OPU Labs Contract Audit
v2.4

New Alchemy

August, 2018

Introduction

During August, 2018, Opu Labs engaged New Alchemy to audit the smart contracts for Opu Coin. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts, finding differences between the contracts' implementation and their behaviour as described in public documentation, and finding any other issues with the contracts that may impact their trustworthiness. Opu Labs provided New Alchemy with access to the relevant source code and whitepapers.

The audit was performed over three days. This document describes the issues discovered in the audit.

Note: The initial version of this document was provided to Opu Labs who then made various changes to their smart contract source code based upon New Alchemy's findings. This document now consists of the original and unchanged audit report (v1.0) overlaid with re-test results (v2.0). Re-test results are reflected in each issue title as 'Fixed', 'Partially Fixed', 'Not Fixed' or 'Informational'. Supporting re-test commentary is attached to a bolded 'Re-test v2.0:' prefix and placed at the end of the relevant section. The bulk of the re-test content relates to an examination of individual issues, along with brief comments in the executive summary and files audited sections. All figures are from the initial version of the document.

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 bugfree status. The audit documentation is for discussion purposes only.

Executive Summary

This audit identifies two critical bugs in the source that cause the behavior of the contracts to differ substantially from its documented behavior:

1. An arithmetic error causes bonus tokens never to be distributed from the `Vesting` contract.
2. An implicit function shadowing error causes the token's total supply to always read as zero.

Additionally, we identified a handful of less critical oversights and "near-misses." Many of the issues identified in the code are simple logic errors that could be identified with additional unit test coverage.

OPU's ICO is "trustful," which has implications for how participants in OPU's ICO should model their risk exposure. We outline these risks in detail in the General Discussion section.

Re-test v2.0: New Alchemy has discussed the prior audit results with Opu Labs, advised on efficient and effective mitigation approaches, and inspected the resulting smart contract code provided for re-testing. New Alchemy has concluded:

- **All non-informational issues isolated to the Opu Labs contracts have been addressed and the associated risks effectively mitigated.**
- The third party library in use, OpenZeppelin, is out of date by 6 weeks.
- A minor issue labeled 'Informational' below is related to short-address attacks which are contentious among developers, the community is trending towards accepting rather than mitigating this risk, and mitigation can only occur at the source outside of the audited contract code. New Alchemy considers Opu Labs' approach appropriate and sufficient.

Files Audited

The code reviewed by New Alchemy is in the GitHub repository OpuLabs/OpuCoin at commit hash `242d081927c907314b68075a1a8d5ef690efa5c9`.

New Alchemy's audit was additionally guided by the Opu Labs whitepaper. The sha512 checksum of the whitepaper at the time of the audit was:

```
80ea3a31d1caa4b43247e0b20e573ba73f44f370905c2dcf4c9f8bb6bf4c2969
6ca0aae32bee5ff10d2f3f4047955a923bd2897a0b33ebc13b8a78aa4c3d7171
```

Re-test v2.0: The whitepaper link appears to be dead. The latest link is https://ico.opu.ai/whitepaper

The scope of this audit did not include third-party dependencies. The following files in the source tree were identified as thrid-party code:

- `Ownable.sol`
- `BasicToken.sol`
- `ERC20Basic.sol`
- `ERC20.sol`
- `MintableToken.sol`
- `StandardToken.sol`

Each third-party source file was vendored from OpenZeppelin at the following commit hash:

`23074676c4bc7099d59b2f5344258173361e93ea`

General Discussion

The audited source code compiles without warnings or errors with `solc` version 0.4.24.

The source code contained 22 truffle unit tests. New Alchemy was unable to get the unit tests to run cleanly, either because of truffle version incompatibilities or because of bugs in the unit tests themselves. In either case, the coverage of the unit tests is likely insufficient, as greater unit test coverage would have uncovered the arithmetic bug in `Vesting.sol` that prevents bonus tokens from being disbursed, as the bug is easily triggered with a variety of inputs. It is New Alchemy's recommendation that the unit tests be updated to test for these cases.

