DYP LOCKS AUDIT

December 2020

BLOCKCHAIN CONSILIUM

Contents

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND BLOCKCHAIN CONSILIUM DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH BLOCKCHAIN CONSILIUM.

Purpose of the report

The Audits 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 the Solidity programming language that could present security risks. Cryptographic tokens and smart contracts are emergent technologies and carry with them high levels of technical risk and uncertainty.

The Audits are not an endorsement or indictment of any particular project or team, and the Audits 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 Audits in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. 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. There is no owed duty to any Third-Party by virtue of publishing these Audits.

Introduction

We first thank dyp.finance for giving us the opportunity to audit their smart contract. This document outlines our methodology, audit details, and results.

dyp.finance asked us to review their DYP Lock smart contracts (GitHub Commit Hash: `90b0a202a0691a4ec931ee71ac823f2272f785f7`). Blockchain Consilium reviewed the system from a technical perspective looking for bugs, issues and vulnerabilities in their code base. The Audit is valid for `90b0a202a0691a4ec931ee71ac823f2272f785f7` GitHub Commit Hash only. The audit is not valid for any other versions of the smart contract. Read more below.

Audit Summary

This code is clean, thoughtfully written and in general well architected. The code conforms closely to the documentation and specification.

Overall, the code is clear on what it is supposed to do for each function. The visibility and state mutability of all the functions are clearly specified, and there are no confusions.

https://github.com/dypfinance/token-lock-vesting-contracts/tree/90b0a202a0691a4ec931ee71ac823f2272f785f7

Audit Result	PASSED
High Severity Issues	None
Moderate Severity Issues	None
Low Severity Issues	None
Informational Observations	None

Overview

The DeFi Yield Protocol is developing a unique platform that allows anyone to provide liquidity and to be rewarded for the first time with Ethereum. At the same time, the platform maintains both token price stability as well as secure and simplified DeFi for end users by integrating a DYP anti-manipulation feature.

Lock smart contracts are meant to lock DYP Tokens for the team for a predefined duration, two types of locks are included, Token Lock and Token Vesting Lock, the Token Lock smart contract Locks up ERC20 Tokens for a certain duration, the lock is extensible, the Token Vesting Lock smart contract continuously releases certain number of tokens over a predefined duration.

The project has one Solidity file for the Token Lock Smart Contract, the token-lock.sol file that contains about 153 lines of Solidity code, and one Solidity file for the Token Vesting Lock smart contract, the token-vesting-lock.sol that contains about 145 lines of solidity code. We manually reviewed each line of code in the smart contract.

Methodology:

Blockchain Consilium manually reviewed the smart contract line-by-line, keeping in mind industry best practices and known attacks, looking for any potential issues and vulnerabilities, and areas where improvements are possible.

We also used automated tools like slither for analysis and reviewing the smart contract. The raw output of these tools is included in the Appendix. These tools often give false-positives, and any issues reported by them but not included in the issue list can be considered not valid.

Classification / Issue Types Definition:

1. **High Severity:** which presents a significant security vulnerability or failure of the contract across a range of scenarios, or which may result in loss of funds.
2. **Moderate Severity:** which affects the desired outcome of the contract execution or introduces a weakness that can be exploited. It may not result in loss of funds but breaks the functionality or produces unexpected behaviour.
3. **Low Severity:** which does not have a material impact on the contract execution and is likely to be subjective.

The smart contract is considered to pass the audit, as of the audit date, if no high severity or moderate severity issues are found.

Attacks & Issues considered while auditing

In order to check for the security of the contract, we reviewed each line of code in the smart contract considering several known Smart Contract Attacks & known issues.

- **Overflows and underflows:**

  An overflow happens when the limit of the type variable uint256 , 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

For instance, if we want to assign a value to a uint bigger than 2 ** 256 it will simple go to 0—this is dangerous.

On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0.For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1.

This is quite dangerous. This contract **DOES** check for overflows and underflows, using **OpenZeppelin's** *SafeMath* for overflow and underflow protection.

- **Reentrancy Attack:**
  One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. This class of bug can take many forms, and both of the major bugs that led to the DAO's collapse were bugs of this sort.

