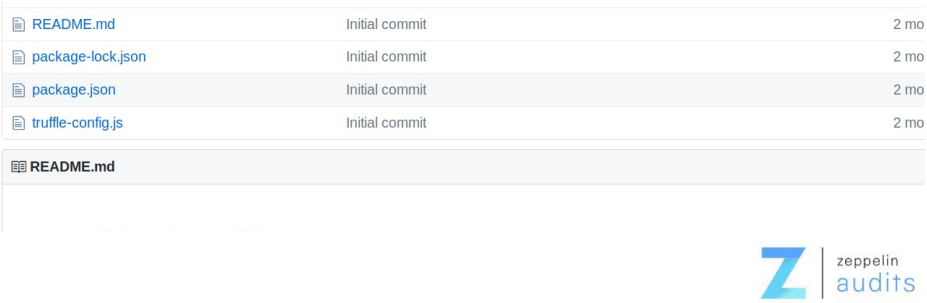
Zeppelin Audit

Srinjoy Chakravarty
Mar 16, 2018 · 11 min read



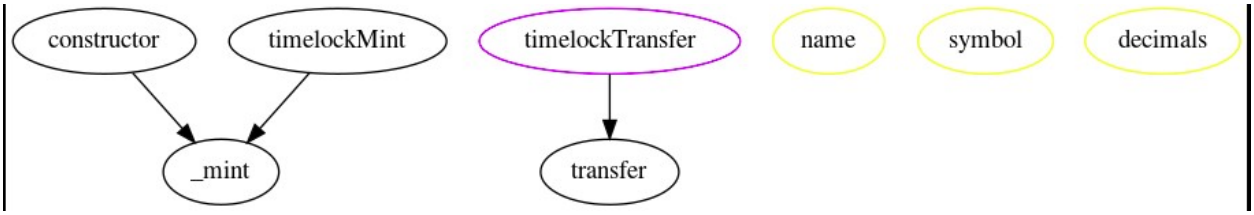
The Zeppelin team asked me to review and audit a ‘Token’ and ‘TokenDistributor’ contract. I looked at the code and now published my results.

Disclaimer: This is not a professional Zeppelin accredited security audit.

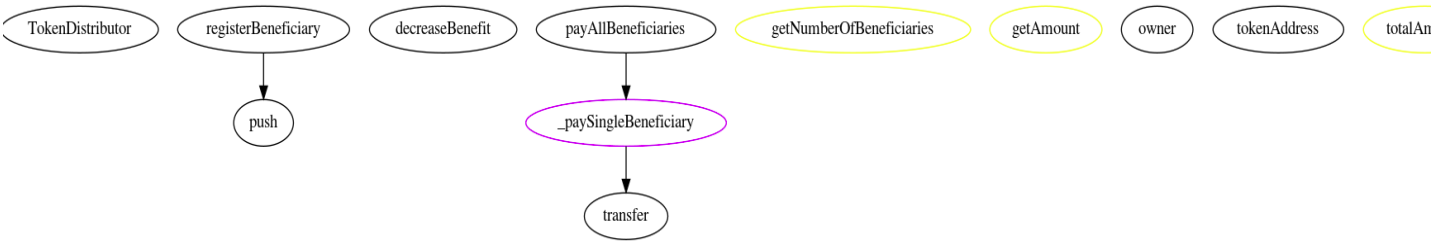
Here is my assessment and recommendations, in order of importance.

Token Contract Flow Visualizations (using Solgraph)

Token.sol



TokenDistributor.sol



Severity: High

Access Control: ‘timelockMint’ can be called by anyone

As per the business rule, minting of tokens through a timelock contract should only be done by the owner of the ‘Token’ contract with a call to the ‘timelockMint’ function. However neither the ‘onlyOwner’ modifier or require(msg.sender == owner) is used.

Access Control: ‘_paySingleBeneficiary’ can be called by anyone

The _paySingleBeneficiary can be called directly by non contract-owners as it is not protected by the modifier ‘onlyOwner’. The function is also missing the ‘private’ keyword as the comments suggests it should be.

