Published in New Alchemy

New Alchemy Follow

Aug 10, 2018 · 30 min read · Listen

Save

ICOVO Smart Contract Audit

ICOVO engaged New Alchemy to audit their DAICOVO platform series of Smart Contracts. These interlocking contracts implement functionality related to ICO fundraising, token and specification management.

The engagement was highly technical in nature and focused on identifying security flaws in the design and implementation of the contracts, finding differences between the contract's implementation and their behavior as described in provided documentation, and finding any other issues with the contracts that may impact their trustworthiness. ICOVO provided New Alchemy with access to their two whitepapers along with the GitHub repository containing source code and an associated `README.md` file.

The audit was performed over 9 person-days. This document describes the issues, observations and supporting notes developed over the course of the audit.

The DAICOVO platform series of Smart Contracts encompasses nine highly custom contract source files, fourteen essentially open-source components, three sources of robust documentation and a series of supporting test cases. The DAICOVO platform contracts exhibited many positive characteristics including extensive reuse of well known libraries, very modularized source code and the provision of approximately 18 unit-level test cases that ran without issue.

However, the audit did uncover a variety of issues ranging from critical to minor. The more significant issues can be summarized as follows:

- A negative-outcome vote will stall the voting system.
- A loss of integer arithmetic precision defeats the intended raiseTap granularity.
- The whitepaper versioning contract has a logic bug that will result in stalls.
- Unbounded `for loops` may encounter issues with gas limits resulting in loss of functionality.
- An inadequate dependency management results in the loss of visibility into emerging vulnerabilities, fixes and improvements in reused open-source components.

The test cases were not strictly in scope but were utilized to help understand and explore specific findings. Significant portions of the contract package were derived from widely-used, reviewed and standardized OpenZeppelin contracts. As a result, the primary focus of the review and findings pertain to the custom overrides and additions contained in the DAICOVO code. The two whitepapers were considered primary documentation.

New Alchemy believes the way forward should include contract source corrections corresponding to the findings listed in this review, an improved dependency management and versioning system, and further test coverage development. The absence of a large number of critical findings is a positive leading indicator.

Re-test v2.0: New Alchemy has discussed the prior re-test results with ICOVO, advised on efficient and effective mitigation approaches, and inspected the resulting smart

- For the original five moderate issues, three have been fixed, one has not been fixed and one has been marked informational.

- For the original six minor issues, one has been fixed, four have not been fixed and one has been marked informational. Three of the not-fixed minor issues relate to weaknesses (ERC20 double spend, single-phase ownership transfer, and short address checks) that are frequently deployed unfixed.

Re-test v2.0: As the code currently stands, there is one moderate issue that remains not fixed and four minor issues that remain not fixed.

Files Audited

The code reviewed by New Alchemy is in the GitHub repository https://github.com/icovo/DAICOVO at commit hash `417b33a6159a21df73221dbdc942cbd73a9c2b06`.

The specific files making up the complete deployed package include the following:

```
https://github.com/icovo/DAICOVO/tree/417b33a6159a21df73221dbdc942cb
d73a9c2b06

crowdsale/Crowdsale.sol
7a51f754f6f2407cb3e52fb8dbfacd33be83a05c

crowdsale/distribution/Finalizabl...sol
c647b6b3573a0f686bf65024e69034fe19d296fd

crowdsale/emission/MintedCrowdsale.sol
754cd9da9c834179611e592a04995949e4122520

crowdsale/validation/Individually...sol
6b76a6758495af5e00f354c75c2ccc4dc3539762

crowdsale/validation/TimedCrowdsale.sol
30e68736b861f1a3c91117dcdbaebc206160acc8

crowdsale/validation/WhitelistedC...sol
ddb20032c78a0fbcb23f8bdedd04b6b7730deb7f

DAICOVO/DaicoPool.sol
```

```
DAICOVO/OVOToken.sol
0e61da7c3fa3f71f2442c8faf16646bd517fd8e8

DAICOVO/TimeLockPool.sol
dc3edac3b5b848d949e1ad6fb459c3dd9301995c

DAICOVO/TokenController.sol
c7e640c04d870df423a68cb1a36b03d56d6dd580

DAICOVO/TokenSaleManager.sol
d28deba98c725c1ea9f2a39634abcdd19d942a1e

DAICOVO/TokenSale.sol
741b946aab355891427e2c3f2ce12b742dc9979a

DAICOVO/Voting.sol
e1a230367b93ad43cee2a7f47a4a16c9522c83b1

DAICOVO/WhitepaperVersioning.sol
d933182be9a809676007e0b1f9369bf49b41817c

math/SafeMath.sol
4268957f43b3bc70c95e5cffbae96de8b349dad8

ownership/Ownable.sol
014271b54e0312e6f1848c58ab97b40f951884c4

token/ERC20/ERC20Interface.sol
9727768db5631995b88d76df4d0e29b48706e6ed

token/ERC20/ERC20Standard.sol
c35aaf379ef05470ba772b55d94e1e0a97b7b759

token/ERC223/ERC223Interface.sol
485ce71c088bd557794293b2e07eaab6c0a5572b

token/ERC223/ERC223ReceivingContract.sol
77d49dbd26942408383c219769a0dad62a1c71ba

token/ERC223/ERC223Standard.sol
8e3564a808f3fa66b2f028d6ad3e1da3dc3b1111

token/extentions/MintableToken.sol
a20b719c3f060b255a017abd755bc63d4dcdd11f
```

New Alchemy's audit was additionally guided by

Open in app
Get started

[^wp1]: shasum = 37ac996a0b9fbfb5eb9ab48d19b3f7fcfbd7865a [^wp2]: shasum = dd00bda08fc9966e2de40b3edca189a6a5d74df4

Re-test v2.0: New Alchemy has inspected the new code located in the same GitHub repository with a commit hash of `cc443f0332217d43ecc9ee3a3af849a3a2766ea1` along with a brief review of the revised documentation20180620_icovo_wp_service_en.pdf[^wp3] and 20180706_icovo_wp_tokensale_en.pdf[^wp4]

[^wp3]: shasum = 96a6d93440f5a6c1c1a3dd3eea7382f5715558ad [^wp4]: shasum = 5ddddd76c29ee45c18c827c404f97058de307d94

General Discussion

The DAICOVO platform implements a broad range of functionality through multiple interlocking smart contracts. These contracts implement custom logic to support token, sale, pool, voting and whitepaper management activities based on over 1000 lines of code. This section discusses the general context around the specific findings elaborated in subsequent sections with an aim to better understand the current findings and prevent future instances.

The stalled voting finding and impacted `raiseTap()` finding both stem from arithmetic-related issues. The former stems from operations with unsigned operands potentially yielding negative results, and the latter stems from a loss of precision due to integer division rounding. New Alchemy strongly recommends continuous review of all current and future arithmetic operations in minute detail for overflow, underflow and precision; as demonstrated, the SafeMath library does not obviate this need for inspection and testing. Whenever possible, perform multiplication prior to division in order to retain maximum precision throughout a calculation.

The logic bug in `WhitepaperVersioning.sol` likely stems from the draft nature of this code. While it is difficult to get the operational logic exactly right, it can be even more difficult to elaborate and handle unexpected malicious corner cases. Deep unit-level

The current reuse of standard open-source components allows the platform to greatly reduce risk through leveraging well-reviewed, well-tested and usage-proven functionality. However, the current dependency management approach does not yet tightly connect the reused components back to their source. As a result, this audit found that many components were out of date and did not take advantage of available improvements. This can be easily remedied.