Of particular interest to those purchasing OpuLabs' tokens is the fact that the **token distribution is performed manually**, and therefore there is no guarantee that tokens are distributed as described in the documentation.

The scope of this audit does not extend to cover OpuLabs' key management practices, and therefore this audit generally assumes that those practices will be sufficient to prevent the compromise of any addresses controlled by OpuLabs.

Hacks that lead to key disclosure, like the recent Bancor hack, are not uncommon, but they violate the soundness of the reviewed contracts, since all of the reviewed code implicitly assumes that the owner address(es) of each contract are solely controlled by OpuLabs.

In short, **the contracts reviewed in this audit are no more secure than OpuLabs' key management practices, and those practices have not been reviewed by New Alchemy.**

Contract / Whitepaper / Website Token Coherence

This section describes how accurately the values from the whitepaper are implemented in the actual contracts. Contracts should aim to implement as closely as possible the various descriptions found in the whitepaper and website.

The documentation regarding the OPU token supply is on page 50 of the whitepaper.

Item name	White Paper Value	Smart contract Value	Discrepancy (%)
total supply	2,700,000,000	2,700,000,000	**0%**
ICO tokens	1,350,000,000	1,350,000,000	**0%**
rewards tokens	189,000,000	189,000,000	**0%**
partner tokens	297,000,000	297,000,000	**0%**
cold storage tokens	189,000,000	189,000,000	**0%**
team tokens	675,000,000	675,000,000	**0%**

Of particular note is that the `mintForRedemption` function in `Allocation.sol` allows the `backend` address (presumably controlled by OpuLabs) to mint an arbitrary number of tokens to any address without any additional checks. Consequently, the total supply and distribution of tokens is entirely contingent on trusting the bearer of the key of the `backend` address to mint tokens as described in the documentation.

The design of the `Allocation` contract requires that an engineer, at the direction of Opu Labs, call the `allocate()` and `allocateIntoHolding()` functions with the correct amounts that correspond to contributions that may or may not have been made on the Ethereum blockchain. In other words, contibutors must entirely trust Opu Labs to distribute the correct number of tokens to the correct addresses; the distribution of tokens is *not* performed automatically (i.e. by a smart contract) during the sale. A "fat-finger error" on the part of the engineers distributing the tokens could lead to unrecoverable tokens. Additionally, the model of sending one transaction to the `Allocation` contract per contribution is needlessly wasteful of gas; it's possible to batch transactions to amortize the per-transaction 21,000 base gas cost.

Critical Issues

Fixed: 1. Vesting.sol: Bonus Tokens Unintentionally Locked

There is an arithmetic bug on line 61 of `Vesting.sol` that prevents any tokens in `additionalHoldingPool` from ever being disbursed.

```
if (!holdings[msg.sender].updatedForFinalization) {
    holdings[msg.sender].updatedForFinalization = true;
    holdings[msg.sender].tokensRemaining = (holdings[msg.sender].tokensRemaining).add(
    // bug on following line:
    (holdings[msg.sender].tokensCommitted).div(totalTokensCommitted).mul(additionalHoldingPool)
    );
}
```

The issue is that the expression `holdings[x].tokensCommitted` will always be less than `totalTokensCommitted`, since `totalTokensCommitted` should be the sum of all `holdings[...].tokensCommitted`. Thus, this division operation will always yield zero, and thus the post-multiplication by `additionalHoldingPool` will also yield zero. Consequently, `tokensRemaining` is not adjusted at all by the statement beginning on line 60, and therefore none of the `additionalHoldingPool` tokens are ever paid out. Moreover, there is no alternative means of withdrawing those tokens from the contract, and consequently they are locked in the contract in perpetuity.

Re-test v2.0: Opu Labs reversed the order of the multiply and divide to multiply first.