  This smart contract follows *checks-effects-interactions* pattern and thus *is not found vulnerable* to re-entrancy attack.

- **Replay attack:**
  The replay attack consists of making a transaction on one blockchain like the original Ethereum's blockchain and then repeating it on another blockchain like the Ethereum's classic blockchain. The ether is transferred like a normal transaction from a blockchain to another. Though it's no longer a problem because since the version 1.5.3 of *Geth* and 1.4.4 of *Parity* both implement the attack protection EIP 155 by Vitalik Buterin.
  So the people that will use the contract depend on their own ability to be updated with those programs to keep themselves secure.

- **Short address attack:**
  This attack affects ERC20 tokens, was discovered by the Golem team and consists of the following:

  A user creates an Ethereum wallet with a trailing 0, which is not hard because it's only a digit. For instance: `0xiofa8d97756as7df5sd8f75g8675ds8gsdg0`
  Then he buys tokens by removing the last zero:
  Buy 1000 tokens from account `0xiofa8d97756as7df5sd8f75g8675ds8gsdg`. If the contract has enough amount of tokens and the buy function doesn't check the length of the address of the sender, the Ethereum's virtual machine will just add zeroes to the transaction until the address is complete.

The virtual machine will return 256000 for each 1000 tokens bought. This is abug of the virtual machine.

Here is a **fix for short address attacks**

```
modifier onlyPayloadSize(uint size) {
    assert(msg.data.length >= size + 4);
    _;
}
function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) {
    // do stuff
}
```

This contract is not an ERC20 Token, thus checks for erc20 short address attacks are not needed.

You can read more about the attack here: ERC20 Short Address Attacks.

- ## Approval Double-spend

ERC20 Standard allows users to approve other users to manage their tokens, or spend tokens from their account till a certain amount, by setting the user's allowance with the standard `approve` function, then the allowed user may use `transferFrom` to spend the allowed tokens.

Hypothetically, given a situation where Alice approves Bob to spend 100 Tokens from her account, and if Alice needs to adjust the allowance to allow Bob to spend 20 more tokens, normally – she'd check Bob's allowance (100 currently) and start a new `approve` transaction allowing Bob to spend a total of 120 Tokens instead of 100 Tokens.

Now, if Bob is monitoring the Transaction pool, and as soon as he observes new transaction from Alice approving more amount, he may send a `transferFrom` transaction spending 100 Tokens from Alice's account with higher gas price and do all the required effort to get his spend transaction mined before Alice's new approve transaction.

Now Bob has already spent 100 Tokens, and given Alice's approve transaction is mined, Bob's allowance is set to 120 Tokens, this would allow Bob to spend a total of 100 + 120 = 220 Tokens from Alice's account instead of the allowed 120 Tokens. This exploit situation is known as Approval Double-Spend Attack.

A potential solution to minimize these instances would be to set the non-zero allowance to 0 before setting it to any other amount.

It's possible for `approve` to enforce this behaviour without interface changes in the ERC20 specification:

```
if ((_value != 0) && (approved[msg.sender][_spender] != 0)) return false;
```

However, this is just an attempt to modify user behaviour. If the user does attempt to change from one non-zero value to another, the double spend might still happen, since the attacker may set the value to zero by already spending all the previously allowed value before the user's new approval transaction.

If desired, a non-standard function can be added to minimize hassle for users. The issue can be fixed with minimal inconvenience by taking a change value rather than a replacement value:

```
function increaseAllowance (address _spender, uint256 _addedValue)
returns (bool success) {
  uint oldValue = approved[msg.sender][_spender];
  approved[msg.sender][_spender] = safeAdd(oldValue, _addedValue);
  return true;
}
```

Even if this function is added, it's important to keep the original for compatibility with the ERC20 specification.

This contract is not an ERC20 Token, thus checks for approval-doublespend are not needed.

For more, see this discussion on GitHub:
https://github.com/ethereum/EIPs/issues/20#issuecomment263524729

Issues Found & Informational Observations

High Severity Issues

No high severity issues were found in the smart contract.

Moderate Severity Issues

No moderate severity issues were found in the smart contract.

Low Severity Issues

No low severity issues were found in the smart contract.