Denial of Service: Gas Limit Risk on Long Loop in ‘payAllBeneficiaries()’

The payAllBeneficiares function loops over a dynamically sized mapping. Since the data structure is uncapped in size, Ethereum’s in-built maximum block gas limit of 21,000 can be reached after a certain number of iterations, stalling at an undesirable / misleading state and rendering the smart contract non-functional.

Inflation Bug: User balances unverified before ‘timeLock’

There is no check to see that the user calling the timelockTransfer is locking more tokens than they have.

Severity: Medium

Access Control: Deprecated constructor allows unauthorized ownership

The old constructor syntax is used whereby the function has the same name as the contract

```
contract TokenDistributor {
       function TokenDistributor(address tokenAddress) public {
```

Unlike the old method, the constructor function does not need to share the same name as the Contract. Instead the 'Constructor' keyword should be used as follows:

```
X > function TokenDistributor(address tokenAddress) public {
✓ > constructor(address tokenAddress) public {
```

Explanation

This is a major security issue leak as anybody can call the constructor since it is a public function. An attacker can effectively overwrite the existing owner and take ownership of the contract. As owner of the contract, the attacker can access any method (as the condition in the modifier is met) and can therefore create havoc by registering beneficiaries, decreasing the benefits of existing beneficiaries etc.

The ‘require’ condition only ensures that the contract is not assigned to the null address (i.e. burned) so that the contract is always reachable. It does not prevent other addresses from becoming the owner of the contract.

```
require(tokenAddress !=address(0));
```

Mitigation

Use the ’constructor’ keyword to write the constructor instead of a function.

```
constructor(address tokenAddress) public {
```

Contracts are created by sending a transaction with an empty ‘to’ field. This data is interpreted as EVM code which includes the constructor function. After being run, the return value which is used as the newly created on-chain contract code will not include the constructor. From here on

For every transaction, the compiler will check at the beginning of the code to compare the first four bytes of transaction data with available function signatures (computed as bytes4(sha3("myFunction(uint256, bytes, address, ...)")). Since, the constructor will not be included in the list of function signatures, the same vulnerability cannot be exploited.

Arithmetic: Safemath directive attached to unit256 but not used

The arithmetic operations remain unsafe as the contract only includes the SafeMath library with the line ‘using SafeMath for uint256;’ but does not replace operators such as ‘-’ with their corresponding ‘.sub()’ function.

Arithmetic: Possible Integer underflow vulnerability in ‘DecreaseBenefit’

There is no check in place to ensure -= amount does not lead to a large positive balance. This means that there is nothing to ensure that the subtrahend is smaller than the minuend and preventing a subtraction resulting in an underflow. For example, let’s assume the beneficiary only has 1 in his account (amount can be initialized as any number over 0 as per the condition in registerBeneficiary). If the decreaseBeneficiary function is called with 2 for this specific beneficiary address, then the return uin256 set to their account will be:

amountsByBeneficiary[beneficiary] -= amount
= beneficiaryAmount – amount
= 1 - 2
*= (2**256) – 1*

However since only the contract owner (admin) has access to call ‘DecreaseBenefit’ to update the variable's uint value, it is not vulnerable to intentional attack (although negligence can still occur).

Reentrancy: Possible recursive call to exhaust ‘_paySingleBeneficiary’

The '_paySingleBeneficiary' as suspected on day 3 manual review, has a re-entrancy vulnerability, as the external call to transfer tokens ('.transfer') happens before decrementing the total amount of token (state variable) to be distributed.

Residual Risk:
An attacker may create a malicious contract masquerading as an IERC20 token, and have the “transfer” function called back into the 'TokenDistribution' contract multiple times, draining some ETH each time.

Risk Reduction:
transfer() is generally safe against reentrancy attacks since it limits the code execution to 2300 gas (only enough to log event)

Recommendation:
Use "Optimistic Accounting" to put the whole state in order before invoking external contract by decrementing the totaAmount and then transferring to beneficiary

Denial of Service: No check to ensure only non-zero balances locked

TokenLock does not check that the user has atleast > 0 tokens to lock. Users can spam and create multiple newTimelock contracts despite having no amount of tokens to deposit.