Finally, many of the minor findings appear to involve an incomplete or slightly outdated development tool chain. This is common due to the rapidly changing nature of the smart contract development landscape. New Alchemy strongly recommends staying current on compilers, (multiple) linters, formal methods and test-coverage frameworks. Maximize the leverage these tools provide, utilize their most conservative settings and aim to eliminate as many errors and warnings as possible early in the development process.

Finally, the whitepaper mentions a Dutch (descending price) auction, but this is not yet implemented in the code. Note that these auctions may be subject to front-running attacks. More information is available from a variety of sources, as well as possible work-arounds.

Whitepaper-to-Contract Coherence

This section discusses the coherence between the terms from the whitepaper and the actual contract implementation. Contracts should aim to implement as closely as possible the various descriptions found in the whitepaper and website. The amount of discrepancies found between the whitepapers and the actual contracts help users decide the level of trust they can put into the contracts as they are implemented.

The DAICOVO platform series of smart contracts are part of a much larger system. Further, the contracts themselves do not implement a single specific ICO but are fundamentally intended to be more abstract, configurable and therefore useful for a broad variety of scenarios. The larger abstract system is out of scope.

Comparative analysis of the whitepapers and smart contract source code revealed:

- The presence of standard ERC20 and ERC223 functionality.

- Confirmed proposal deposit stake of `proposalCostWei = 1 * 10**18`.
- Dutch and English auction logic is not yet implemented.
- Rate-limited tap functionality is present.
- Full voting logic is not yet implemented (e.g. VOTING_PERIOD is declared but never used, `start_time` and `end_time` are not initialized).

The correctness of the above functionality is addressed in subsequent sections of this audit.

The critical constants listed in the token whitepaper are not yet hardcoded into the contract. As a result, New Alchemy judges the alignment of whitepaper constants with contract source code to be insufficient.

Critical Issues

1. Fixed: Negative-Outcome Vote Stalls Voting System

In the `Voting.sol` contract, the `finalizeVoting()` function is used to finalize a vote that is past its `end_time` and will call the `isPassed()` function to determine the outcome of the vote. The relevant code is excerpted below.

```
function finalizeVoting () external {
    uint256 pid = this.getCurrentVoting();
    require(proposals[pid].end_time <= block.timestamp);
    require(!proposals[pid].isFinalized);

    proposals[pid].isFinalized = true;

    if (isPassed(pid)) {
        if (isSubjectRaiseTap(pid)) {

DaicoPool(poolAddr).raiseTap(proposals[pid].tapMultiplierRate);
            queued[uint(Subject.RaiseTap)] = false;
        } else if (isSubjectDestruction(pid)) {
    }
}

function isPassed (uint256 pid) public constant returns(bool) {
    require(proposals[pid].isFinalized);
    uint256 ayes = getAyes(pid);
    uint256 nays = getNays(pid);
    uint256 absent =
ERC20Interface(votingTokenAddr).totalSupply().sub(ayes).sub(nays);
    return (ayes.sub(nays).add(absent.div(6)) > 0);
}
```

At the end of the `isPassed()` function, the `return` statement tests for the difference between `ayes` and `nays` using a subtraction performed with `safeMath`. If the number of `nays` is greater than the number of `ayes` the call to `sub()` will assert an error which will revert the whole transaction. This reverting of the transaction will undo all the prior changes that were performed in the transaction, including setting `isFinalized` to true. Since no more votes can be cast and the outcome won't change, the contract enters an implicit stalled state where the proposal can not be finalized, the next proposal cannot be put to a vote and the tokens used to vote on this specific proposal cannot be returned to their owners (and are stuck forever in this contract).

Separately, in the case where the `sub()` function does not assert, the outcome of the `return` statement will almost always be positive since it tests for an unsigned integer (`uint256`) value to be greater than 0 (e.g. the only case where this returns false is where this value is exactly 0).

To prevent stalling the contract, the `isPassed` function `return` statement needs to be changed to the following:

```
return (ayes.add(absent.div(6)) > nays);
```

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has been fixed and the associated risk mitigated. The `isPassed()` function has been recast as shown below:

```
    uint256 ayes = getAyes(pid);
    uint256 nays = getNays(pid);
    uint256 absent =
ERC20Interface(votingTokenAddr).totalSupply().sub(ayes).sub(nays);
    return (ayes > nays.add(absent.div(6)));
}
```

Moderate Issues

2. Fixed: Loss of Precision Defeats raiseTap Granularity

Upon a positive outcome to a vote on a proposal to "raise tap" from the `Voting.sol` contract, a call to `raiseTap` on the `DaicoPool.sol` contract will be made to increase the `tap` by a value between 1.01 and 2.00 (by allowing the `_tapMultiplierRate` value from `Voting.sol` to range from 101 to 200 included).

```
function raiseTap(uint256 tapMultiplierRate) external onlyVoting {
    updateReleasedBalance();
    updateTap(tap.mul(tapMultiplierRate.div(100)));
}
```

However because of the order of operation and truncating integer math, only two actual multiplier values will ever be used. Either a multiply value of 1 if `tapMultiplierRate` is a value ranging from 101 to 199 or a multiply value of 2 if `tapMultiplierRate` is exactly 200. This defeats the purpose of the vote to increase the `tap` for any value of `tapMultiplierRate` different from 200.

To fix this issue, the order of operations needs to be shifted in the following manner to minimize the intermediate loss of precision from integer math:

```
updateTap(tap.mul(tapMultiplierRate).div(100));
```

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has been fixed and the associated risk mitigated. The operation code now

Open in app Get started

An attacker may be able to stall this contract through a carefully crafted malicious `post` transaction.

Consider the scenario where:

- `WhitepaperVersioning.sol` is freshly deployed and the whitepapers structure is empty.
- An attacker submits a `post(_ipfsHash="HEAD", _version=255, _prev="HEAD")` transaction. The logic bug revolves around `_ipfsHash="HEAD"` which may be an unexpected (or undesired) input with a version parameter equal to `MAX_INT8`.
- This transaction creates a `whitepapers["HEAD"]` entry with version equal to 255.

The code pertinent to this scenario is excerpted below:

```
function post (string _ipfsHash, uint8 _version, string _prev)
public ... {
  // Check if the IPFS hash doesn't exist already.
  require(!whitepapers[_ipfsHash].initialized);

  // Check if the specified version is counted up
  require(_version > whitepapers[_prev].version);

  // Check if previous whitepaper's author is identical to posting
whitepaper's
  // author or the posting whitepaper is the initial version (HEAD
version)
  require(keccak256(_prev) == keccak256("HEAD") ||
          whitepapers[_prev].author == msg.sender);
  // Check if there is no fork from the previous version
  require(bytes(whitepapers[_prev].next).length == 0);

  whitepapers[_prev].next = _ipfsHash;
  whitepapers[_ipfsHash] = Whitepaper(_version, msg.sender, _prev,
"", true);
  emit Post(_ipfsHash, _version, msg.sender);
  return true;
}
```

Any subsequent post transaction seeking to create a legitimate fresh entry (which

Open in app Get started

recommends this contract be more thoroughly tested (including this scenario) and the logic around `"HEAD"` revisited.

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has been fixed and the associated risk mitigated. ICOVO has effectively rewritten the algorithm to manage whitepaper tracking.