***Fixed*: 2. OPUCoin.sol: Shadowing of Total Supply**

The `OPUCoin` contract in `OPUCoin.sol` declares a public contract variable `totalSupply`, which implicitly declares a `totalSupply()` accessor function that shadow the same function from the `MintableToken` contract. Consequently, reading `totalSupply()` from the OPUCoin contract always returns 0 rather than the real total supply, which breaks ERC20 conformance.

Re-test v2.0: Opu Labs removed the unused `totalSupply` variable from OPUCoin.sol.

Moderate Issues

3. ***Fixed*: `Token.finishMinting()` Never Called**

The `Allocation` contract that owns the `OPUCoin` token never calls `finishMinting()`. Consequently, any minting capabilities present in the `Allocation` contract present an ongoing security liability with respect to the total supply of tokens. In particular, `Allocation.mintForRedepmtion()` allows the `backend` address to mint tokens in perpetuity. Thus, if the private key for the `backend` address is compromised (either through key disclosure or factoring of a key generated with a weak entropy source), the token supply can be manipulated up to the 2.7 billion token limit.

Re-test v2.0: Opu Labs added a new function, `finishMinting`, calling the MintableToken.finishMinting on the OPUCoin contract, allowing the contract creator to finish minting.

Minor Issues

4. *Not Fixed*: OpenZeppelin: Improper Vendoring

The OpenZeppelin project recommends vendoring their code through npm, rather than copying the source files out of their source tree. Vendoring specific source files makes it more difficult for auditors to determine the lineage of the code in question. Additionally, the vendored code needs to be manually inspected for changes after-the-fact.

5. *Fixed*: Vesting.sol: Undesirable State Re-initialization

The `claimTokens()` function in `Vesting.sol` re-initializes `holdings[msg.sender]` after deleting that mapping entry. Consequently, it is possible for a `Holding` entry to be present in `holdings` with a false `isValue` field but a non-zero `batchesClaimed` count.

```
if (periodsPassed >= totalPeriods) {
    tokensToRelease = holdings[msg.sender].tokensRemaining;
    delete holdings[msg.sender]; // <-- clear holdings[msg.sender]
} else {
    tokensToRelease = tokensPerBatch.mul(batchesToClaim);
    holdings[msg.sender].tokensRemaining = \
        (holdings[msg.sender].tokensRemaining).sub(tokensToRelease);
}

// holdings[msg.sender] re-initialized here:
holdings[msg.sender].batchesClaimed = holdings[msg.sender] \
                      .batchesClaimed.add(batchesToClaim);
```

Although this oversight does not appear to cause any subsequent logic errors, it would be safer if the code never wrote to deleted map entries. Additionally, writing a non-zero value to a previously-deleted map entry will waste gas unnecessarily.

Re-test v2.0: Opu Labs no longer re-initializes state.

6. *Fixed*: Allocation.sol / Vesting.sol: Fragile Vesting Initialization

The `initializeVesting()` function in the `Vesting` contract that is called by the `Allocation` contract can only be called once per user. Consequently, if a bug in the code that calls `allocateIntoHolding()` were to under-allocate tokens to a particular address, it would be impossible to add additional tokens to that addresses `holdings` entry in `Vesting.sol`.

Allowing the `allocateIntoHolding()` functionality in `Allocation` to increase token amounts would improve the robustness of the code in the face of operator error.

Re-test v2.0: Opu Labs added logic to handle the case of increasing allocation in `_initializeVesting()`

7. ***Fixed***: Vesting.sol: Vesting Allowed after Finalization

The `_initializeVesting` function allows new entries to `holdings` to be added after `finalizeVestingAllocation()` has been called. In other words, tokens can be added to the `Vesting` contract after the vesting period has started. `_initializeVesting()` should require that `vestingStarted` not be set.