Line by line comments

token-lock.sol

- Line 1:
  The compiler version is specified as 0.6.11, this means the code can be compiled with solidity compilers with 0.6.11 only, the latest compiler version at the time of auditing is 0.7.5.

- Lines 3 to 33:
  SafeMath library is included to check for underflow and overflows.

- Lines 34 to 111:
  Ownable contract is implemented to provide basic access control for transferring out other tokens from this smart contract.

- Lines 113 to 119:
  Token interfaces is included to interact with ERC20 tokens.

- Lines 121 to 153:
  TokenLock contract is implemented, inheriting from Ownable contract, this contract allows owners to claim any ERC20 Tokens from this smart contract, including Legacy ERC20 Tokens, once unlock time has reached, the lock can be extended by owner within the range of MAX_EXTENSION_ALLOWED duration.

token-vesting-lock.sol

- Line 1:
  The compiler version is specified as 0.6.11, this means the code can be compiled with solidity compilers with 0.6.11 only, the latest compiler version at the time of auditing is 0.7.5.

- Lines 5 to 33:
  SafeMath library is included to check for underflow and overflows.

- Lines 35 to 74:
  Ownable contract is implemented to provide basic access control for transferring out other tokens from this smart contract.

- Lines 76 to 84:
  Token interfaces is included to interact with ERC20 tokens.

- Lines 86 to 145:
  TokenVestingLock is implemented inheriting from Ownable contract, this contract allows owner to claim DYP Tokens which are vested to owner’s wallet over a predefined duration, once the lock duration is over, owner can transfer out remaining tokens from this smart contract, if any.

Appendix

Smart Contract Summary

token-lock.sol:

- Contract SafeMath (Most derived contract)
    - From SafeMath
        - add(uint256,uint256) (internal)
        - div(uint256,uint256) (internal)
        - mul(uint256,uint256) (internal)
        - sub(uint256,uint256) (internal)
- Contract Ownable
    - From Ownable
        - _transferOwnership(address) (internal)
        - claimOwnership() (public)
        - constructor() (internal)
        - isOwner() (public)
        - owner() (public)
        - transferOwnership(address) (public)
- Contract Token (Most derived contract)
    - From Token
        - transfer(address,uint256) (external)
- Contract LegacyToken (Most derived contract)
    - From LegacyToken
        - transfer(address,uint256) (external)
- Contract TokenLock (Most derived contract)
    - From Ownable
        - _transferOwnership(address) (internal)
        - claimOwnership() (public)
        - constructor() (internal)
        - isOwner() (public)
        - owner() (public)

        - transferOwnership(address) (public)
    - From TokenLock
        - claim(address,address,uint256) (external)
        - claimLegacyToken(address,address,uint256) (external)
        - constructor(uint256) (public)
        - extendLock(uint256) (external)
        - isUnlocked() (public)

token-vesting-lock.sol:

- Contract SafeMath (Most derived contract)
    - From SafeMath
        - add(uint256,uint256) (internal)
        - div(uint256,uint256) (internal)
        - mul(uint256,uint256) (internal)
        - sub(uint256,uint256) (internal)
- Contract Ownable
    - From Ownable
        - constructor() (public)
        - transferOwnership(address) (public)
- Contract Token (Most derived contract)
    - From Token
        - balanceOf(address) (external)
        - transfer(address,uint256) (external)
        - transferFrom(address,address,uint256) (external)
- Contract LegacyToken (Most derived contract)
    - From LegacyToken
        - transfer(address,uint256) (external)
- Contract TokenVestingLock (Most derived contract)
    - From Ownable
        - transferOwnership(address) (public)
    - From TokenVestingLock
        - claim() (external)
        - constructor() (public)

- getPendingUnlocked() (public)
- transferAnyERC20Tokens(address,address,uint256) (external)
- transferAnyLegacyERC20Tokens(address,address,uint256) (external)