4. Not Fixed: Unbounded Loops

A number of contracts include `for` loops that iterate over each element in a data structure of indeterminate size. As each iteration consumes additional gas, a large data structure may cause function execution to be impacted by block gas limits. This may ultimately result in an unusable contract. Scenario risk is heightened if an attacker is able to directly increase the data structure length and/or if each loop iteration involves significant gas cost.

The following table shows instances of contract, line and loop specifics for this issue:

```
IndividuallyCapped...sol:39:  for (uint256 i = 0; i <
_beneficiaries.length; i++)
WhitelistedCrowdsale.sol:42:  for (uint256 i = 0; i <
_beneficiaries.length; i++)
TokenSaleManager.sol:126:     for (uint256 i = 0; i <
tokenSales.length; i++ )
TokenSaleManager.sol:133:     for (uint256 i = 0; i <
tokenSales.length; i++ )
TokenSaleManager.sol:160:     for (uint256 i = 0; i <
tokenSales.length; i++ )
TimeLockPool.sol:90:          for (uint256 i = 0; i <
lockedBalances[account]

[tokenAddr].length; i++)
TimeLockPool.sol:128:         for (uint256 i = 0; i <
lockedBalances[account]

[tokenAddr].length; i++)
TimeLockPool.sol:148:         for (uint256 i = 0; i <
lockedBalances[account]

[tokenAddr].length; i++)
TimeLockPool.sol:168:         for (uint256 i = 0; i <
lockedBalances[account]
```

```
Voting.sol:119:                for (uint256 i = 0; i <
accounts.length; i++)
Voting.sol:125:                for (uint256 i = 0; i <
proposals.length; i++)
```

In some cases it may be preferable to keep running counts during operation such that a summary function does not have to loop over an entire data structure to calculate total counts. In other cases it may be preferable to invert control by exposing helper functions to support external operations. In rare cases, an additional 'helper' data structure may be required.

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has not been fixed. A specific instance is described in the following paragraph.

Re-test v2.0: An attacker can greatly expand the `lockedBalances` data structure in `TimeLockPool.sol` by making a large number of small deposits via the `depositERC20()` function with distinct `releaseTime` values. When the victim subsequently attempts to call the `getAvailableBalanceOf()` function, its unbounded loop on line 131 may run out of gas before finishing the walk through the `lockedBalances` data structure. As a result, this function effectively no longer works and any other functionality that depends on this function (such as a withdrawal process) may no longer work either.

5. Informational: Inadequate Dependency Management

The primary contracts make excellent use of standardized functionality. However, a robust strategy for version control and dependency management is lacking. As a result, it is difficult to definitively determine file versions, heritage and freshness. This will impact vulnerability awareness and resolution.

The `crowdsale` directory provides an illustrative example. These files contain standardized functionality sourced from the OpenZeppelin project. Since some details have been injected into the files themselves as comments, it is now no longer possible to compare file hashes with the source project. Further, it is difficult to track the emergence of new vulnerabilities and their applicability. Many of the line-by-line comments noted in a later section of this report have already been fixed in

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has not been addressed. However, it has been marked as informational because it relates to best practices and not to a specific security bug.

6. Fixed: Unused Contract Code

The `MintedCrowdsale.sol` and `IndividuallyCappedCrowdsale.sol` source files are included in the project but are never used. While these are likely the result of normal development refactoring work, unused files provide the potential for confusion and mistakes.

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has been fixed and the associated risk mitigated. The two source files noted above have been removed.

Minor Issues

7. Fixed: Potential for Tokens and ETH Lock-up

In `TimeLockPool.sol` , both deposit functions (`depositERC20()` and `depositETH()`) allow the destination account to be set to an address of `0x0` . However, the `withdraw()` function requires `account` to be different from `0x0` for the transaction to succeed. This means that any tokens or ether sent to the `0x0` address will stay locked in the contract forever with no way to recover them. New Alchemy recommends adding a requirement to both deposit functions so that they check that `account` is different from the `0x0` address to avoid loss of funds or tokens.

```
require(account != address(0x0));
```

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has been fixed and the associated risk mitigated. The two functions noted above now check for the `0x0` address via the `require` statement specified above.

8. Not Fixed: Constant Return Values

`true` and it is only called via a require statement in the `TokenSaleManager.sol::mintTimeLocked()` function, which is also hardcoded to always return `true`. This may be inadvertently misleading or confusing when read or used out of context.

Examples of contracts, line numbers and actual excerpt are shown below:

```
MintableToken.sol:42:          return true;
MintableToken.sol:52:          return true;
ERC223Standard.sol:74:         return true;
ERC20Standard.sol:37:          return true;
ERC20Standard.sol:55:          return true;
ERC20Standard.sol:71:          return true;
ERC20Standard.sol:113:         return true;
ERC20Standard.sol:134:         return true;
TokenController.sol:79:        return true;
TokenController.sol:90:        return true;
WhitepaperVersioning.sol:52:   return true;
DaicovoStandardToken.sol:35:   return true;
TokenSaleManager.sol:104:      return true;
TokenSaleManager.sol:121:      return true;
TimeLockPool.sol:61:           return true;
TimeLockPool.sol:79:           return true;
TimeLockPool.sol:107:          return true;
TimeLockPool.sol:112:          return true;
```

New Alchemy recommends avoiding constant return values by checking both positive and negative conditions and returning the appropriate value. Subsequently, ensure all return values are checked and handled. Where this is not possible, the function should not specify a return value.

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has not been fixed. Many functions continue to return hardcoded values as shown, including the scenario in the introductory paragraph above.

9. Not Fixed: ERC20 double-spend attack

The standard ERC20 interface, implemented in `ERC20Token`, has a design flaw: if some user Alice wants to change the allowance that it grants to some other user Bob, then spend both the pre-change and post-change allowances. 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, it may be necessary for the attacker to wait until the transaction to change the allowance is issued, and then issue a spend transaction with an unusually high gas price to ensure that the spend transaction is mined before the allowance change. More detail on this flaw is 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, safe usage of the standard ERC20 `approve()` interface to change allowances requires that:

1. allowances only change between zero and non-zero
2. allowances change at most once in a block

These restrictions allow 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, then finally setting the allowance to the new value. Strictly requiring only 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.

To make it more convenient to safely change allowances, New Alchemy recommends providing `increaseApproval()` and `decreaseApproval()` functions that add or subtract to the existing allowances rather than overwriting them. These functions are indeed included in the `ERC20Standard.sol` contract -- which is excellent. This effectively moves the check of whether Bob has spent his 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()` .

However, the contracts under review do not utilize `increaseApproval()` nor `decreaseApproval()` but instead continue use of `approve()` in `TokenSaleManager.sol` .

Open in app Get started

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. This logic has not been implemented in the source contracts under review.

Since both approaches are outside of the ERC20 standard, both approaches require user cooperation to work properly. Accordingly, ICOVO should provide documentation advising its users on how ERC20 allowances are safely managed.

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has not been fixed. The `TokenSaleManager.sol` contract continues to use the `approve()` function. New Alchemy reviewed the latest available whitepapers and was unable to located specific directions around ERC20 allowances.

10. Not Fixed: Lack of two-phase ownership transfer

In contracts that inherit the common `Ownable` contract from the OpenZeppelin project, 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.

In order to preclude this, New Alchemy recommends implementing 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;
        _;
    }

    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: New Alchemy has inspected the new code and has determined that this issue has not been fixed. However, ICOVO does not stand out from the community for choosing not to fix this.

