coverage" is not a silver bullet. Our review also included a inspection of the test suite, to ensure that testing included important edge cases.

The state of test coverage at the time of our review can be viewed in coverage_output.md.

Appendix 1 - File Hashes

The SHA1 hashes of the source code files in scope of the audit are listed in the table below.

Contract file name	SHA1 Hash
uniswap_exchange.vy	9b058dc847040594bcac502effab5bda0de5fa3c
uniswap_factory.vy	97d49145ec4fc6aa31099cb51c0c2f69b6e487b7

Appendix 2 - Severity

A.2.1 - Minor

Minor issues are generally subjective in nature, or potentially deal with topics like "best practices" or "readability". Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgment as to whether addressing these issues improves the codebase.

A.2.2 - Medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

A.2.3 - Major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable, or may require a certain condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

A.2.4 - Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.

Appendix 3 - 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 Solidity code and only the Solidity code we note as being within the scope of our review within this report. 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 Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) -- on its Github account (https://github.com/ConsenSys). 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.