Severity: Low

Solidity Breaking Changes v0.5: Missing ‘storage’ from constants

All state constants should contain the '**storage**' keyword since state variables declared at contract level don't need to change at runtime and in fact require persistent storage through a reserved slot in the EVM

Solidity Breaking Changes v0.5: Missing ‘calldata’ from parameters

The parameters ('beneficiary', 'amount', 'releaseTime') of external functions ('timelockTransfer', 'timelockMint') require a explicit data location of '**calldata**' so that the ABI encoder can properly pad byte arrays and strings.

Syntactical: Possible Typographical error in function call

The '**_mint**' call inside the constructor and 'timelockMint' function should be '**mint**' instead, as that is the precise function name that 'Token' inherits from ‘ERC20Mintable’.

Syntactical: Syntax ordering is erroneous

The function visibilty for 'registerBeneficiary' must be stated before the modifier.

Structural: State constant declared as local variable

‘base' should be declared as a state constant and not a local variable inside the constructor, since it is hardcoded.

Economics: Inefficient visibility set for ‘DecreaseBenefit’ function

It is inefficient and nearly twice as expensive to use ‘public’ for ‘DecreaseBenefit’ instead of ‘external’ as it is not called by any other code in the contract. This is because ‘public’ functions make the compiler generate code that allocates memory (expensive) to have array arguments referenced by internal calls, while ‘external’ functions read directly from calldata (cheap).

Economics: Gas consumption inefficiency in ‘payAllBeneficiaries’ loop

The for loop in 'payAllBeneficiaries()' uses 'beneficiaries.length' in the condition several times, causing every iteration of the loop to consume extra gas

Recommendation:
Holding the value of ‘beneficiaries.length’ in a local variable to be gas efficient, for example in ‘numOfBeneficiaries’.

Privacy: Missing ‘view’ keyword from multiple getters

The 'owner()' and 'tokenAddress()' getters should have its state mutability restricted to '**view**',
as they do not need to modify state variables.

Privacy: Missing visibility marker for state variables

The address array for beneficiaries does not have its 'state-visibility' stated. The ‘amountsByBeneficiary' mapping is also missing explicit state-visibility declaration. Both should be explicitly marked with a ‘private’ keyword.

EOL: Outdated compiler version usage

Solidity Compiler being used (v0.4.24) is 7 versions behind the current recommended version (v0.5.5)

Vulnerability:
The 0.4.24 compiler for this contract does not enforce the usage of the STATICCALL opcode on the Ethereum Virtual Machine even though a view function is called, so state modifications are possible through the use of invalid explicit type conversions.

Recommendation:
Use the statement ‘pragma solidity 0.5.5 ; ‘
(solc v 0.6.0 is not yet available for download even in the cutting edge developer version)

Severity: Informational

Timestamp Dependence: Token unlock prone to 30 second variation

The contract utilizes OpenZeppelin’s TokenTimelock contract which uses ‘block.timestamp’ to

But because timestamp is manipulatable, the miner could set a timestamp 2 seconds in the future and prevent the alarm from getting defused (another transaction is needed to stop the alarm)

Stylistic: Extraneous import call to ‘SafeMath.sol’

There is an unnecessary import of SafeMath.sol in 'Token' contract as it is not called/used again in the file.

Stylistic: Extraneous import call to ‘SafeMath.sol’

As per default configuration rules of the style guide, the _name, _symbol, and _decimals constants in the token contract need capitalization (for e.g. _NAME, _SYMBOL, _DECIMALS)

Stylistic: Newline characters not in line with formatting best practice

The TokenDistributor.sol contract definition needs a newline character between the previous imports to comply with (two-line-top-level separator standard)

Missing Functionality: The contract is not set up to receive ether

The constructor does not have the 'payable' keyword. This may be by design as it is not a crowdsale contract and there is no requirement for ether to ever be sent to it. However this is a decision that business heads should review, as they may forsee a need for this to change down the road.

Chronology: Functions not written in order of visibility

The 'payAllBeneficiaries' function and 'Getters' needs to be shuffled before 'decreaseBenefit' and '_paySingleBeneficiary' to respect best-practice where functions are supposed to be written in descending order of visibility (External > Public > Internal > Private)

Chronology: Getters not written in order of read access

Getters rearranged so that getters marked with the 'view' keyword (substitute for depracated 'constant') are written after getters with no 'view' keyword specified as it follows the ascending order of state restraints (returns > view returns > pure returns)