11. Not Fixed: Lack of short-address attack protections

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", 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 DAICOVO contracts, such methods include the following:

Open in app Get started

```
  _value) exte...
  ERC20Standard.sol:29:       function transfer(address _to, uint256
  _value) exte...
  ERC20Standard.sol:46:       function transferFrom(address _from,
  address _to, u...
  ERC20Standard.sol:68:       function approve(address _spender,
  uint256 _value) ...
  ERC20Standard.sol:96:       function allowance(address _owner,
  address _spender...
  ERC20Standard.sol:110:      function increaseApproval(address
  _spender, uint _a...
  TokenController.sol:50:     function mint (address to, uint256
  amount) external...
  DaicovoStandardT...sol:28:  function forceTransfer(address _to, uint
  _value) ex...
  TokenSaleManager.sol:101:   function mint (address _beneficiary,
  uint256 _token...
```

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 `ERC20Standard::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 test 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

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has not been fixed. As stated above, ICOVO does not stand out from the community because they have chosen not to fix this.

12. Informational: Incorporate Assertions

The contracts do not incorporate any assertions (outside of the `SafeMath` library) to check contract invariants. New Alchemy recommends incorporating assertions near the end of functions to confirm expected contract state wherever possible, particularly when balances are being adjusted.

One example is line 100 of `DaicoPool.sol::withdraw()` where several calculations are made involving balances followed by a transfer. An assertion relating to the expected final state could potentially be incorporated here. The `TimeLockPool.sol` contract has multiple opportunities along the same lines.

Re-test v2.0: New Alchemy has inspected the new code and has determined that this issue has not been addressed. However, it has been marked as informational because it relates to best practices and not to a specific security bug.

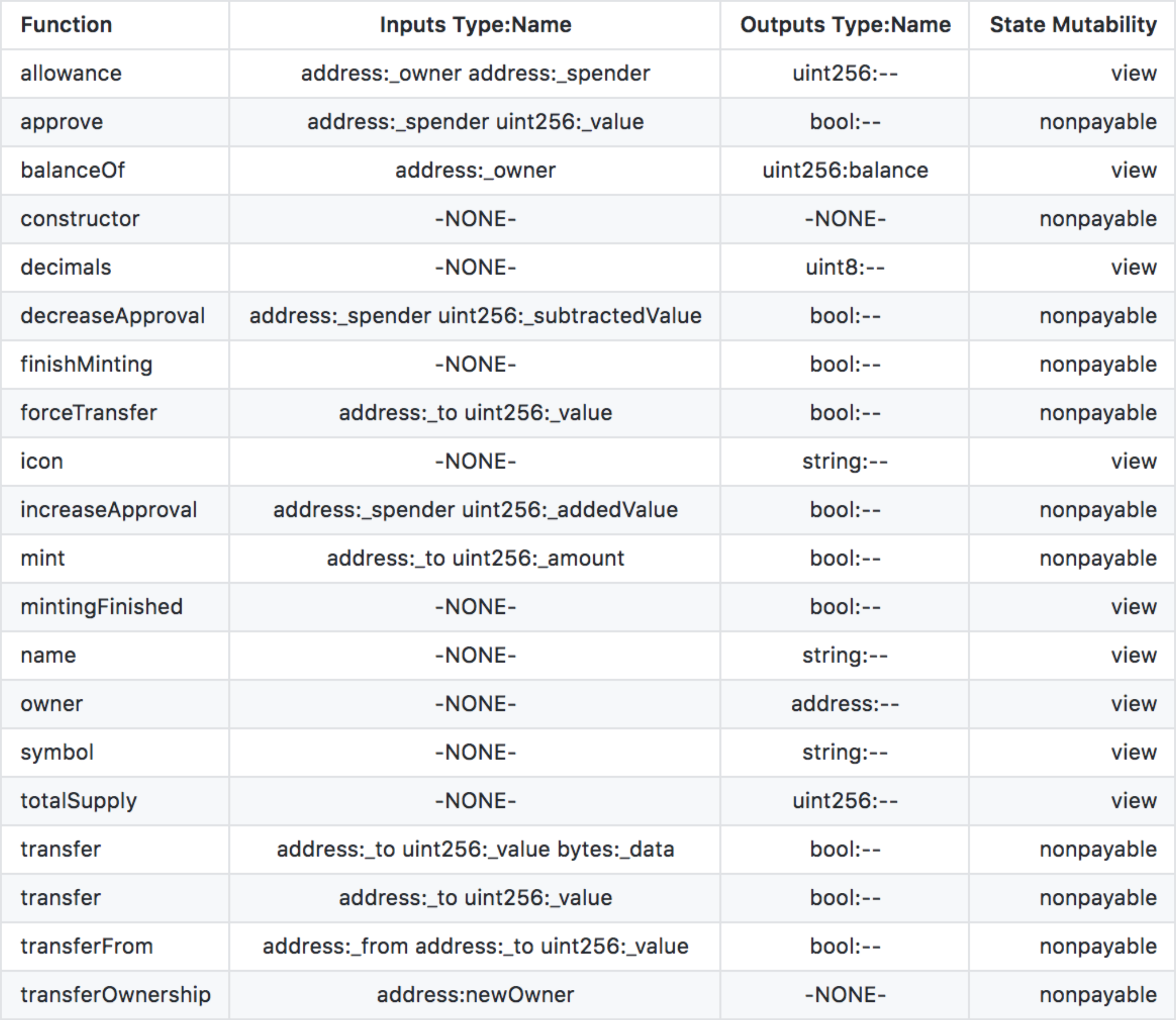
Smart Contract ABI Reference

This section provides the fully-elaborated application binary interface (ABI) for four of the primary contracts as seen at runtime. The ABI defines how the contracts may be interacted with while running inside the EVM. The tables make for very easy inspection of top-level specifics including:

- The definitive reference of all exposed functions
- Correct input and output function signatures
- Presence of constructors and fallbacks as expected
- Correct state mutability (pure, view, nonpayable, payable)
- Extraneous functions which should be private

