SmartDec

Monti Software Smart Contracts Security Analysis

This report is public.

Published: January 24, 2019.

Abstract

In this report, we consider the security of the Monti Software project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audit is not an investment advice.

Summary

In this report, we considered the security of Monti Software smart contracts. We performed our audit according to the procedure described below.

The initial audit showed no critical issues. However, a number of medium and low severity issues were found. Most of them were fixed in the latest version of the code.

General recommendations

The contracts code is of good code quality. However, we recommend implementing Tests and deployment scripts.

Checklist

Security

The audit showed no vulnerabilities.
Here by vulnerabilities we mean security issues that can be exploited by an external attacker. This does not include low severity issues, documentation mismatches, overpowered contract owner, and some other kinds of bugs.

Compliance with the documentation

The audit showed no discrepancies between the code and the provided documentation.

ERC20 compliance

We checked ERC20 compliance during the audit. The audit showed that **Vexchange V1 - Uniswap Fork** token was fully ERC20 compliant.

ERC20 MUST

The audit showed no ERC20 “MUST” requirements violations.

ERC20 SHOULD

The audit showed no ERC20 “SHOULD” requirements violations.

Tests

The tests in the repository are not for the audited smart-contracts.

The text below is for technical use; it details the statements made in Summary and General recommendations.

Procedure

In our audit, we consider the following crucial features of the smart contract code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).
3. Whether the code meets best practices in efficient use of gas, code readability, etc.

We perform our audit according to the following procedure:

- automated analysis
  - we scan project's smart contracts with our own Vyper static code analyzer SmartCheck
  - we manually verify (reject or confirm) all the issues found by the tool
- manual audit
  - we manually analyze smart contracts for security vulnerabilities
  - we check smart contracts logic and compare it with the one described in the documentation
  - we check ERC20 compliance
- report
  - we reflect all the gathered information in the report

Checked vulnerabilities

We have scanned Monti Software smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered (the full list includes them but is not limited to them):

- Reentrancy
- Front Running
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Limit and Loops
- Locked money
- Unchecked external call
- ERC20 Standard violation
- Unsafe use of timestamp
- Using blockhash for randomness
- Balance equality
- Unsafe transfer of ether
- Fallback abuse
- Private modifier

Project overview

Project description

In our analysis we consider Monti Software specification ("Vexchange Documentation.docx", sha1sum: 25d73c616dc2216f64d8a6df7f1a52bdbddf9501), documentation of Uniswap project (the one that audited project was forked from), and smart contracts' code (version on commit 166eec03ec7c1e7ab137b5def77c5448c915bf0c).

The latest version of the code

After the initial audit, some fixes were applied and the code was updated to the latest version (version on commit d73e3c6eb74add88e5281d12a1c1d1a5cb6ea984).

Project architecture

For the audit, we were provided with the project, which is a `pip` package.

- The project successfully compiles with `vyper` command

The total LOC of audited Vyper sources is 425.

Automated analysis

We used publicly available automated Vyper analysis tool. Here are the results of SmartCheck scanning. All the issues found by the tool were manually checked (rejected or confirmed).

True positives are constructions that were discovered by the tool as vulnerabilities and can actually be exploited by attackers or lead to incorrect contracts operation.

False positives are constructions that were discovered by the tool as vulnerabilities but do not consist a security threat.

Cases when these issues lead to actual bugs or vulnerabilities are described in the next section.

Tool	Rule	True positives	False positives
SmartCheck	Vyper erc20 approve	1	
	Vyper private modifier does not hide data	12	3
Total SmartCheck		13	3
Total Overall		**13**	**3**

Manual analysis

The contracts were completely manually analyzed, their logic was checked and compared with the one described in the documentation. Besides, the results of the automated analysis were manually verified. All confirmed issues are described below.

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

The audit showed no critical issues.

Medium severity issues

Medium issues can influence smart contracts operation in current implementation. We highly recommend addressing them.

Bugs

There are several bugs in the contracts code:

- **uniswap_exchange.vy**, lines 60–66:

```
def sqrt(x: uint256) -> uint256:
    z: uint256 = (x + 1) / 2
    y: uint256 = x
    for i in range(18):
        y = z
        z = ((x / z) + z) / 2
    return y
```

  This function fails with an exception due to division by zero if `x` equals zero (`sqrt(0)`). Also, `sqrt()` function will execute the senseless loop if `x` equals one (`sqrt(1)`). We recommend fixing `sqrt()` function by separately considering the case when `x` is equal to zero.

- **uniswap_factory.vy**, lines 24–25:

```
self.default_max_platform_fee = 1000
self.default_max_swap_fee = 1000
```

  However, the accuracy of calculations is `1/10000` in **uniswap_exchange.vy** file.