Slither Results

token-lock.sol

```
> slither token-lock.sol

INFO:Detectors:
LegacyToken (token-lock.sol#117-119) has incorrect ERC20 function
interface:LegacyToken.transfer(address,uint256) (token-lock.sol#118)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#incorrect-erc20-interface
INFO:Detectors:
TokenLock.constructor(uint256) (token-lock.sol#129-132) uses timestamp for
comparisons
        Dangerous comparisons:
        - require(bool,string)(initialUnlockTime > now,Cannot set an unlock time
in past!) (token-lock.sol#130)
TokenLock.isUnlocked() (token-lock.sol#134-136) uses timestamp for comparisons
        Dangerous comparisons:
        - now > unlockTime (token-lock.sol#135)
TokenLock.extendLock(uint256) (token-lock.sol#138-142) uses timestamp for
comparisons
        Dangerous comparisons:
        - require(bool,string)(extendedUnlockTimestamp > now,Cannot set an unlock
time in past!) (token-lock.sol#139)
        - require(bool,string)(extendedUnlockTimestamp.sub(now) <=
MAX_EXTENSION_ALLOWED,Cannot extend beyond MAX_EXTENSION_ALLOWED period!) (token-
lock.sol#140)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-
timestamp
INFO:Detectors:
Parameter Ownable.transferOwnership(address)._newOwner (token-lock.sol#89) is not
in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
owner() should be declared external:
        - Ownable.owner() (token-lock.sol#57-59)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (token-lock.sol#89-92)
claimOwnership() should be declared external:
        - Ownable.claimOwnership() (token-lock.sol#97-100)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-
function-that-could-be-declared-external
INFO:Slither:token-lock.sol analyzed (5 contracts with 46 detectors), 8 result(s)
found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and
Github integration
```

token-vesting-lock.sol:

```
> slither token-vesting-lock.sol

INFO:Detectors:
LegacyToken (token-vesting-lock.sol#82-84) has incorrect ERC20 function
interface:LegacyToken.transfer(address,uint256) (token-vesting-lock.sol#83)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#incorrect-erc20-interface
INFO:Detectors:
Reentrancy in TokenVestingLock.claim() (token-vesting-lock.sol#113-122):
        External calls:
        - contractBalance = Token(tokenAddress).balanceOf(address(this)) (token-
vesting-lock.sol#115)
        -
require(bool,string)(Token(tokenAddress).transfer(owner,amountToSend),Could not
transfer Tokens.) (token-vesting-lock.sol#120)
        State variables written after the call(s):
        - lastClaimedTime = now (token-vesting-lock.sol#121)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
TokenVestingLock.claim() (token-vesting-lock.sol#113-122) uses timestamp for
comparisons
        Dangerous comparisons:
        - contractBalance < pendingUnlocked (token-vesting-lock.sol#117)
        -
require(bool,string)(Token(tokenAddress).transfer(owner,amountToSend),Could not
transfer Tokens.) (token-vesting-lock.sol#120)
TokenVestingLock.transferAnyERC20Tokens(address,address,uint256) (token-vesting-
lock.sol#135-138) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(tokenContractAddress != tokenAddress || now >
deployTime.add(lockDuration),Cannot transfer out locked tokens yet!) (token-
vesting-lock.sol#136)
TokenVestingLock.transferAnyLegacyERC20Tokens(address,address,uint256) (token-
vesting-lock.sol#141-144) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(tokenContractAddress != tokenAddress || now >
deployTime.add(lockDuration),Cannot transfer out locked tokens yet!) (token-
vesting-lock.sol#142)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-
timestamp
INFO:Detectors:
Constant TokenVestingLock.tokenAddress (token-vesting-lock.sol#92) is not in
UPPER_CASE_WITH_UNDERSCORES
Constant TokenVestingLock.tokensLocked (token-vesting-lock.sol#95) is not in
UPPER_CASE_WITH_UNDERSCORES
Constant TokenVestingLock.lockDuration (token-vesting-lock.sol#98) is not in
UPPER_CASE_WITH_UNDERSCORES
Constant TokenVestingLock.unlockRate (token-vesting-lock.sol#101) is not in
UPPER_CASE_WITH_UNDERSCORES
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (token-vesting-lock.sol#69-73)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-
function-that-could-be-declared-external
INFO:Slither:token-vesting-lock.sol analyzed (5 contracts with 46 detectors), 10
result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and
Github integration
```