Re-test v2.0: Opu Labs added `require( !vestingStarted );` to `_initializeVesting()`

8. ***Informational*: ERC20 Double-Spend Attack Vulnerability**

The standard ERC20 interface, implemented in `ERC20Token`, has a design flaw: if some user Alice wants to change the allowance granted to another user Bob, then Alice checks if Bob has already spent his allowance before issuing the transaction to change Bob's allowance. However, Bob can still spend the original allowance before the transaction changing the allowance is mined, which thus allows Bob to spend both the pre-change and post-change allowances[1]. In order to have a high probability of successfully spending the pre-change allowance after the victim has verified that it is not yet spent, the attacker to waits until the transaction to change the allowance is issued, then issues a spend transaction with an unusually high gas price to ensure that the spend transaction is mined before the allowance change. More details on this flaw are available at https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/ and https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729.

Due to this flaw, safely using the ERC20 `approve()` interface to change allowances requires that allowances only change between zero and non-zero. This restriction allows Alice to safely change Bob's allowance by first setting it to zero, waiting for that transaction to be mined, verifying that Bob didn't spend its original allowance, and then finally setting the allowance to the new value. Requiring this sequence of operations in `approve()` would violate the ERC20 standard, though users can still take this approach even without changes in the `approve()` implementation.

There are two alternative approaches that contracts can take to mitigate this flaw:

- To make it more convenient to change allowances, New Alchemy recommends providing `increaseApproval()` and `decreaseApproval()` functions that add or subtract to the existing allowances rather than overwriting them. This effectively moves the check of whether Bob has spent its allowance to the time that the transaction is mined, removing Bob's ability to double-spend. Clients that are aware of this non-standard interface can use it rather than `approve`; using `approve()` remains open to abuse. This is approach is implemented by the current version of the OpenZeppelin library[2], and is therefore already implemented by Opu Labs' code.

- Only allow `approve()` to change allowances between zero and non-zero and don't allow multiple changes to a user's allowance in the same block. In order to change Bob's allowance, Alice must first set it to zero, wait until that transaction is mined, verify that the original allowance was not spent, then finally set the allowance to the new value. Requiring this sequence of operations by implementing restrictions in `approve()` would violate the ERC20 standard, though users can still take this approach even without changes in the `approve()` implementation. The OpenZeppelin code that OPU Labs' is using does not implement this mitigation.

Since both approaches are outside of the ERC20 standard, both approaches require user cooperation to work properly. Accordingly, Opu Labs should provide documentation advising its users on how they should manage other users' allowances.

[1] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approve
[2] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

9. *Fixed*: Lack of Two-phase Ownership Transfer

In contracts that inherit the common `Ownable` contract from the OpenZeppelin project[3], a contract has a single owner. That owner can unilaterally transfer ownership to a different address. However, if the owner of a contract makes a mistake in entering the address of an intended new owner, then the contract can become irrecoverably unowned. This is analogous to the well-known "Locked Ether" scenario resulting in several million US dollars (over 7000 Ether) irretrievably locked at address 0x0[4].

In order to prevent this high-impact scenario, New Alchemy recommends implementing a two-phase ownership transfer. In this model, the original owner designates a new owner but does not actually transfer ownership. The new owner then accepts ownership and completes the transfer. This can be implemented as follows:

```
contract Ownable {
    address public owner;
    address public newOwner

    event OwnershipTransferred(address indexed oldOwner, address indexed newOwner);

    function Ownable() public {
    owner = msg.sender;
    newOwner = address(0);
    }

    modifier onlyOwner() {
    require(msg.sender == owner);
    _;
    }

    function transferOwnership(address _newOwner) public onlyOwner {
    require(address(0) != _newOwner);
    newOwner = _newOwner;
    }

    function acceptOwnership() public {
    require(msg.sender == newOwner);
    OwnershipTransferred(owner, msg.sender);
    owner = msg.sender;
    newOwner = address(0);
    }
}
```

Re-test v2.0: Opu Labs added the above suggested functionality.