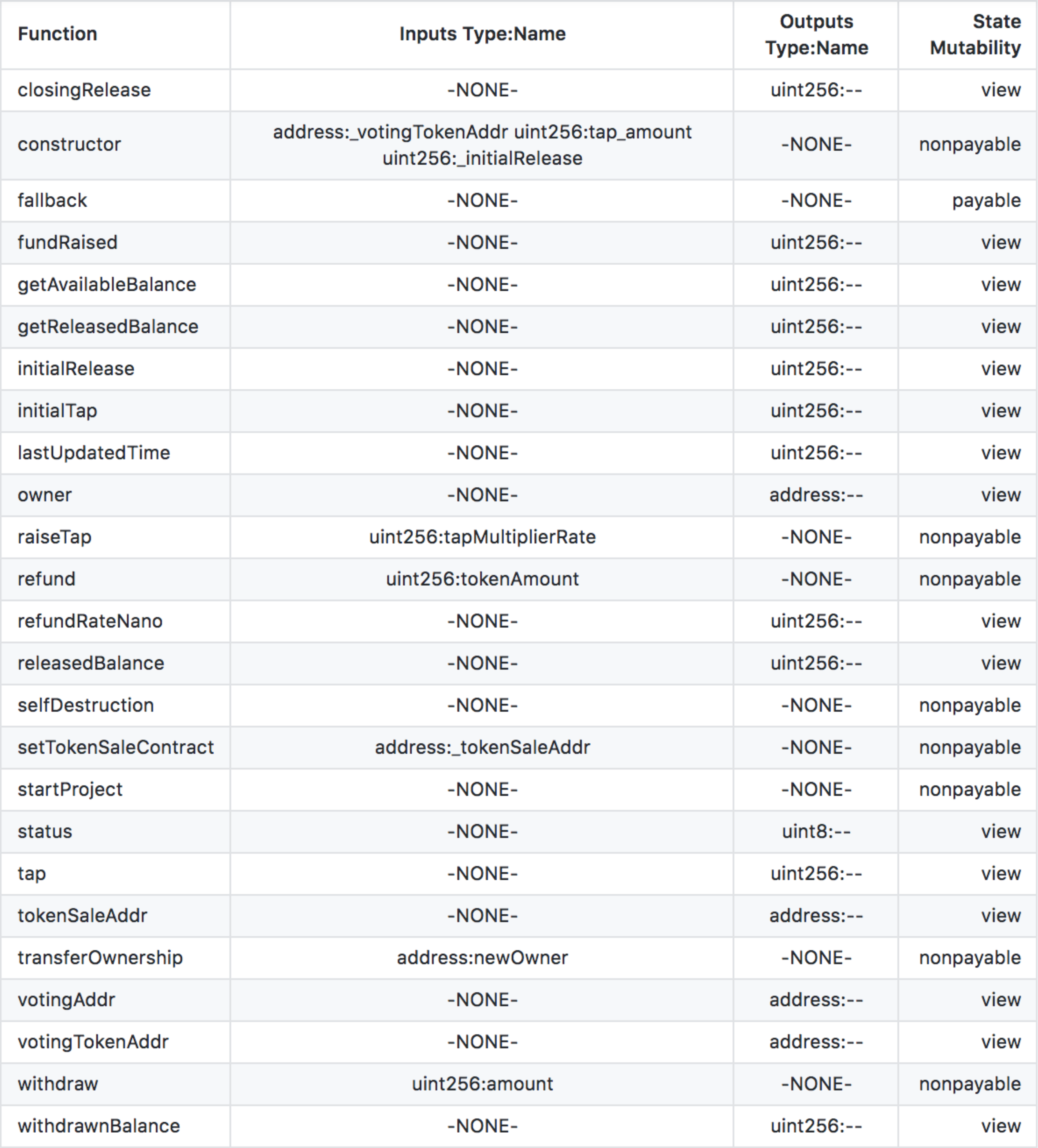
From inspecting the tables below it becomes clear that many return values have no function return (output) values may also represent missed opportunities to report then check/handle status above and beyond simple transaction failures.

`OVOToken.sol` ABI Reference



Function	Inputs Type:Name	Outputs Type:Name	State Mutability
allowance	address:_owner address:_spender	uint256:--	view
approve	address:_spender uint256:_value	bool:--	nonpayable
balanceOf	address:_owner	uint256:balance	view
constructor	-NONE-	-NONE-	nonpayable
decimals	-NONE-	uint8:--	view
decreaseApproval	address:_spender uint256:_subtractedValue	bool:--	nonpayable
finishMinting	-NONE-	bool:--	nonpayable
forceTransfer	address:_to uint256:_value	bool:--	nonpayable
icon	-NONE-	string:--	view
increaseApproval	address:_spender uint256:_addedValue	bool:--	nonpayable
mint	address:_to uint256:_amount	bool:--	nonpayable
mintingFinished	-NONE-	bool:--	view
name	-NONE-	string:--	view
owner	-NONE-	address:--	view
symbol	-NONE-	string:--	view
totalSupply	-NONE-	uint256:--	view
transfer	address:_to uint256:_value bytes:_data	bool:--	nonpayable
transfer	address:_to uint256:_value	bool:--	nonpayable
transferFrom	address:_from address:_to uint256:_value	bool:--	nonpayable
transferOwnership	address:newOwner	-NONE-	nonpayable