Also, the comment in **uniswap_exchange.vy**, line 33 says:

```
# must be between 1 to 10000 that represent the the fee
percent from 10000
```

We recommend replacing the values of `default_max_platform_fee` and `default_max_swap_fee` variables with `10000`.

- The formula used in `cacluate_platform_profit()` function is incorrect (**uniswap_exchange.vy**, line 73):

```
platform_liquidity_minted: uint256 = (total_liquidity *
platform_profit * self.platform_fee
/(10000+platform_profit*(10000 - self.platform_fee)))/1000
```

`platform_profit` variable in the denominator of the fraction should be divided by `1000`. We recommend replacing this formula with the following one:

```
platform_liquidity_minted: uint256 = total_liquidity *
platform_profit * self.platform_fee
/(10000000+platform_profit*(10000 - self.platform_fee))
```

The initial formula was modified. The recheck did not reveal any vulnerabilities in the new one.

The issues have been fixed and are not present in the latest version of the code.

Invalid code logic

There is a code logic issue in **uniswap_exchange.vy**, line 578:

```
def token_scrape(token_addr: address, deadline: timestamp)
```

`token_scrape()` function works incorrectly: ETH received after `Exchange(exchange_addr).tokenToEthSwapInput()` call will trigger `__default__()` function of the current exchange contract. Thus, `ethToTokenInput()` function will be called with `2300 gas` stipend provided by `send()` function. Therefore, the whole transaction will revert with an out-of-gas exception. Hence, it is impossible to unlock stuck tokens. We highly recommend fixing this functionality.

The issue has been fixed and is not present in the latest version of the code.

No tests and deployment script

The provided code does not contain tests. Testing is crucial for code security and audit does not replace tests in any way.
We highly recommend both covering the code with tests and making sure that the test coverage is sufficient.

There is also no deployment script. However, the contracts deployment does not seem trivial. Bugs and vulnerabilities often appear in deployment scripts and severely endanger system's security. We highly recommend developing and testing deployment scripts very carefully.

Low severity issues

Low severity issues can influence smart contracts operation in future versions of code. We recommend taking them into account.

Lack of the documentation

The documentation provided with the code is not complete. The information regarding the formula used in `cacluate_platform_profit()` function (**uniswap_exchange.vy**, line 70) is not sufficient. We recommend clarifying the description of this formula in the documentation.

Bug

There is a bug in **uniswap_exchange.vy**, line 112:

```
self.previous_invariant = as_unitless_number(msg.value) *
max_tokens
```

We recommend calculating `self.previous_invariant` variable in the same way as on line 99:

```
self.previous_invariant = as_unitless_number(self.balance) *
(token_reserve + token_amount)
```

This formula is more secure as ETH or tokens can be transferred to the contract before the first `addLiquidity()` function call.

The issue has been fixed and is not present in the latest version of the code.

ERC20 approve

There is ERC20 approve issue in **uniswap_exchange.vy**, line 536: changing the approved amount from a nonzero value to another nonzero value allows a double spending with a front-running attack.

We recommend implementing `increaseApproval()` and `decreaseApproval()` functions. Moreover, we recommend instructing users to follow one of two ways:

- not to use `approve()` function directly and to use `increaseApproval()/decreaseApproval()` functions instead
- to change the approved amount to 0, wait for the transaction to be mined, and then to change the approved amount to the desired value

The issue has been fixed and is not present in the latest version of the code. Nevertheless, we still recommend instructing users in the ways described above.

Invalid values

The values of `name` and `symbol` state variables used for the liquidity tokens are the same as in the original project (**uniswap_exchange.vy**, lines 47–48):

```
self.name = 0x556e697377617020563100000...
self.symbol = 0x554e492d56310000...
```

We recommend replacing them with the new ones.

The issue has been fixed and is not present in the latest version of the code.

Wrong ABI

In `abi` folder, `uniswap_exchange.json` and `uniswap_factory.json` files do not match the corresponding ABI files, generated by the compiler. We recommend replacing wrong ABI with the correct ones.

The issue has been fixed and is not present in the latest version of the code.

Typo

We found several typos in the code:

- **uniswap_factory.vy**:

```
defualt_swap_fee: uint256
```

  There should be `default_swap_fee` instead of `defualt_swap_fee` across the file.

- **uniswap_exchange.vy**, line 70:

```
def cacluate_platform_profit(eth_reserve: uint256,
token_reserve: uint256) -> uint256:
```

  There should be `calculate_platform_profit` instead of `cacluate_platform_profit`

- **uniswap_exchange.vy**, line 77:

```
@dev min_amount has a djfferent meaning when total UNI
supply is 0.
```

  There should be `different` instead of `djfferent`.