[3] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol
[4] https://etherscan.io/address/0x0000000000000000000000000000000000000000

10. ***Informational***: Lack of Short-Address Attack Protection

Some Ethereum clients may create malformed messages if a user is persuaded to call a method on a contract with an address that is not a full 20 bytes long. In such a "short-address attack"[5], an attacker generates an address whose last byte is 0x00, then sends the first 19 bytes of that address to a victim. When the victim makes a contract method call, it appends the 19-byte address to `msg.data` followed by a value. Since the high-order byte of the value is almost certainly 0x00, reading 20 bytes from the expected location of the address in `msg.data` will result in the correct address. However, the value is then left-shifted by one byte, effectively multiplying it by 256 and potentially causing the victim to transfer a much larger number of tokens than intended. `msg.data` will be one byte shorter than expected, but due to how the EVM works, reads past its end will just return 0x00.

This attack effects methods that transfer tokens to destination addresses, where the method parameters include a destination address followed immediately by a value. In the Opu Labs' contracts, such methods include:

```
Allocation.optAddressIntoHolding(address, uint256);
Allocation.allocate(address, uint256, uint256);
ColdStorage.initializeHolding(address, uint256);
OPUCoin.approve(address, uint256);
OPUCoin.transfer(address, uint256);
OPUCoin.transferFrom(address, address, uint256);
OPUCoin.mint(address, uint256);
OPUCoin.decreaseApproval(address, uint256);
OPUCoin.increaseApproval(address, uint256);
Vesting.initializeVesting(address, uint256);
```

While the root cause of this flaw is buggy serializers and how the EVM works, it can be easily mitigated in contracts. When called externally, an affected method should verify that `msg.data.length` is *at least* the minimum length of the method's expected arguments (for instance, `msg.data.length` for an external call to `OPUCoin.transfer()` should be at least 68: 4 for the hash, 32 for the address (including 12 bytes of padding), and 32 for the value; some clients may add additional padding to the end). This can be implemented in a modifier. External calls can be detected in the following ways:

- Compare the first four bytes of `msg.data` against the method hash. If they don't match, then the call is internal and no short-address check is necessary.
- Avoid creating `public` methods that may be subject to short-address attacks; instead create only `external` methods that check for short addresses as described above. `public` methods can be simulated by having the external methods call `private` or `internal` methods that perform the actual operations and that do not check for short-address attacks.

Whether or not it is appropriate for contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. While it is New Alchemy's position that there is value in

[5] How to Find $10M Just by Reading the Blockchain https://blog.golemproject.net/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95

protecting users by incorporating low-cost mitigations into likely target functions, Opu Labs would not stand out from the community if they also choose not to do so.

Re-test v2.0: New Alchemy has discussed this issue with the Opu Labs team at length. As noted above, this is a contentious issue among smart contract developers, there are clear community trends towards accepting rather than mitigating this vulnerability, and there are external dependencies such as UI and tools that must be considered. Opu Labs has chosen to accept this risk at this time and consider future mitigation as their application evolves. New Alchemy considers this an entirely reasonable approach.

Line by Line Comments

This section lists comments on design decisions and code quality made by New Alchemy during the review. They are not known to represent security flaws.

Fixed: MintableToken.sol: Incorrect Supply Cap

```
 uint constant public supplyHardCap = 12 * 1e9 * 1e18;
```

The maximum total supply in the `MintableToken` contract is governed by the constant `supplyHardCap`, which is set to 12 billion, and not 2.7 billion as documented in the whitepaper.

Re-test v2.0: Opu Labs set the token supply to 2.7 billion; see SUPPLY_HARD_CAP in MintableToken.sol

Fixed: Vesting.sol:107: Unnecessary Ternary Expression

```
    updatedForFinalization: (_isFounder) ? (true) : (false),
```

The ternary expression on this line can simply be replaced with `_isFounder`.

Re-test v2.0: Opu Labs simplified the expression.

Fixed: Compiler Version Unfixed

Source files

`Allocation.sol` Line 1 `ColdStorage.sol` Line 1 `Migrations.sol` Line 1 `OPUCoin.sol` Line 1 `Ownable.sol` Line 6 `Token/BasicToken.sol` Line 6 `Token/ERC20.sol` Line 6 `Token/ERC20Basic.sol` Line 6 `Token/MintableToken.sol` Line 6 `Token/StandardToken.sol` Line 6 `Vesting.sol` Line 1 `math/SafeMath.sol` Line 6

It is recommended that the compiler version be fixed to a specific version before code is deployed.