`DaicoPool.sol` ABI Reference

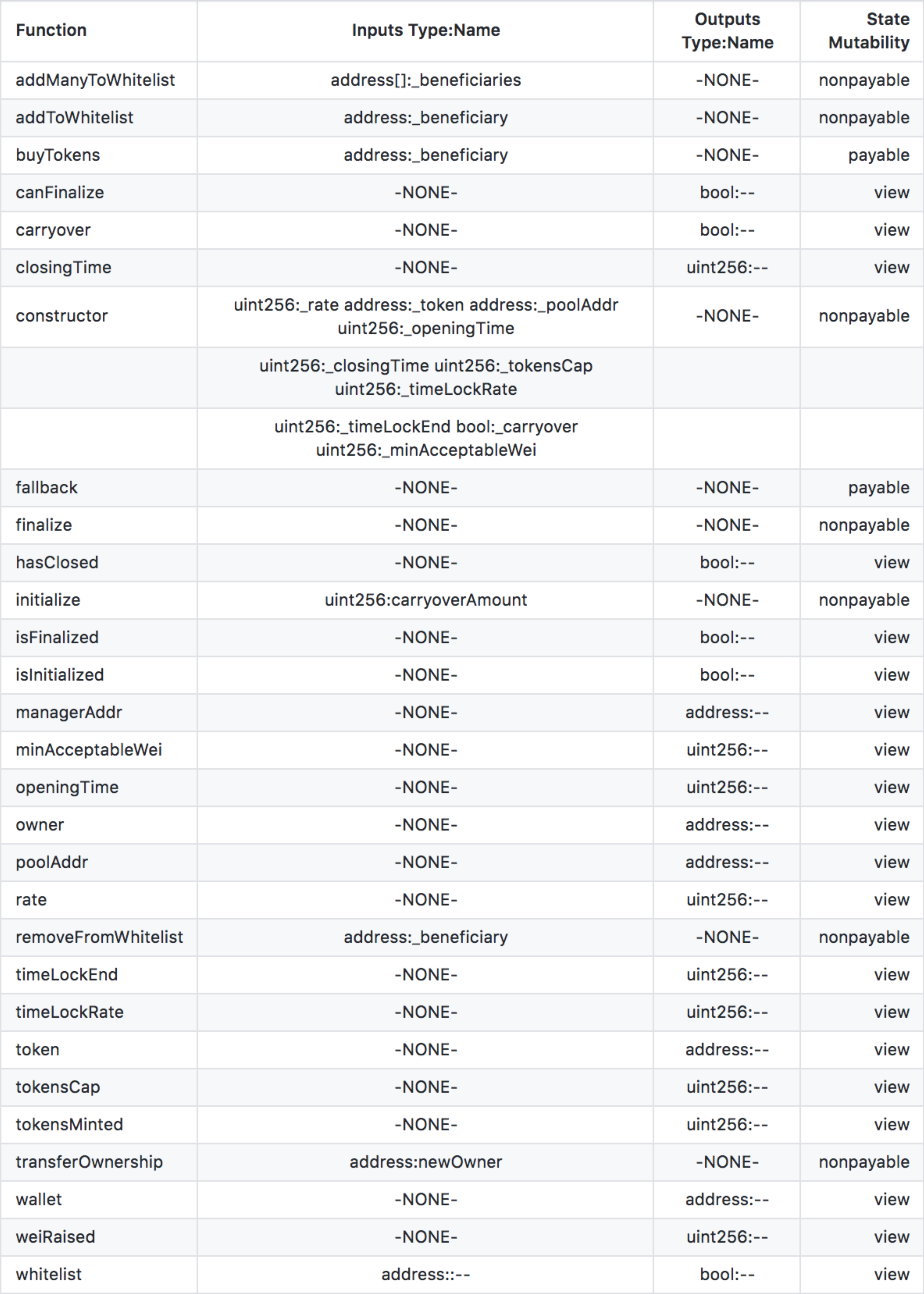


Function	Inputs Type:Name	Outputs Type:Name	State Mutability
closingRelease	-NONE-	uint256:--	view
constructor	address:_votingTokenAddr uint256:tap_amount uint256:_initialRelease	-NONE-	nonpayable
fallback	-NONE-	-NONE-	payable
fundRaised	-NONE-	uint256:--	view
getAvailableBalance	-NONE-	uint256:--	view
getReleasedBalance	-NONE-	uint256:--	view
initialRelease	-NONE-	uint256:--	view
initialTap	-NONE-	uint256:--	view
lastUpdatedTime	-NONE-	uint256:--	view
owner	-NONE-	address:--	view
raiseTap	uint256:tapMultiplierRate	-NONE-	nonpayable
refund	uint256:tokenAmount	-NONE-	nonpayable
refundRateNano	-NONE-	uint256:--	view
releasedBalance	-NONE-	uint256:--	view
selfDestruction	-NONE-	-NONE-	nonpayable
setTokenSaleContract	address:_tokenSaleAddr	-NONE-	nonpayable
startProject	-NONE-	-NONE-	nonpayable
status	-NONE-	uint8:--	view
tap	-NONE-	uint256:--	view
tokenSaleAddr	-NONE-	address:--	view
transferOwnership	address:newOwner	-NONE-	nonpayable
votingAddr	-NONE-	address:--	view
votingTokenAddr	-NONE-	address:--	view
withdraw	uint256:amount	-NONE-	nonpayable
withdrawnBalance	-NONE-	uint256:--	view

`TokenSale.sol` ABI Reference

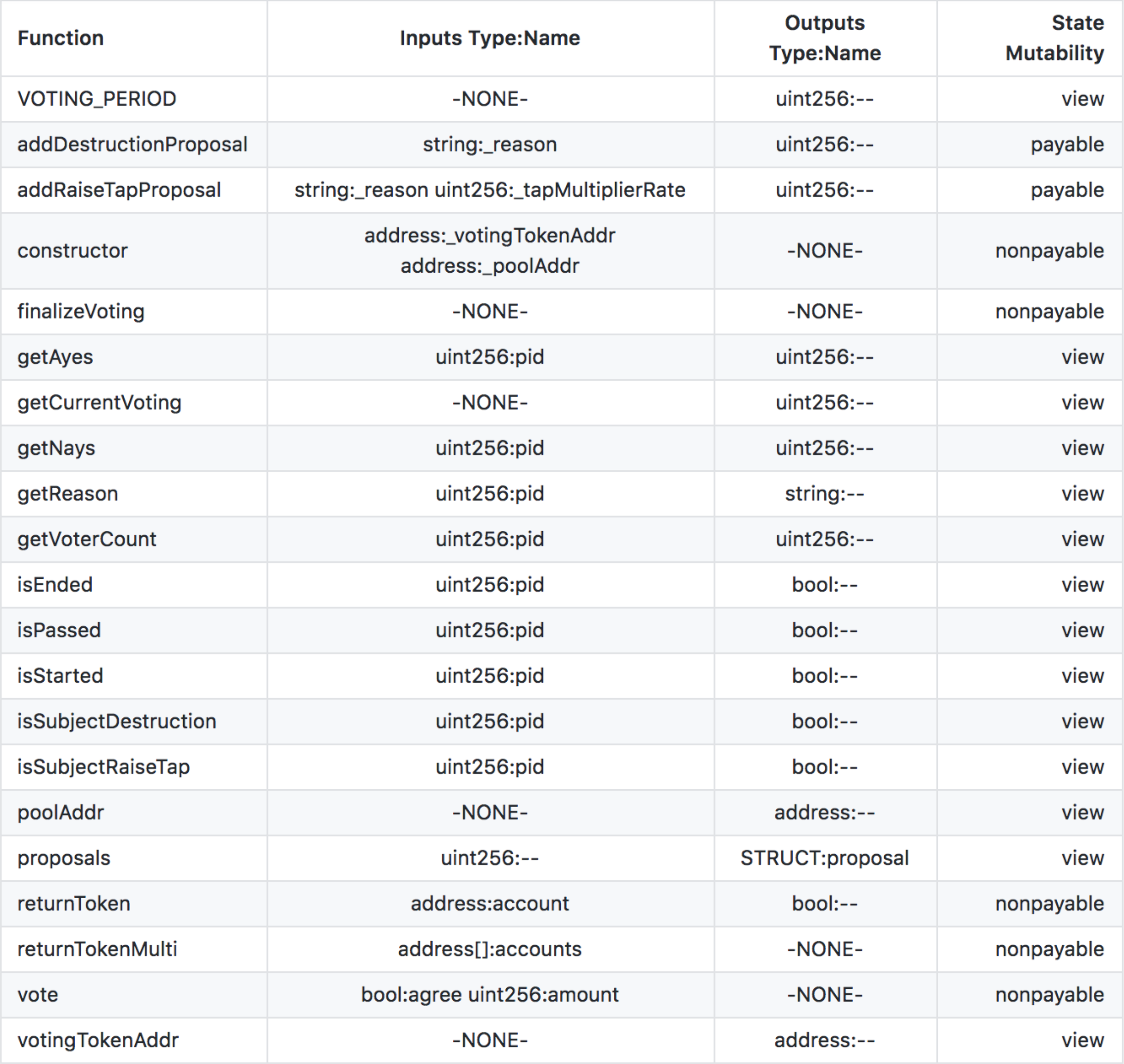
Open in app

Get started



Function	Inputs Type:Name	Outputs Type:Name	State Mutability
addManyToWhitelist	address[]:_beneficiaries	-NONE-	nonpayable
addToWhitelist	address:_beneficiary	-NONE-	nonpayable
buyTokens	address:_beneficiary	-NONE-	payable
canFinalize	-NONE-	bool:--	view
carryover	-NONE-	bool:--	view
closingTime	-NONE-	uint256:--	view
constructor	uint256:_rate address:_token address:_poolAddr uint256:_openingTime	-NONE-	nonpayable
	uint256:_closingTime uint256:_tokensCap uint256:_timeLockRate		
	uint256:_timeLockEnd bool:_carryover uint256:_minAcceptableWei		
fallback	-NONE-	-NONE-	payable
finalize	-NONE-	-NONE-	nonpayable
hasClosed	-NONE-	bool:--	view
initialize	uint256:carryoverAmount	-NONE-	nonpayable
isFinalized	-NONE-	bool:--	view
isInitialized	-NONE-	bool:--	view
managerAddr	-NONE-	address:--	view
minAcceptableWei	-NONE-	uint256:--	view
openingTime	-NONE-	uint256:--	view
owner	-NONE-	address:--	view
poolAddr	-NONE-	address:--	view
rate	-NONE-	uint256:--	view
removeFromWhitelist	address:_beneficiary	-NONE-	nonpayable
timeLockEnd	-NONE-	uint256:--	view
timeLockRate	-NONE-	uint256:--	view
token	-NONE-	address:--	view
tokensCap	-NONE-	uint256:--	view
tokensMinted	-NONE-	uint256:--	view
transferOwnership	address:newOwner	-NONE-	nonpayable
wallet	-NONE-	address:--	view
weiRaised	-NONE-	uint256:--	view
whitelist	address::--	bool:--	view