- In **uniswap_exchange.vy**, lines 147,161:

```
@dev Pricing functon for converting between ETH and Tokens.
```

  There should be `function` instead of `functon`.

The issues have been fixed and are not present in the latest version of the code.

NatSpec mismatch

There are several NatSpec comments mismatches:

- **uniswap_exchange.vy**, line 79: `max_tokens` should be used instead of `min_amount`.
- **uniswap_exchange.vy**, line 452: `exchange_addr` should be used instead of `token_addr`.

We highly recommend fixing all the mismatches and using the NatSpec accurately and correctly.

The issues have been fixed and are not present in the latest version of the code.

Code style

In **uniswap_factory.vy** file, `initializeFactory()` function is used for variables initialization. However, it is not the best practice. We recommend using the constructor for state variables initialization instead.

The issue has been fixed and is not present in the latest version of the code.

Redundant code

The following line is redundant (**uniswap_exchange.vy**, line 108):

```
token_amount: uint256 = max_tokens
```

`token_amount` variable is redundant as it is equal to `max_token` variable. We highly recommend removing redundant code in order to improve code readability and transparency and decrease cost of deployment and execution.

The issue has been fixed and is not present in the latest version of the code.

Missing checks

There are several missing checks in the code:

- **uniswap_factory.vy**, lines 47–50, 53–56:

  ```
  def setMaxPlatformFee(_default_max_platform_fee: uint256)
  def setMaxSwapFee(_default_max_swap_fee: uint256)
  ```

  In these functions the new maximum value is not checked to be greater than the corresponding current fee value (`default_platform_fee` or `default_swap_fee` respectively). We highly recommend implementing the following checks in `setMaxPlatformFee()` and `setMaxSwapFee()` functions respectively:

  ```
  assert _default_max_platform_fee >=
  self.default_platform_fee
  assert _default_max_swap_fee >= self.default_swap_fee
  ```

- In `tokenToTokenTransferInput()` and `tokenToTokenTransferOutput()` functions (**uniswap_exchange.vy**, lines 355–357, 397–399) the following check is missing:

```
assert recipient != self
```

  We highly recommend implementing missing check in order to avoid loss of tokens.

The issues have been fixed and are not present in the latest version of the code.

Notes

Malicious tokens

The platform is designed to be used with various tokens with an unknown code. This can lead to the undesired consequences for the corresponding **uniswap_exchange** contracts:

- Reentrancy at lines 256–257:

```
send(recipient, wei_bought)
assert self.token.transferFrom(buyer, self, tokens_sold)
```

- Reentrancy at lines 291–292:

```
send(recipient, eth_bought)
assert self.token.transferFrom(buyer, self, tokens_sold)
```

- Impact on the economy due to unlimited mint/burn functionality. For instance, if the token owner burns tokens from **uniswap_exchange** contract, the invariant will decrease.

As this problem cannot be fixed, we recommend taking it into account.

Unexpected ETH/tokens

`previous_invariant` state variable of **uniswap_exchange** contract is supposed to keep invariant at the moment of the last `addLiquidity()/removeLiquidity()` functions call. Moreover, the difference between the new invariant and the previous one is supposed to appear based on the fees from the trade volume. However, it can also be changed by sending ETH or tokens to the contract using `selfdestruct()` and `transfer()` functions respectively without calling the `addLiquidity()/removeLiquidity()` functions. We recommend taking it into account.

NatSpec misuse

The NatSpec comments in the code violates the Vyper Documentation (starting from v0.1.0-beta.6). We recommend using triple quotes (`"""`) instead of hashes (`#`) to highlight NatSpec comments.

The issue has been fixed and is not present in the latest version of the code.

Private visibility level

Private visibility level is used in the following lines:

- **uniswap_exchange.vy**, lines 27–38:

```
balances: uint256[address]
allowances: (uint256[address])[address]
token: address(ERC20)
factory: Factory
last_invariant: decimal
owner: address
platform_fee: uint256
platform_fee_max: uint256
swap_fee: uint256
swap_fee_max: uint256
anotherToken: address(ERC20)
previous_invariant: uint256
```

- **uniswap_factory.vy**, lines 12–15:

```
defualt_swap_fee: uint256
default_platform_fee :uint256
default_max_swap_fee: uint256
default_max_platform_fee: uint256
```

We recommend taking into account that, contrary to a popular misconception, anyone can see values of private variables in the blockchain.

This analysis was performed by SmartDec.

Alexander Seleznev, Chief Business Development Officer
Boris Nikashin, Project Manager
Evgeniy Marchenko, Lead Developer
Igor Sobolev, Analyst
Pavel Kondratenkov, Analyst
Alexander Drygin, Analyst
Kirill Gorshkov, Analyst

Sergey Pavlin, Chief Operating Officer

January 24, 2019