Re-test v2.0: Opu Labs has pinned the compiler version to solc 0.4.24 in all files.

Fixed: Missing State Visibility Specifiers

Source files

- `Allocation.sol` Lines 23, 28, 30, 31, 32, 33, 34, 36, 37
- `ColdStorage.sol` Line 11
- `Token/BasicToken.sol` Lines 20, 22
- `Vesting.sol` Lines 11, 13, 15, 16, 17, 18, 21, 23, 24

State variables, like functions, should always include an explicit visibility specifier.

Re-test v2.0: Opu Labs now explicitly specifies visibility in each case.

Fixed: Improper Convention for Constants

Source files

- `Allocation.sol` Lines 28, 30, 31, 32, 33, 34
- `OPUCoin.sol` Lines 6, 7, 8
- `Token/MintableToken.sol` Line 21
- `Vesting.sol` Lines 15, 16, 17, 18

Conventionally, constant names are in capitalized SNAKE_CASE.

Re-test v2.0: Opu Labs now uses SNAKE_CASE where appropriate.

Fixed: Missing SafeMath

Source files

- `Allocation.sol` Lines 28, 30, 31, 32, 33, 34
- `Token/MintableToken.sol` Line 21
- `math/SafeMath.sol` Lines 26, 46, 53

To avoid integer overflows and underflows, use OpenZeppelin's SafeMath for all integer math, or include a comment explaining why the operation will never over/underflow.

Re-test v2.0: Opu Labs now uses SafeMath where appropriate.

Fixed: Improper Function Visibility Modifier Location

Source files

- `ColdStorage.sol` Line 28
- `Token/MintableToken.sol` Line 62
- `Vesting.sol` Lines 87, 92

Conventionally, visibility modifiers should be first in list of modifiers.

Re-test v2.0: Opu Labs moved the visibility modifiers ahead of others in each case.

Informational: Dependency on `now` and/or `block.timestamp`

Source files

- `ColdStorage.sol` Lines 32, 39
- `Vesting.sol` Lines 51, 53, 95

It is safest to avoid make time-based decisions in your business logic. Programmers should be aware that block timestamps are somewhat malleable to minors, and therefore special care must be taken to account for that malleability.

Fixed: Unconventional Function Ordering

Source files

- `ColdStorage.sol` Line 38

Convetionally, external functions come before public functions in declaration ordering.

Informational: Empty Block

Source files

- `OPUCoin.sol` Line 14

The source code contains empty block. It should be removed.

Fixed: Improper Compiler Version Specification for Syntax

Source files

- `OPUCoin.sol` Line 14

The `constructor` keyword is not available before solidity 0.4.22, but the code specifies compiler ˆ0.4.18.

Re-test v2.0: Opu Labs now specifies compiler version 0.4.24.

Fixed: Division before Multiplication

Source files

- `Vesting.sol` Line 61

Multiply before divide to avoid losing information due to integer math. (This issue results in critical bug 1.)

Re-test v2.0: Opu Labs now multiplies before divide.