Open in app
Get started



Function	Inputs Type:Name	Outputs Type:Name	State Mutability
VOTING_PERIOD	-NONE-	uint256:--	view
addDestructionProposal	string:_reason	uint256:--	payable
addRaiseTapProposal	string:_reason uint256:_tapMultiplierRate	uint256:--	payable
constructor	address:_votingTokenAddr address:_poolAddr	-NONE-	nonpayable
finalizeVoting	-NONE-	-NONE-	nonpayable
getAyes	uint256:pid	uint256:--	view
getCurrentVoting	-NONE-	uint256:--	view
getNays	uint256:pid	uint256:--	view
getReason	uint256:pid	string:--	view
getVoterCount	uint256:pid	uint256:--	view
isEnded	uint256:pid	bool:--	view
isPassed	uint256:pid	bool:--	view
isStarted	uint256:pid	bool:--	view
isSubjectDestruction	uint256:pid	bool:--	view
isSubjectRaiseTap	uint256:pid	bool:--	view
poolAddr	-NONE-	address:--	view
proposals	uint256:--	STRUCT:proposal	view
returnToken	address:account	bool:--	nonpayable
returnTokenMulti	address[]:accounts	-NONE-	nonpayable
vote	bool:agree uint256:amount	-NONE-	nonpayable
votingTokenAddr	-NONE-	address:--	view

Line-by-Line Comments

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

There are several items that are pervasive across the code base:

- Compiler `Pragma` . New Alchemy recommends using the `pragma` annotation to latest stable compiler: `pragma solidity 0.4.24`. This issue pertains to every contract in the source set.

- Constructors. Solidity v0.4.22, released in April 2018, introduced a new constructor syntax using the `constructor()` keyword that will become required in the upcoming v0.5.0 release. The practice of matching the constructor name to the contract name has been deprecated. New Alchemy recommends migrating to the new syntax. This issue pertains to every contract in the source set except the following:

- `FinalizableCrowdsale.sol`
- `MintedCrowdsale.sol`
- `IndividuallyCappedCrowdsale.sol`
- `WhitelistedCrowdsale.sol`
- `SafeMath.sol`
- `ERC20Interface.sol`
- `ERC20Standard.sol`
- `ERC223Interface.sol`
- `ERC223ReceivingContract.sol`
- `ERC223Standard.sol`
- `MintableToken.sol`

- Initialization New Alchemy recommends that all contract state variables be explicitly initialized rather than relying upon default values. This eliminates the potential for 'accidentally correct' behavior that can be inadvertently broken in subsequent edits, e.g. enumerated status variables initialized to the first value in a list.

of license terms prevents matching hashes to a specific version. From comparing the latest version, the OZ file includes `using SafeERC20 for ERC20`, and references ERC20 directly rather than an interface.

Line 91

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

Lines 108, 118, 145

The `_preValidatePurchase()`, `_postValidatePurchase()` and `updatePurchasingState()` functions do not read from or modify the contract state. As a result, they may be marked `pure` in this isolated context. However, these functions may be further implemented or overridden in a larger context, so this may change. For now, New Alchemy recommends including the `pure` modifier until its removal is necessary.

B. crowdsale/distribution/FinalizableCrowdsale.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/crowdsale/distribution/FinalizableCrowdsale.sol (OZ), however the insertion of license terms prevents matching hashes to a specific version. From comparing the latest versions, no significant differences were noted with the exception of the items listed below.

Line 35

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

C. crowdsale/emission/MintedCrowdsale.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/crowdsale/emission/MintedCrowdsale.sol (OZ), this is a misspelling of the English word `extensions`). This file appears to be an unused orphan.

D. crowdsale/validation/IndividuallyCappedCrowdsale.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/crowdsale/validation/CappedCrowdsale.sol (OZ), however the insertion of license terms prevents matching hashes to a specific version. From comparing the latest versions, no significant differences were noted. This file appears to be an unused orphan.

E. crowdsale/validation/TimedCrowdsale.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/crowdsale/validation/TimedCrowdsale.sol (OZ), however the insertion of license terms prevents matching hashes to a specific version. From comparing the latest versions, it appears the most significant difference involves the addition of 'lint' directives to the OZ file addressing the comment for line 27, 37 and 49 below.

Line 27, 37, 49

Contract logic involves the `block.timestamp` globally available variable. Note that this value can be manipulated by the block miner by approximately 30 seconds. It does not appear that fundamental contract integrity is dependent upon precise timestamps, but this vulnerability should be kept in mind as the code evolves.

F. crowdsale/validation/WhitelistedCrowdsale.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/crowdsale/validation/WhitelistedCrowdsale.sol however the insertion of license terms prevents matching hashes to a specific version. From inspecting the latest versions, this file has a significantly different code and inheritance structure.

G. contracts/DAICOVO/DaicoPool.sol

Line 22

The `deposits` mapping should have its visibility explicitly specified (defaults to `public`).

Line 60

The constructor requires the voting token address to be nonzero but does not check the tap amount or initial release parameters. New Alchemy recommends adding checks here. Consider adding comments to clarify value units (to avoid confusion).

Lines 83, 127, 143

Contract logic involves the `block.timestamp` globally available variable. Note that this value can be manipulated by the block miner by approximately 30 seconds. It does not appear that fundamental contract integrity is dependent upon precise timestamps, but this vulnerability should be kept in mind as the code evolves.

Line 86

Starting with Solidity v0.5.0, contracts will not derive from the address type. Thus, accessing "balance" inherited from the address type is deprecated. Convert this statement to `fundRaised = address(this).balance`.

Line 94

New Alchemy recommends avoiding assignments that overwrite function parameters, e.g. `amount` is passed into the function as a parameter on line 89, and then potentially assigned a new value on line 94.

Lines 100, 122, 148

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

Line 126, 131

The `constant` function modifier was deprecated in Solidity v0.4.17 released in

Line 34

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

I. DAICOVO/OVOToken.sol

J. DAICOVO/TimeLockPool.sol

Lines 3, 4

Common Solidity coding style convention prefers the usage of double quotes over a single quote for file import statements.

Line 19: Inaccurate Comment

The following mapping contains a table or list of the `LockedBalance` struct, not necessarily a single item.

```
mapping (address => mapping (address => LockedBalance[]))
lockedBalances
```

New Alchemy recommends to remove or update the comment to reflect the current code.

Line 22

The `lockedBalances` mapping should have its visibility explicitly specified (defaults to `public`).

Lines 59, 77, 101

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

There is a function named `withdraw()` and a related event named `Withdraw()` . New Alchemy recommends avoiding names with such close similarity due to the potential for confusion. Consider naming events with a `Log` prefix, e.g. this event becomes `LogWithdraw()` .

Lines 92, 129, 149, 169

Contract logic involves the `block.timestamp` globally available variable. Note that this value can be manipulated by the block miner by approximately 30 seconds. It does not appear that fundamental contract integrity is dependent upon precise timestamps, but this vulnerability should be kept in mind as the code evolves.

Line 101

Consider moving the Withdraw event to the end of the function so that it is only emitted upon a successful transfer. This will require only minor restructuring of the `return true` statements.

Lines 122, 142, 162

The `constant` function modifier was deprecated in Solidity v0.4.17, released in September 2017 and will be dropped in v0.5.0. The `view` keyword has identical meaning and should replace `constant` .

K. DAICOVO/TokenController.sol

Lines 3, 4, 5, 6

Common Solidity coding style convention prefers the usage of double quotes over a single quote for file import statements.

Line 71: implicit onlyOwner

The require statement (`require(msg.sender == owner);`) at this line could be replaced by adding the `onlyOwner` modifier on the function. This would clarify the code.

Lines 95, 101, 107

The `constant` function modifier was deprecated in Solidity v0.4.17, released in

Open in app Get started

L. DAICOVO/TokenSaleManager.sol

Line 17

The `token` variable should have its visibility explicitly specified (defaults to `public`).

M. DAICOVO/TokenSale.sol

Line 55

Common Solidity coding conventions suggest placing the `public` visibility modifier before the `onlyOwner` function modifier.

Line 61

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

Line 66

The `constant` function modifier was deprecated in Solidity v0.4.17 released in September 2017 and will be dropped in v0.5.0. The `view` keyword has identical meaning and should replace `constant`.

N. DAICOVO/Voting.sol

Lines 3, 4, 5

Common Solidity coding style convention prefers the usage of double quotes over a single quote for file import statements.

Lines 13, 14, 16

The `deposits` and `queued` mappings, and the `proposalCostWei` variable should have their visibility explicitly specified (defaults to `public`).

Line 63

The function [illegible] calculates and returns a `uint256` [illegible] ID for assignment

Contract logic involves the `block.timestamp` globally available variable. Note that this value can be manipulated by the block miner by approximately 30 seconds. It does not appear that fundamental contract integrity is dependent upon precise timestamps, but this vulnerability should be kept in mind as the code evolves.

Lines 124, 133, 141, 150, 159, 164, 169, 174, 179, 185

The `constant` function modifier was deprecated in Solidity v0.4.17, released in September 2017, and will be dropped in v0.5.0. The `view` keyword has identical meaning and should replace `constant`.

Lines 159, 174

New Alchemy recommends grouping functions of identical visibility together. The contract follows this practice, with the exception of `getReason()` and `getVoterCount()`.

O. DAICOVO/WhitepaperVersioning.sol

P. math/SafeMath.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol (OZ), however the insertion of license terms prevents matching hashes to a specific version. From comparing the latest versions, no significant differences were noted with the exception of the item listed below.

Q. ownership/Ownable.sol

Notes: This file is derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/ownership/Ownable.sol (OZ), however the insertion of license terms prevents matching hashes to a specific version. From comparing the latest versions, the OZ file includes an additional `Event` and related function allowing ownership to be renounced. This file incorporates a simplified `transferOwnership()` function.

Line 43

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be

R. token/ERC20/ERC20Interface.sol

Notes: This file appears to meet and define the ERC20 interface described athttps://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. However, it does not match any readily available OpenZeppelin code currently available on GitHub.

Lines 14–16

Solidity automatically creates getter functions for declared state variables so these lines are effectively equivalent to three declared functions. However, it is slightly better to instead directly declare the three functions because this allows the actual state variables to remain 'hidden' leaving flexibility for future change. This is sometimes referred to as 'information hiding'.

Line 24

There is a function named `transfer()` and a related event named `Transfer()` . New Alchemy recommends avoiding names with such close similarity due to the potential for confusion. Consider naming events with a `Log` prefix, e.g. this event becomes `LogTransfer()` . In this instance the interface is dictated by the ERC20 standard and so cannot be changed lightly, but this should be kept in mind for future contract development.

S. token/ERC20/ERC20Standard.sol

Notes: Along with the above file, this one appears to diverge from commonly used foundations. Its precise heritage is unclear.

Lines 19, 22

The `balances` mapping and `totalSupply_` variable should have their visibility explicitly specified (defaults to `public`).

Lines 36, 54, 70, 112, 133

Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

Notes: This file appears to be a modified version of https://github.com/Dexaran/ERC223-token-standard/blob/master/token/ERC223/ERC223_interface.sol. However there are significant differences. This file includes additional state variables, function visibility is marked `external`, the `transfer()` functions return a boolean, and 'uints' are specified as `uint256`.

Line 14

The `constant` function modifier was deprecated in Solidity v0.4.17, released in September 2017 and will be dropped in v0.5.0. The `view` keyword has identical meaning and should replace `constant`.

Line 17

There is a function named `transfer()` and a related event named `Transfer()`. New Alchemy recommends avoiding names with such close similarity due to the potential for confusion. Consider naming events with a `Log` prefix, e.g. this event becomes `LogTransfer()`. In this instance the interface is dictated by the ERC223 standard and so cannot be changed lightly, but this should be kept in mind for future contract development.

U. token/ERC223/ERC223ReceivingContract.sol

Notes: This file appears to be a modified version of https://github.com/Dexaran/ERC223-token-standard/blob/master/token/ERC223/ERC223_receiving_contract.sol. The primary difference involves marking the `tokenFallback()` function visibility as `public`.

V. token/ERC223/ERC223Standard.sol

Notes: This file appears to be derived from https://github.com/Dexaran/ERC223-token-standard/blob/master/token/ERC223/ERC223_token.sol. However, there is very significant divergence and the external source file has several issues such as an odd (or broken) filepath for the ‘import SafeMath’ statement.

Lines 46, 73

become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

W. token/extentions/MintableToken.sol

Notes: This file appears to be derived from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/MintableToken.sol. However, the insertion of license terms prevents matching hashes to a specific version. There is significant divergence between these files due to intended ERC20 vs ERC223 usage.

Line 37

There is a function named `mint()` and a related event named `Mint()`. New Alchemy recommends avoiding names with such close similarity due to the potential for confusion. Consider naming events with a `Log` prefix, e.g. this event becomes `LogMint()`. In this instance the interface is dictated by the ERC20 standard and so cannot be changed lightly, but this should be kept in mind for future contract development.

Lines 37, 49

Common Solidity coding conventions suggest placing the `public` visibility modifier before the `onlyOwner` and `canMint` function modifiers.

Lines 40, 41, 51

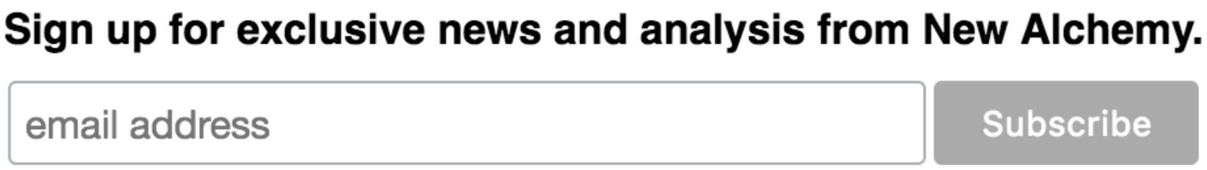
Solidity v0.4.21, released in March 2018, added support for the `emit` keyword to be added prior to `Event()` invocations to help differentiate events from functions. This will become required in v0.5.0 and the current absence of this keyword is deprecated syntax. New Alchemy recommends migrating to the new syntax.

Lines 42, 53

The `mint()` and `finishMinting()` functions are defined to return a boolean indicating successful function completion. However, the return value is hardcoded to `true`.

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 the fitness of the contracts to purpose, or their bug-free status. The audit documentation is for discussion purposes only.

New Alchemy is a leading blockchain strategy and technology group specializing in tokenized capital solutions for the most innovative companies worldwide. **New Alchemy's Blockchain Security** division is a highly trusted name that has assisted clients in securely raising over $500m through custom-tailored solutions, smart contract audits and comprehensive security strategy. ***Get in touch with us at Hello@NewAlchemy.io***



Sign up for exclusive news and analysis from New Alchemy.

email address

Subscribe