Appendix A: Class Hierarchy

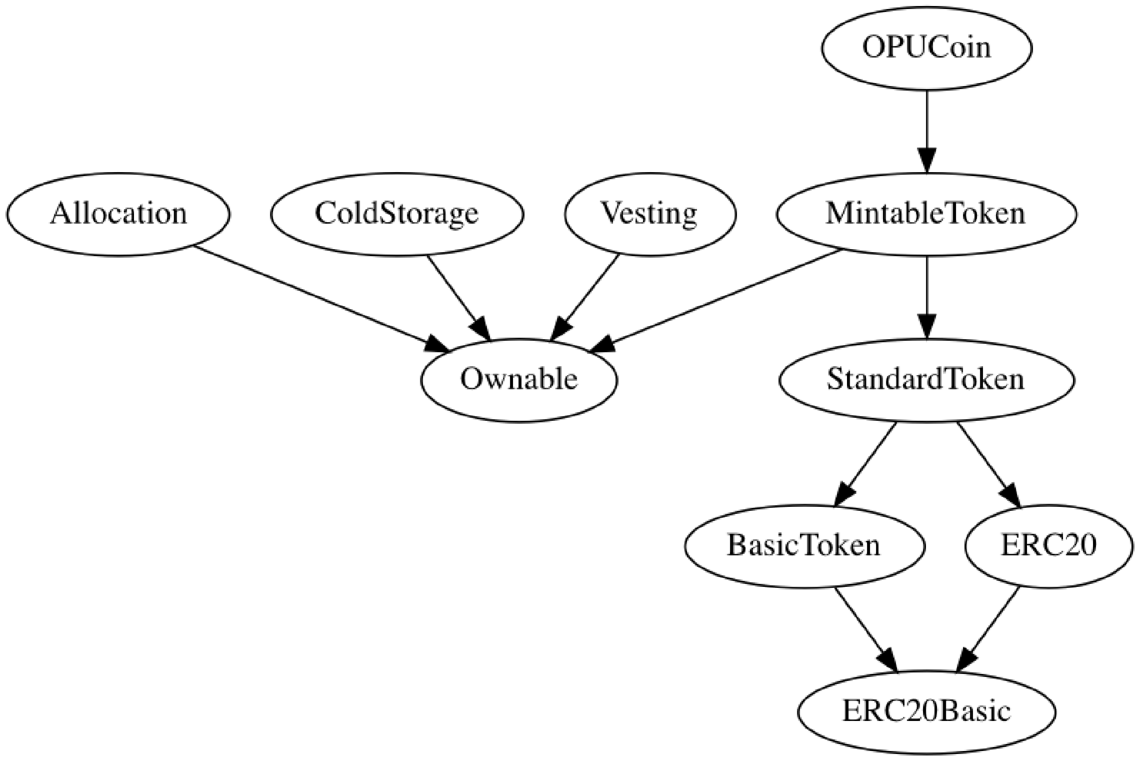

Figure 1: Class Hierarchy

Appendix B: Function Listing

This appendix lists every function definition along with its visibility specifier.

```
Name                                     Visibility
---------------------------------------------------
Allocation._allocateTokens               internal
Allocation.allocate                      public
Allocation.allocateIntoHolding           public
Allocation.checkCapsAndUpdate            internal
Allocation.emergencyPause                public
Allocation.emergencyUnpause              public
Allocation.finalizeHoldingAndTeamTokens  public
Allocation.holdTokens                    internal
Allocation.mintForRedemption             public
Allocation.optAddressIntoHolding         public
Allocation.vestTokens                    internal
BasicToken.balanceOf                     public
BasicToken.totalSupply                   public
BasicToken.transfer                      public
ColdStorage.claimTokens                  external
ColdStorage.initializeHolding            public
ERC20.allowance                          public
ERC20.approve                            public
ERC20.transferFrom                       public
ERC20Basic.balanceOf                     public
ERC20Basic.totalSupply                   public
ERC20Basic.transfer                      public
MintableToken.finishMinting              public
MintableToken.mint                       public
Ownable._transferOwnership               internal
Ownable.renounceOwnership                public
Ownable.transferOwnership                public
SafeMath.add                             internal
SafeMath.div                             internal
SafeMath.mul                             internal
SafeMath.sub                             internal
StandardToken.allowance                  public
StandardToken.approve                    public
StandardToken.decreaseApproval           public
StandardToken.increaseApproval           public
StandardToken.transferFrom               public
Vesting._initializeVesting               internal
Vesting.claimTokens                      external
Vesting.finalizeVestingAllocation        public
Vesting.initializeVesting                public
```

`Vesting.tokensRemainingInHolding`	`public`